CSE344 – System Programming Spring 2024

Final Project Report

Alperen Duran 200104004024

In this Project ı implemented Pide Shop server-client simulation using multiple threads to efficiently manage and process client orders in a concurrent environment. The server utilizes socket programming to handle client-server communication, and threading to parallelize the tasks of order processing and delivery.

Design

Server Side

1. **Server Initialization**:
   - The server initializes by parsing command-line arguments for configuration settings such as IP address, port, thread pool sizes, oven capacity, and delivery speed.
   - It sets up the server socket, binds it to the specified IP and port, and prepares to listen for incoming client connections.
2. **Client Handling**:
   - Each client connection is accepted and handled in a separate thread. The client sends an order, which is then read and parsed to extract necessary details such as client ID, location, and order specifics.
   - The order is then placed in a queue for processing.
3. **Order Processing**:
   - Orders are managed in a queue system, with separate queues for pending orders and ready orders.
   - Cook threads retrieve orders from the pending orders queue, simulate preparation by performing computational tasks (such as calculating the pseudo-inverse of a matrix), and then place the prepared orders into the oven.
   - Once cooked, the orders are moved to the ready orders queue.

4. **Order Delivery**:
   - Delivery personnel threads pick up orders from the ready orders queue, simulate delivery by calculating the distance to the client's location and sleeping for a duration proportional to the distance.
   - Once delivered, the client is notified, and the order is marked as completed.
5. **Successful Shutdown**:
   - The server includes signal handling to manage shutdowns gracefully. Upon receiving a SIGINT (Ctrl+C) signal, the server ensures all active orders are completed before terminating.

Client Side

1. **Initialization**:

- The client starts by parsing command-line arguments to retrieve the server's IP address, the number of orders to be placed, maximum x and y coordinates for order locations, and the optional port number.
- It sets up signal handling to capture SIGINT (Ctrl+C) for graceful shutdown.
- The random number generator is seeded to ensure varied order locations.

2. **Server Communication**:

- The client establishes an initial connection to the server to send the total number of orders.
- For each order, the client generates a random location and establishes a new connection to send the order details to the server.

3. **Order Handling**:

- Each order's location is determined using the get_random_location function, which generates a random coordinate within the specified bounds.
- Orders are sent in a loop, where for each order, the client connects to the server, sends the order details, and then starts a thread to listen for server responses.

4. **Multi-threaded Listening**:

- A dedicated listener thread is created for each order to handle incoming messages from the server. This thread reads server messages and logs them to a file.
- The main thread waits for the listener thread to finish processing before moving to the next order.

5. **Successful Shutdown**:

- If a SIGINT signal is received, the client sets a stop flag and sends a shutdown message to the server.
- The client ensures all ongoing orders are properly cancelled and logs the shutdown process.

6. **Logging and Final Notification**:

- After all orders are processed or cancelled, the client logs a summary to a file.
- It sends a final message to the server indicating that all customers have been served.

Algorithm

Server Side

Initialization:

- Parse input arguments to configure server settings.
- Set up server socket and bind to specified IP and port.
- Initialize thread pools for cooks and delivery personnel.

Client Handling:

- Accept incoming client connections.
- Create a new thread for each client to handle order reading and parsing.
- Place parsed orders into the pending orders queue.

Cook Threads:

- Wait for orders in the pending orders queue.
- Process orders by simulating preparation (e.g., calculating matrix pseudo-inverse).
- Move orders to the oven and then to the ready orders queue.

Delivery Threads:

- Wait for orders in the ready orders queue.
- Collect and deliver orders, simulating delivery time based on distance.
- Notify clients upon successful delivery.

Shutdown Process:

- Handle SIGINT signal to initiate shutdown.
- Ensure all orders in the queue are processed before terminating the server.
- Close server socket and log the shutdown process.

Client Side

Initialization:

- Parse input arguments to configure client settings.
- Set up signal handling for graceful shutdown.
- Seed the random number generator.

Initial Server Communication:

- Establish a connection to the server.
- Send the total number of orders to the server.

Order Placement:

- Loop through the number of orders.
- For each order:
    - Generate a random location.
    - Establish a connection to the server.
    - Send order details to the server.
    - Create a listener thread to handle server responses.
    - Wait for the listener thread to finish.

Listener Thread:

- Read server messages.
- Log messages to a file.
- Terminate after receiving all messages for an order.

Graceful Shutdown:

- On receiving SIGINT, set the stop flag.
- Send a shutdown message to the server.
- Ensure all active orders are cancelled and logged.

Final Logging and Notification:

- Log a summary to a file.
- Send a final message to the server indicating all customers are served.

