

Online Auction System using Client-Server Architecture

Introduction

The online auction system is a client-server application designed to facilitate online auctions for various items. The system allows clients to bid on items and retrieve auction status in real-time. With the increasing demand for online auction systems, it is essential to develop a secure, efficient, and reliable platform that can handle a large number of bidders and provide accurate results. The Auction System aims to address this need by providing a robust and scalable solution for conducting online auctions. This documentation provides an overview of the system's design, implementation, and functionality. This documentation provides an overview of the system's design, implementation, and functionality.

Objective

The primary objective of the Auction System is to provide a secure, efficient, and reliable platform for conducting online auctions. The system aims to ensure the integrity and accuracy of the bidding process, while also providing a user-friendly interface for bidders to place their bids. The specific objectives of the Auction System are:

- To develop a secure and reliable online auction system that can handle a large number of bidders.
- To provide a user-friendly interface for bidders to place their bids
- To ensure the accuracy and integrity of the bidding process
- To provide real-time updates of auction status
- To handle multiple client connections concurrently using multi-threading

System Requirements

1. Hardware Requirements

- Operating System: Linux/Unix-based
- Processor: Multi-core processor
- Memory: 4 GB RAM or more
- Storage: 10 GB or more

2. Software Requirements

- Programming Language: C
- Libraries: sys/socket.h, netinet/in.h, arpa/inet.h, unistd.h, pthread.h
- Compiler: GCC

Functionality

- Bid Placement: Clients can place bids on specific items.
- Auction Management: The server manages the auction process, updating the highest bid and bidder for each item.
- Auction Status Retrieval: Clients can retrieve the current auction status.
- Real-time Updates: The server updates the auction status in real-time as new bids are placed.
- Status Update: Clients can request the current status of an item, including the highest bid and bidder.
- Concurrency: The server handles multiple client connections concurrently using multi-threading.
- Winning Notification: The server notifies the winning bidder when the auction is closed.

Modules

In this project, we have two modules:

1. Server module
2. Client module

1. Server module

The server is responsible for maintaining the auction status and handling client connections. It uses a multi-threaded approach to handle multiple client connections concurrently. Initializes a list of auction items, each item with a name, highest bid, and highest bidder. To handle client requests, it processes commands from clients, including placing bids, checking the status of items, list the items, ending the auction and final results. For maintain

concurrency it uses pthreads to handle multiple client connections concurrently. For synchronization it utilizes mutexes to ensure that bid updates are thread-safe and prevent data races.

Socket Programming: The server creates a socket and binds it to a specific address and port. It listens for incoming client connections and accepts them.

Multi-Threading: The server creates a new thread for each incoming client connection. Each thread handles a single client connection and executes the `handle_client()` function.

Mutexes and Condition Variables: The server uses a mutex (lock) to protect access to the auction status map (items). It also uses a condition variable to notify waiting threads when a new bid is placed.

Auction Status Map: The server maintains a map (items) to store the auction status. The map is updated whenever a new bid is placed.

2.Client module

The client is responsible for bidding on items and retrieving auction status. It connects to the server using socket programming and sends requests to bid on items or retrieve auction status and final results. Here for place bids, sends bid commands to the server in the format. For check status, it requests the current status of all auction items. And also, it implements end Auction and display the final results. When give exit command, it closes the connection with the server.

Socket Connection:

Create and connect to the server socket. The client creates a socket and connects to the server's socket. It sends requests to place bids or retrieve auction status.

User Interaction:

Provide a command-line interface for user input. Handle commands for placing bids, checking status, listing items and ending the auction.

Data Communication:

Send commands to the server. Receive and display responses from the server.

Implementation

In this project the implementation is done using C programming, and use several components.

Socket programming:

The server creates a socket and binds it to a specific address and port. Clients create sockets and connect to the server's socket. Both the server and clients use `send()` and `recv()` to exchange data. The client is implemented using sockets to communicate with the server.

Multi-threading:

The server creates a new thread for each incoming client connection using `pthread_create()`. Each thread handles a single client connection and executes the `handle_client()` function. The server is implemented using POSIX threads (pthreads) to handle multiple client connections concurrently.

Mutexes and condition variables:

The server uses a mutex (lock) to protect access to the auction status map (items). When a client places a bid, the server locks the mutex, updates the auction status, and notifies waiting threads using the condition variable.