Detailed Explanation of Key Threads

```c
void *handle_client(void *arg)
{
    client_t *client = (client_t *)arg;
    char buffer[1024] = {0};
    read(client->socket, buffer, 1024); // Read the order from the client

    // Parse the client PID from the buffer
    sscanf(buffer, "Client PID: %d", &client->pid);

    int num_orders;
    if (sscanf(buffer, "%d", &num_orders) == 1)
    {
        printf("%d new customers... serving\n", num_orders);
        fflush(stdout);
        fprintf(log_file, "%d new customers... serving\n", num_orders);
        close(client->socket);
        free(client);
        pthread_exit(NULL);
    }
    if (strncmp(buffer, "Ctrl+C received by client", 25) == 0)
    {
        printf("order cancelled PID %d\n", client->pid); // Print message on server side
        fflush(stdout);
        fprintf(log_file, "order cancelled PID %d\n", client->pid);
        fflush(log_file);
        printf("^C.. Upps quiting.. writing log file\n");
        fflush(stdout);
        close(client->socket);
        free(client);
        pthread_exit(NULL);
    }

    if (strncmp(buffer, "All customers served", 20) == 0)
    {
        // Find the most efficient cook
        int most_efficient_cook = 0;
        int max_cook_orders = 0;
        for (int i = 0; i < cook_thread_pool_size; i++)
        {
            if (cook_order_count[i] > max_cook_orders)
            {
                max_cook_orders = cook_order_count[i];
                most_efficient_cook = i;
            }
        }

        // Find the most efficient delivery person
        int most_efficient_delivery = 0;
        int max_delivery_orders = 0;
        for (int i = 0; i < delivery_thread_pool_size; i++)
        {
            if (delivery_order_count[i] > max_delivery_orders)
            {
                max_delivery_orders = delivery_order_count[i];
```

```c
            {
                max_delivery_orders = delivery_order_count[i];
                most_efficient_delivery = i;
            }
        }

        printf("Thanks Cook %d and Moto %d\n", most_efficient_cook, most_efficient_delivery);
        fflush(stdout);
        fprintf(log_file, "Thanks Cook %d and Moto %d\n", most_efficient_cook, most_efficient_delivery);
        fflush(log_file);
        printf("active waiting for connections\n");
        fflush(stdout);
        fprintf(log_file, "active waiting for connections\n");
        fflush(log_file);

        close(client->socket);
        free(client);
        pthread_exit(NULL);
    }

    fprintf(log_file, "Received order: %s\n", buffer);
    fflush(log_file);

    sscanf(buffer, "Order from client %d (PID: %d) at location (%d, %d)", &client->client_id, &client->pid, &client->location_x, &client->location_y);

    pthread_mutex_lock(&lock);
    order_t order = {client->client_id, client->location_x, client->location_y, -1, -1, client->pid, client->socket};
    enqueue_order(order_queue, &order_queue_start, &order_queue_end, order);
    total_orders++;
    pending_orders++;
    pthread_cond_signal(&order_cond);
    pthread_mutex_unlock(&lock);

    // Send order placed confirmation message to the client
    char confirmation_message[256];
    snprintf(confirmation_message, sizeof(confirmation_message), "Order for client %d (PID: %d) placed successfully", client->client_id, client->pid);
    send(client->socket, confirmation_message, strlen(confirmation_message), 0);

    while (pending_orders > 0)
    {
        sleep(1);
    }

    close(client->socket);
    free(client);
    pthread_exit(NULL);
}
```

The `handle_client` thread function manages communication with a client in a multi-threaded Pide Shop server. It reads the client's order details from the socket, including the client PID, and processes different types of messages: the number of new orders, order cancellation, or all orders being served. For new orders, it parses the order location, logs the details, and enqueues the order for processing by cooks and delivery personnel. It also finds and acknowledges the most efficient cook and delivery person when all orders are served. Throughout the process, it sends confirmation messages back to the client and ensures proper cleanup by closing the socket and freeing memory before exiting.

```c
void *cook_thread(void *arg)
{
    int cook_id = *(int *)arg;
    free(arg);

    while (1)
    {
        pthread_mutex_lock(&lock);
        while (is_queue_empty(order_queue_start, order_queue_end) && !stop_flag)
        {
            pthread_cond_wait(&order_cond, &lock);
        }
        if (stop_flag)
        {
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }

        order_t order = dequeue_order(order_queue, &order_queue_start, &order_queue_end);
        order.cook_id = cook_id;
        cook_order_count[cook_id]++; // Increment the cook's order count
        pthread_mutex_unlock(&lock);

        fprintf(log_file, "Cook %d received order from client %d (PID: %d) for location (%d, %d)\n", cook_id, order.cli
        fflush(log_file);

        // Simulate preparation time by calculating the pseudo-inverse of a 30x40 matrix
        double matrix[MATRIX_ROWS * MATRIX_COLS];
        double pseudo_inv[MATRIX_COLS * MATRIX_ROWS];

        // Initialize matrix with random values
        for (int i = 0; i < MATRIX_ROWS * MATRIX_COLS; i++)
        {
            matrix[i] = rand() / (double)RAND_MAX;
        }

        pseudo_inverse(matrix, MATRIX_ROWS, MATRIX_COLS, pseudo_inv);

        // Send order prepared message to the client
        char prepared_message[256];
        snprintf(prepared_message, sizeof(prepared_message), "Order for client %d (PID: %d) has been prepared by cook %
        send(order.client_socket, prepared_message, strlen(prepared_message), 0);

        // Place the meal in the oven
        pthread_mutex_lock(&lock);
        while (oven_count >= OVEN_CAPACITY)
        {
            pthread_cond_wait(&oven_cond, &lock);
        }
        oven_count++;
        pthread_mutex_unlock(&lock);

        // Simulate cooking time
        sleep(1);
        char cook_message[256];
        snprintf(cook_message, sizeof(cook_message), "Order for client %d (PID: %d) has been cooked by cook %d.", order
        send(order.client_socket, cook_message, strlen(cook_message), 0);
        // Meal is ready, remove from oven
        pthread_mutex_lock(&lock);
        oven_count--;
        enqueue_order(ready_queue, &ready_queue_start, &ready_queue_end, order);
        pthread_cond_signal(&oven_cond);
        pthread_cond_signal(&ready_cond);
        pthread_mutex_unlock(&lock);

        fprintf(log_file, "Cook %d prepared and placed pide for client %d (PID: %d) at location (%d, %d)\n", cook_id, o
        fflush(log_file);
    }

    pthread_exit(NULL);
}
```

The `cook_thread` function in the Pide Shop server simulates the work of a cook by continuously processing orders in an infinite loop. Each cook thread waits for new orders, and once an order is received, it logs the details, simulates preparation by calculating a pseudo-inverse of a matrix, and sends a preparation confirmation to the client. The thread then manages oven space, ensuring that the meal is cooked by waiting if the oven is full, and after cooking, it logs the completion, updates the oven status, and signals that the order is ready for delivery. This process repeats, allowing the cook to handle multiple orders efficiently until the server is stopped.

```c
void *delivery_thread(void *arg)
{
    int delivery_id = *(int *)arg;
    free(arg);

    while (1)
    {
        pthread_mutex_lock(&lock);

        // Collect up to DELIVERY_CAPACITY orders
        order_t delivery_orders[DELIVERY_CAPACITY];
        int order_count = 0;

        while (order_count < DELIVERY_CAPACITY && !is_queue_empty(ready_queue_start, ready_queue_end))
        {
            delivery_orders[order_count] = dequeue_order(ready_queue, &ready_queue_start, &ready_queue_end);
            delivery_orders[order_count].delivery_id = delivery_id;
            order_count++;
        }

        // Wait if there are no orders to deliver
        while (order_count == 0 && !stop_flag)
        {
            pthread_cond_wait(&ready_cond, &lock);
            while (order_count < DELIVERY_CAPACITY && !is_queue_empty(ready_queue_start, ready_queue_end))
            {
                delivery_orders[order_count] = dequeue_order(ready_queue, &ready_queue_start, &ready_queue_end);
                delivery_orders[order_count].delivery_id = delivery_id;
                order_count++;
            }
        }

        if (stop_flag)
        {
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }

        pthread_mutex_unlock(&lock);

        // Deliver the collected orders
        for (int i = 0; i < order_count; i++)
        {
            order_t order = delivery_orders[i];
            double distance = findSQRT(order.location_x * order.location_x + order.location_y * order.location_y);
            sleep(distance / delivery_speed);

            fprintf(log_file, "Delivered to location (%d, %d) by delivery person %d for client %d (PID: %d)\n", order.location_
            fflush(log_file);

            printf("Done serving client %d (PID: %d) @ (%d, %d)\n", order.client_id, order.pid, order.location_x, order.locatio
            fflush(stdout);

            // Send acknowledgment to client
            int client_socket = order.client_socket;
            char ack_message[1024];
            sprintf(ack_message, "Order for client %d (PID: %d) has been delivered by delivery person %d.", order.client_id, or
            send(client_socket, ack_message, strlen(ack_message), 0);
            close(client_socket);

            pthread_mutex_lock(&lock);
            delivery_order_count[delivery_id]++; // Increment the delivery person's order count
            pending_orders--;
            if (pending_orders == 0)
            {
                all_orders_serviced = 1;
                pthread_cond_signal(&order_cond); // Wake up the main thread if waiting
            }
            pthread_mutex_unlock(&lock);
        }
    }

    pthread_exit(NULL);
}
```

The delivery_thread function handles the delivery of orders in the Pide Shop server by continuously checking for ready orders and delivering them. Each delivery thread waits for up to DELIVERY_CAPACITY orders to become available in the ready queue, and if none are available, it waits for the condition signal indicating new orders are ready. Once orders are collected, the thread calculates the distance to each delivery location, simulates the delivery time based on this distance, logs the delivery details, sends an acknowledgment to the client, and updates the delivery count. If all orders are serviced, it signals the main thread. This loop ensures efficient and continuous order deliveries until the server stops.

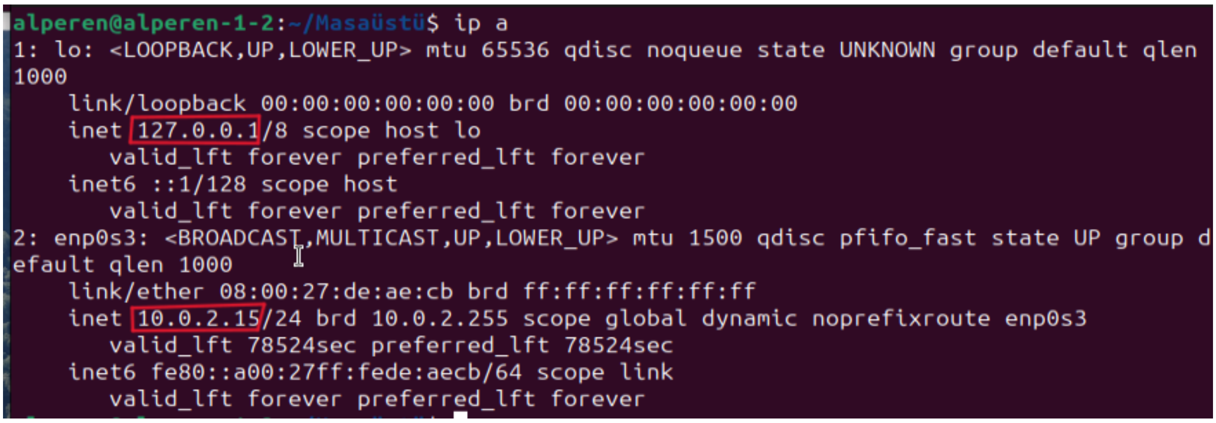
Test Cases

To run the program you should first learn the ip address.