Deadlock prevention:

The server avoids deadlocks by locking the mutex before accessing the shared resource (auction status map). The server also avoids nested locks, which can lead to deadlocks.

Testing

Unit Testing: Each module is tested individually to ensure that it functions correctly.

Integration Testing: The modules are integrated and tested to ensure that the system functions as a whole.

System Testing: The system is tested with multiple clients to ensure that it can handle concurrent connections and auctions.

Source code

ServerCode.c

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

#define PORT 8080
#define MAX_CLIENTS 10
#define ITEM_COUNT 3

typedef struct {
    char name[256];
    int highestBid;
    char highestBidder[256];
} Item;

Item items[ITEM_COUNT];

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int client_id = 1; // Initialize the client ID counter

typedef struct {
    int socket;
    int id;
} ClientData;

void* handle_client(void* arg) {
    ClientData *client_data = (ClientData*)arg;
    int client_sock = client_data->socket;
    int client_id = client_data->id;
```

```

free(arg);

char buffer[1024];

int bytes_read;

while ((bytes_read = read(client_sock, buffer, sizeof(buffer) - 1)) > 0) {
    buffer[bytes_read] = '\0';

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

    pthread_mutex_lock(&lock);

    char* command = strtok(buffer, " ");

    if (strcmp(command, "status") == 0) {
        char response[1024] = "Item status:\n";

        for (int i = 0; i < ITEM_COUNT; i++) {
            char item_status[600];

            snprintf(item_status, sizeof(item_status),
                "Item: %s, Highest Bid: %d, Highest Bidder: %s\n",
                items[i].name, items[i].highestBid, items[i].highestBidder);

            strcat(response, item_status);
        }

        write(client_sock, response, strlen(response));
    } else if (strcmp(command, "end") == 0) {
        char response[1024] = "Auction ended. Final results:\n";

        for (int i = 0; i < ITEM_COUNT; i++) {
            char item_status[600];

            snprintf(item_status, sizeof(item_status),
                "Item: %s, Final Bid: %d, Winner: %s\n",
                items[i].name, items[i].highestBid, items[i].highestBidder);
        }
    }
}

```

```

        strcat(response, item_status);
    }
    write(client_sock, response, strlen(response));
} else if (strcmp(command, "list") == 0) {
    char response[1024] = "Available items for bidding:\n";
    for (int i = 0; i < ITEM_COUNT; i++) {
        char item_info[256];
        snprintf(item_info, sizeof(item_info),
            "Item: %s\n", items[i].name);
        strcat(response, item_info);
    }
    write(client_sock, response, strlen(response));
} else {
    char* item_name = command;
    char* bid_str = strtok(NULL, " ");
    if (bid_str) {
        int bid = atoi(bid_str);
        int item_found = 0;
        for (int i = 0; i < ITEM_COUNT; i++) {
            if (strcmp(items[i].name, item_name) == 0) {
                item_found = 1;
                if (bid > items[i].highestBid) {
                    items[i].highestBid = bid;
                    snprintf(items[i].highestBidder,
                        sizeof(items[i].highestBidder),
"Client %d", client_id);
                    snprintf(buffer, sizeof(buffer),

```

```

        "New highest bid for %s: %d by %s",
        items[i].name, bid, items[i].highestBidder);
    } else {
        snprintf(buffer, sizeof(buffer),
            "Bid too low for %s. Current highest bid: %d",
            items[i].name, items[i].highestBid);
    }
    break;
}
}

if (!item_found) {
    snprintf(buffer, sizeof(buffer), "Item %s not found", item_name);
}

} else {
    snprintf(buffer, sizeof(buffer), "Invalid bid format. Use: <ItemName>
<BidAmount>");
}

write(client_sock, buffer, strlen(buffer));
}

pthread_mutex_unlock(&lock);
}

printf("Client %d disconnected\n", client_id);
close(client_sock);
return NULL;
}

```



```

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    pthread_t thread_id;

    snprintf(items[0].name, sizeof(items[0].name), "Painting");
    snprintf(items[1].name, sizeof(items[1].name), "Car");
    snprintf(items[2].name, sizeof(items[2].name), "Jewellery");

    for (int i = 0; i < ITEM_COUNT; i++) {
        items[i].highestBid = 0;
        items[i].highestBidder[0] = '\0';
    }