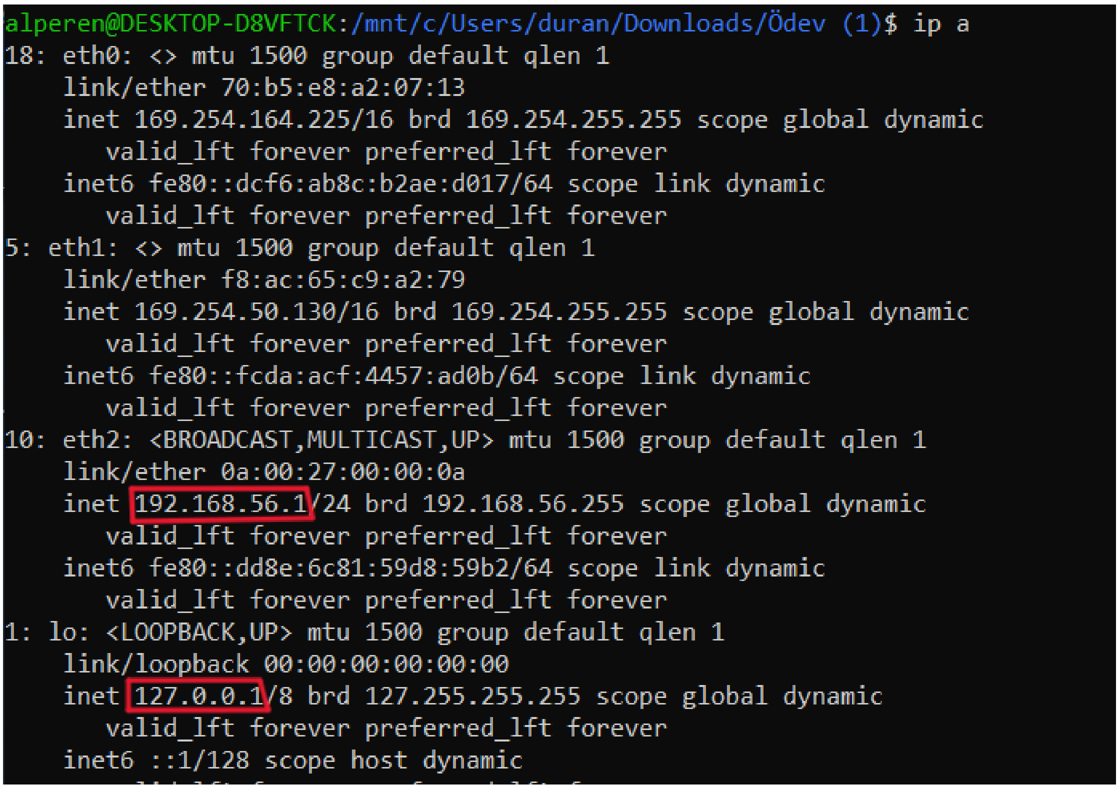
127.0.0.1 for localhost.

10.0.2.15 for VM IP address.

192.168.56.1 is Windows IP addres to run the program on different computers.



```
alperen@alperen-1-2:~/Masaüstü$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group d
efault qlen 1000
    link/ether 08:00:27:de:ae:cb brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 78524sec preferred_lft 78524sec
    inet6 fe80::a00:27ff:fede:aecb/64 scope link
       valid_lft forever preferred_lft forever
```

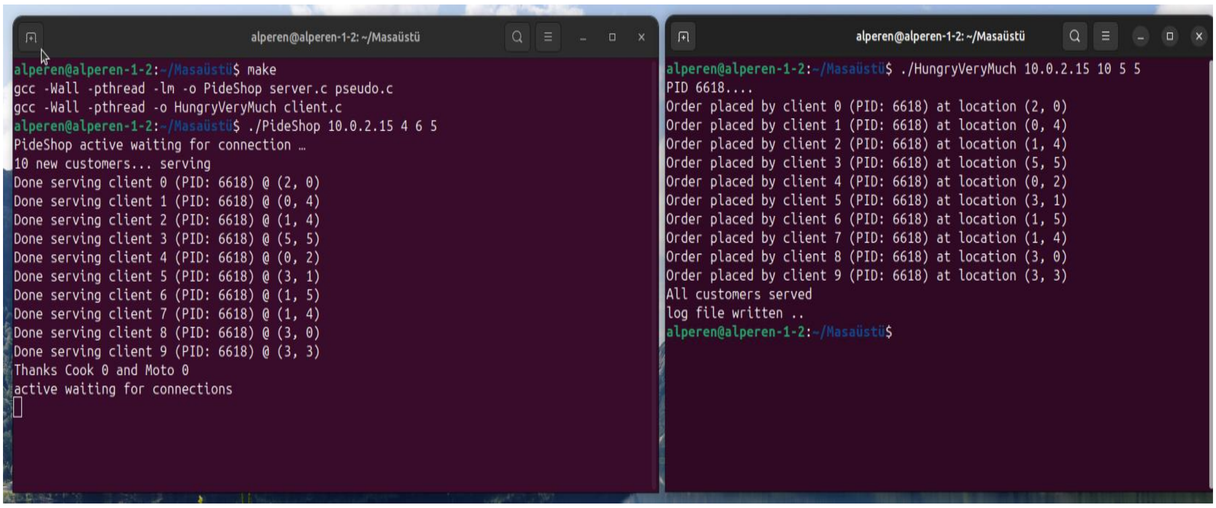


```
alperen@DESKTOP-D8VFTCK:/mnt/c/Users/duran/Downloads/Ödev (1)$ ip a
18: eth0: <> mtu 1500 group default qlen 1
    link/ether 70:b5:e8:a2:07:13
    inet 169.254.164.225/16 brd 169.254.255.255 scope global dynamic
       valid_lft forever preferred_lft forever
    inet6 fe80::dcf6:ab8c:b2ae:d017/64 scope link dynamic
       valid_lft forever preferred_lft forever
5: eth1: <> mtu 1500 group default qlen 1
    link/ether f8:ac:65:c9:a2:79
    inet 169.254.50.130/16 brd 169.254.255.255 scope global dynamic
       valid_lft forever preferred_lft forever
    inet6 fe80::fcda:acf:4457:ad0b/64 scope link dynamic
       valid_lft forever preferred_lft forever
10: eth2: <BROADCAST,MULTICAST,UP> mtu 1500 group default qlen 1
    link/ether 0a:00:27:00:00:0a
    inet 192.168.56.1/24 brd 192.168.56.255 scope global dynamic
       valid_lft forever preferred_lft forever
    inet6 fe80::dd8e:6c81:59d8:59b2/64 scope link dynamic
       valid_lft forever preferred_lft forever
1: lo: <LOOPBACK,UP> mtu 1500 group default qlen 1
    link/loopback 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope global dynamic
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host dynamic
```

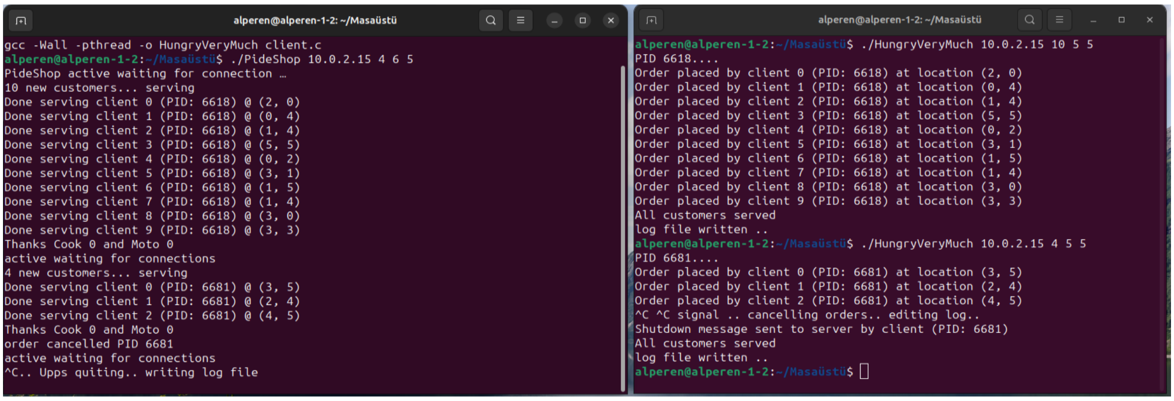
Usage <IP> <cook_thread_pool_size> <oven_capacity> <delivery_speed> [<port>]

- Port is optional if you don’t enter port number it assign as 8080.

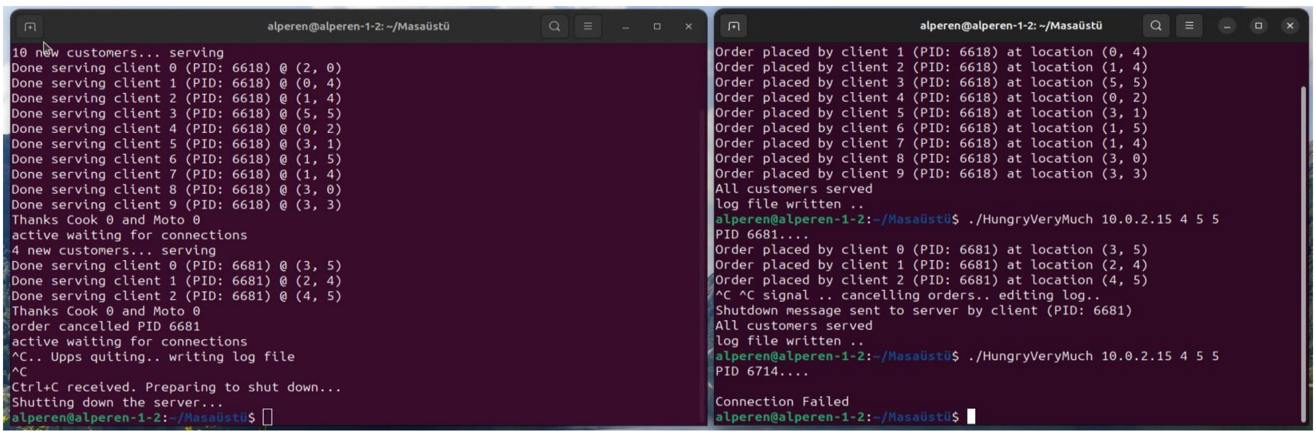
Case 1: Example terminal ouputs for 10 client.