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, MAX_CLIENTS) < 0) {

```

```

    perror("listen");
    exit(EXIT_FAILURE);
}
printf("Server started and listening on port %d\n", PORT);
while (1) {
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    ClientData *new_client = malloc(sizeof(ClientData));
    new_client->socket = new_socket;
    new_client->id = client_id++;
    if (pthread_create(&thread_id, NULL, handle_client, (void*)new_client) != 0)
    {
        perror("Could not create thread");
        free(new_client);
    }
    pthread_detach(thread_id);
}
return 0;
}

```

ClientCode.c

```

#include <stdio.h>
#include <stdlib.h>

```

```

#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int sock = 0;

    struct sockaddr_in serv_addr;

    char buffer[1024] = {0};

    char input[1024];

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\nSocket creation error\n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;

    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        printf("\nInvalid address/ Address not supported\n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed\n");
        return -1;
    }

    while (1) {
        printf("Enter command (format: ItemName BidAmount, 'status', 'end', 'list', or 'exit'):");

```

```
fgets(input, sizeof(input), stdin);

input[strcspn(input, "\n")] = '\0'; // Remove trailing newline

if (strcmp(input, "exit") == 0) {
    break;
}

send(sock, input, strlen(input), 0);

int valread = read(sock, buffer, sizeof(buffer) - 1);

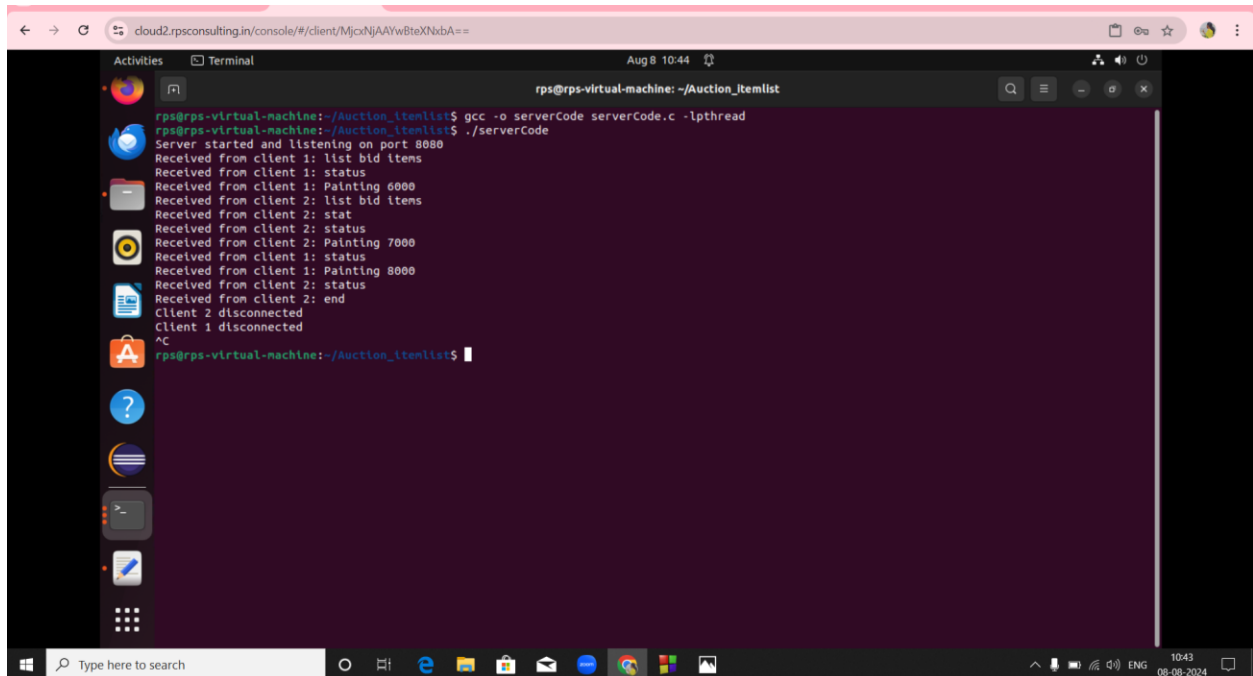
buffer[valread] = '\0';

printf("Server: %s\n", buffer);
}

close(sock);