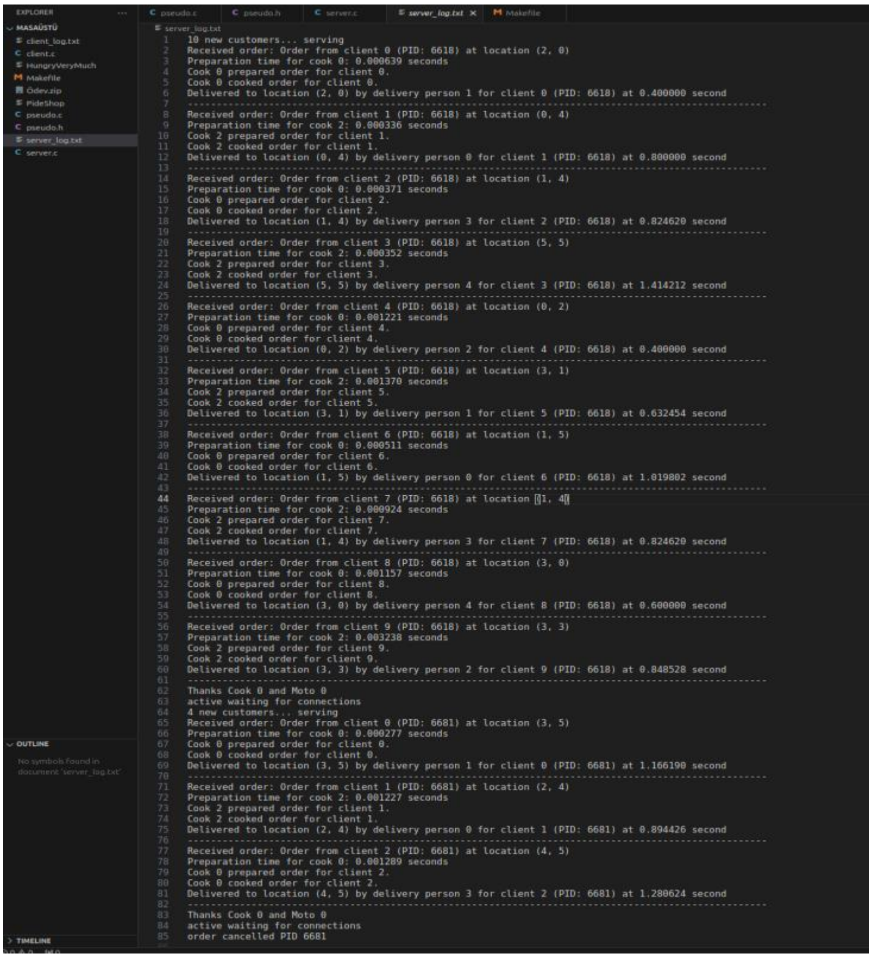
Case 2: Order cancelletion with ctrl^C.



Case 3: Server shutdown with ctrl^C. After server shuts down client can not connect to the server.

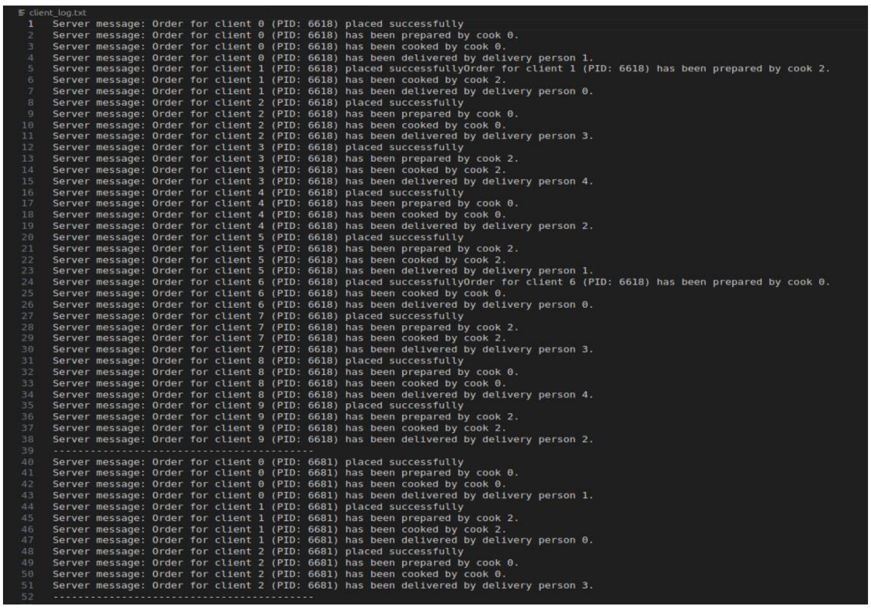


Example server log file. It print all information about order (including preparation time and delivery time) and the server.



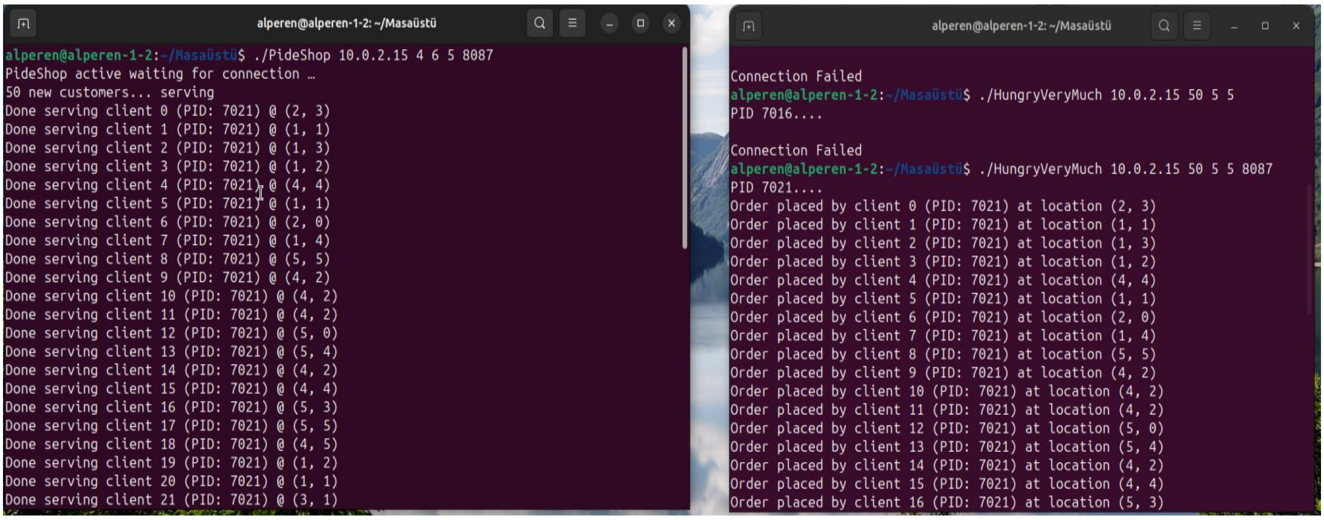
```
10 new customers... serving
Received order: Order from client 0 (PID: 6618) at location (2, 0)
Preparation time for cook 0: 0.000639 seconds
Cook 0 prepared order for client 0.
Cook 0 cooked order for client 0.
Delivered to location (2, 0) by delivery person 1 for client 0 (PID: 6618) at 0.400000 second
----------------------------------------------------------------------------------------------
Received order: Order from client 1 (PID: 6618) at location (0, 4)
Preparation time for cook 2: 0.000336 seconds
Cook 2 prepared order for client 1.
Cook 2 cooked order for client 1.
Delivered to location (0, 4) by delivery person 0 for client 1 (PID: 6618) at 0.800000 second
----------------------------------------------------------------------------------------------
Received order: Order from client 2 (PID: 6618) at location (1, 4)
Preparation time for cook 0: 0.000371 seconds
Cook 0 prepared order for client 2.
Cook 0 cooked order for client 2.
Delivered to location (1, 4) by delivery person 3 for client 2 (PID: 6618) at 0.824620 second
----------------------------------------------------------------------------------------------
Received order: Order from client 3 (PID: 6618) at location (5, 5)
Preparation time for cook 2: 0.000352 seconds
Cook 2 prepared order for client 3.
Cook 2 cooked order for client 3.
Delivered to location (5, 5) by delivery person 4 for client 3 (PID: 6618) at 1.414212 second
----------------------------------------------------------------------------------------------
Received order: Order from client 4 (PID: 6618) at location (0, 2)
Preparation time for cook 0: 0.001221 seconds
Cook 0 prepared order for client 4.
Cook 0 cooked order for client 4.
Delivered to location (0, 2) by delivery person 2 for client 4 (PID: 6618) at 0.400000 second
----------------------------------------------------------------------------------------------
Received order: Order from client 5 (PID: 6618) at location (3, 1)
Preparation time for cook 2: 0.001370 seconds
Cook 2 prepared order for client 5.
Cook 2 cooked order for client 5.
Delivered to location (3, 1) by delivery person 1 for client 5 (PID: 6618) at 0.632454 second
----------------------------------------------------------------------------------------------
Received order: Order from client 6 (PID: 6618) at location (1, 5)
Preparation time for cook 0: 0.000511 seconds
Cook 0 prepared order for client 6.
Cook 0 cooked order for client 6.
Delivered to location (1, 5) by delivery person 0 for client 6 (PID: 6618) at 1.019802 second
----------------------------------------------------------------------------------------------
Received order: Order from client 7 (PID: 6618) at location (1, 4)
Preparation time for cook 2: 0.000924 seconds
Cook 2 prepared order for client 7.
Cook 2 cooked order for client 7.
Delivered to location (1, 4) by delivery person 3 for client 7 (PID: 6618) at 0.824620 second
----------------------------------------------------------------------------------------------
Received order: Order from client 8 (PID: 6618) at location (3, 0)
Preparation time for cook 0: 0.001157 seconds
Cook 0 prepared order for client 8.
Cook 0 cooked order for client 8.
Delivered to location (3, 0) by delivery person 4 for client 8 (PID: 6618) at 0.600000 second
----------------------------------------------------------------------------------------------
Received order: Order from client 9 (PID: 6618) at location (3, 3)
Preparation time for cook 2: 0.003238 seconds
Cook 2 prepared order for client 9.
Cook 2 cooked order for client 9.
Delivered to location (3, 3) by delivery person 2 for client 9 (PID: 6618) at 0.848528 second
----------------------------------------------------------------------------------------------
Thanks Cook 0 and Moto 0
active waiting for connections
4 new customers... serving
Received order: Order from client 0 (PID: 6681) at location (3, 5)
Preparation time for cook 0: 0.000277 seconds
Cook 0 prepared order for client 0.
Cook 0 cooked order for client 0.
Delivered to location (3, 5) by delivery person 1 for client 0 (PID: 6681) at 1.166190 second
----------------------------------------------------------------------------------------------
Received order: Order from client 1 (PID: 6681) at location (2, 4)
Preparation time for cook 2: 0.001227 seconds
Cook 2 prepared order for client 1.
Cook 2 cooked order for client 1.
Delivered to location (2, 4) by delivery person 0 for client 1 (PID: 6681) at 0.894426 second
----------------------------------------------------------------------------------------------
Received order: Order from client 2 (PID: 6681) at location (4, 5)
Preparation time for cook 0: 0.001289 seconds
Cook 0 prepared order for client 2.
Cook 0 cooked order for client 2.
Delivered to location (4, 5) by delivery person 3 for client 2 (PID: 6681) at 1.280624 second
----------------------------------------------------------------------------------------------
Thanks Cook 0 and Moto 0
active waiting for connections
order cancelled PID 6681
```

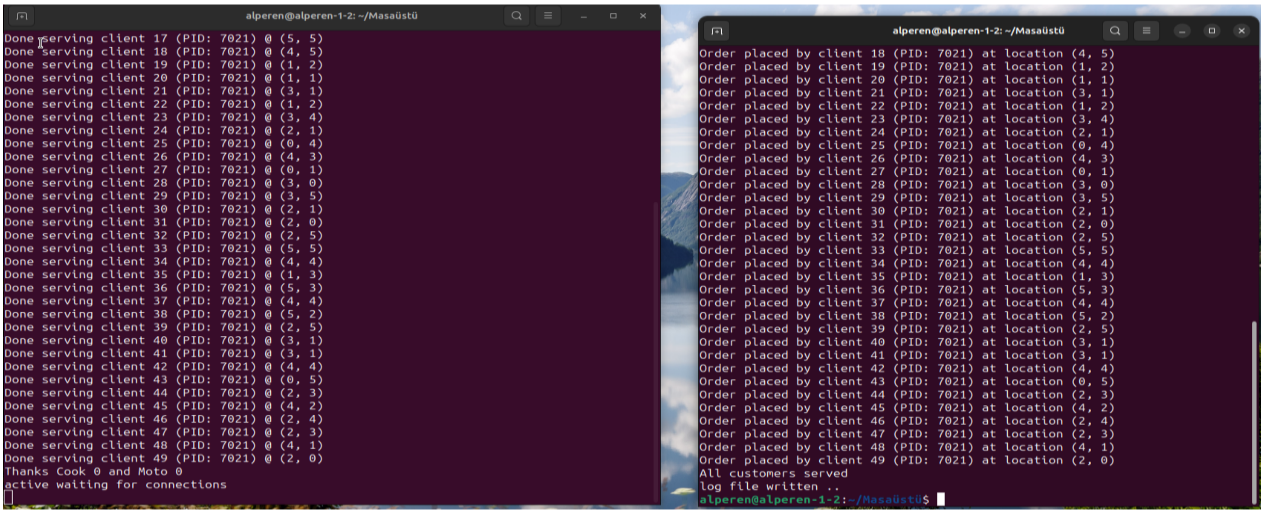
Example client log file it prints information messages when order placed, prepared, cooked and delivered.



```
Server message: Order for client 0 (PID: 6618) placed successfully
Server message: Order for client 0 (PID: 6618) has been prepared by cook 0.
Server message: Order for client 0 (PID: 6618) has been cooked by cook 0.
Server message: Order for client 0 (PID: 6618) has been delivered by delivery person 1.
Server message: Order for client 1 (PID: 6618) placed successfullyOrder for client 1 (PID: 6618) has been prepared by cook 2.
Server message: Order for client 1 (PID: 6618) has been cooked by cook 2.
Server message: Order for client 1 (PID: 6618) has been delivered by delivery person 0.
Server message: Order for client 2 (PID: 6618) placed successfully
Server message: Order for client 2 (PID: 6618) has been prepared by cook 0.
Server message: Order for client 2 (PID: 6618) has been cooked by cook 0.
Server message: Order for client 2 (PID: 6618) has been delivered by delivery person 3.
Server message: Order for client 3 (PID: 6618) placed successfully
Server message: Order for client 3 (PID: 6618) has been prepared by cook 2.
Server message: Order for client 3 (PID: 6618) has been cooked by cook 2.
Server message: Order for client 3 (PID: 6618) has been delivered by delivery person 4.
Server message: Order for client 4 (PID: 6618) placed successfully
Server message: Order for client 4 (PID: 6618) has been prepared by cook 0.
Server message: Order for client 4 (PID: 6618) has been cooked by cook 0.
Server message: Order for client 4 (PID: 6618) has been delivered by delivery person 2.
Server message: Order for client 5 (PID: 6618) placed successfully
Server message: Order for client 5 (PID: 6618) has been prepared by cook 2.
Server message: Order for client 5 (PID: 6618) has been cooked by cook 2.
Server message: Order for client 5 (PID: 6618) has been delivered by delivery person 1.
Server message: Order for client 6 (PID: 6618) placed successfullyOrder for client 6 (PID: 6618) has been prepared by cook 0.
Server message: Order for client 6 (PID: 6618) has been cooked by cook 0.
Server message: Order for client 6 (PID: 6618) has been delivered by delivery person 0.
Server message: Order for client 7 (PID: 6618) placed successfully
Server message: Order for client 7 (PID: 6618) has been prepared by cook 2.
Server message: Order for client 7 (PID: 6618) has been cooked by cook 2.
Server message: Order for client 7 (PID: 6618) has been delivered by delivery person 3.
Server message: Order for client 8 (PID: 6618) placed successfully
Server message: Order for client 8 (PID: 6618) has been prepared by cook 0.
Server message: Order for client 8 (PID: 6618) has been cooked by cook 0.
Server message: Order for client 8 (PID: 6618) has been delivered by delivery person 4.
Server message: Order for client 9 (PID: 6618) placed successfully
Server message: Order for client 9 (PID: 6618) has been prepared by cook 2.
Server message: Order for client 9 (PID: 6618) has been cooked by cook 2.
Server message: Order for client 9 (PID: 6618) has been delivered by delivery person 2.
-----------------------------------------
Server message: Order for client 0 (PID: 6681) placed successfully
Server message: Order for client 0 (PID: 6681) has been prepared by cook 0.
Server message: Order for client 0 (PID: 6681) has been cooked by cook 0.
Server message: Order for client 0 (PID: 6681) has been delivered by delivery person 1.
Server message: Order for client 1 (PID: 6681) placed successfully
Server message: Order for client 1 (PID: 6681) has been prepared by cook 2.
Server message: Order for client 1 (PID: 6681) has been cooked by cook 2.
Server message: Order for client 1 (PID: 6681) has been delivered by delivery person 0.
Server message: Order for client 2 (PID: 6681) placed successfully
Server message: Order for client 2 (PID: 6681) has been prepared by cook 0.
Server message: Order for client 2 (PID: 6681) has been cooked by cook 0.
Server message: Order for client 2 (PID: 6681) has been delivered by delivery person 3.
-----------------------------------------
```

Case 4: Connection with different port number and huge number of clients.





Case 5: Connection on different machines with ctrl^C signal.