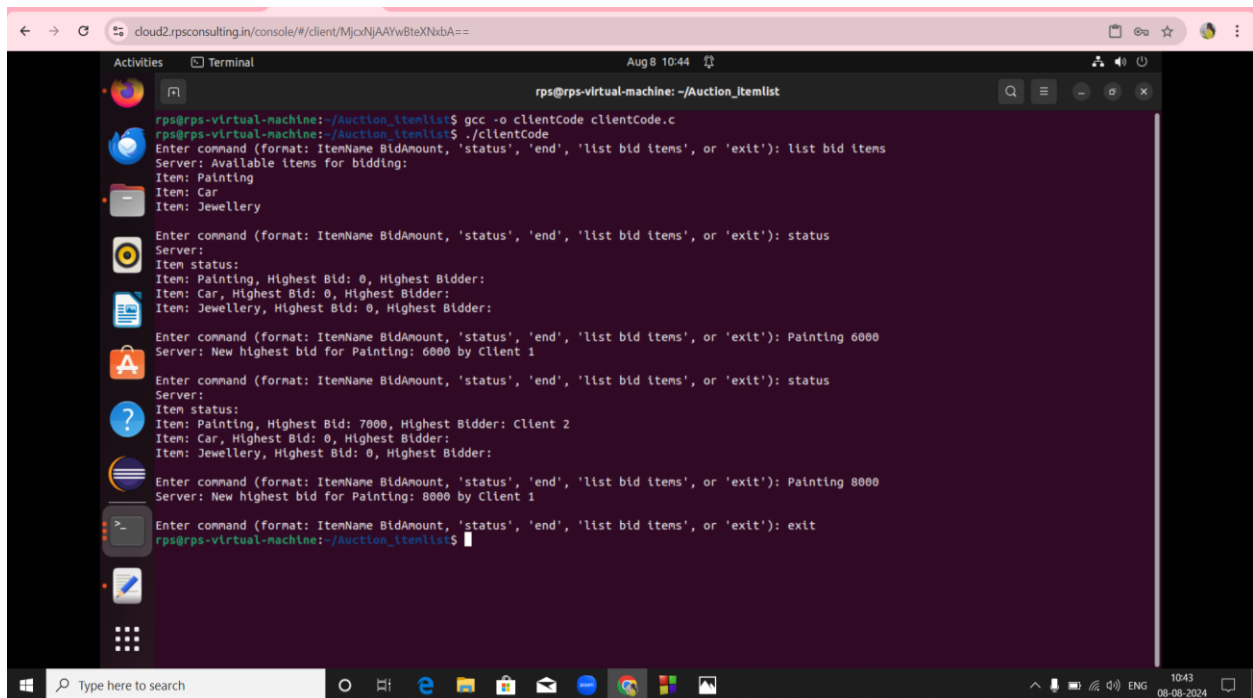
return 0;
}
```

Output Screenshot



```
rps@rps-virtual-machine: ~/Auction_ItemList
rps@rps-virtual-machine:~/Auction_ItemList$ gcc -o serverCode serverCode.c -lpthread
rps@rps-virtual-machine:~/Auction_ItemList$ ./serverCode
Server started and listening on port 8080
Received from client 1: list bid items
Received from client 1: status
Received from client 1: Painting 6000
Received from client 2: list bid items
Received from client 2: stat
Received from client 2: status
Received from client 2: Painting 7000
Received from client 1: status
Received from client 1: Painting 8000
Received from client 2: status
Received from client 2: end
Client 2 disconnected
Client 1 disconnected
rps@rps-virtual-machine:~/Auction_ItemList$
```

Server terminal



```
rps@rps-virtual-machine: ~/Auction_ItemList
rps@rps-virtual-machine:~/Auction_ItemList$ gcc -o clientCode clientCode.c
rps@rps-virtual-machine:~/Auction_ItemList$ ./clientCode
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): list bid items
Server: Available items for bidding:
Item: Painting
Item: Car
Item: Jewellery

Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): status
Server:
Item status:
Item: Painting, Highest Bid: 0, Highest Bidder:
Item: Car, Highest Bid: 0, Highest Bidder:
Item: Jewellery, Highest Bid: 0, Highest Bidder:

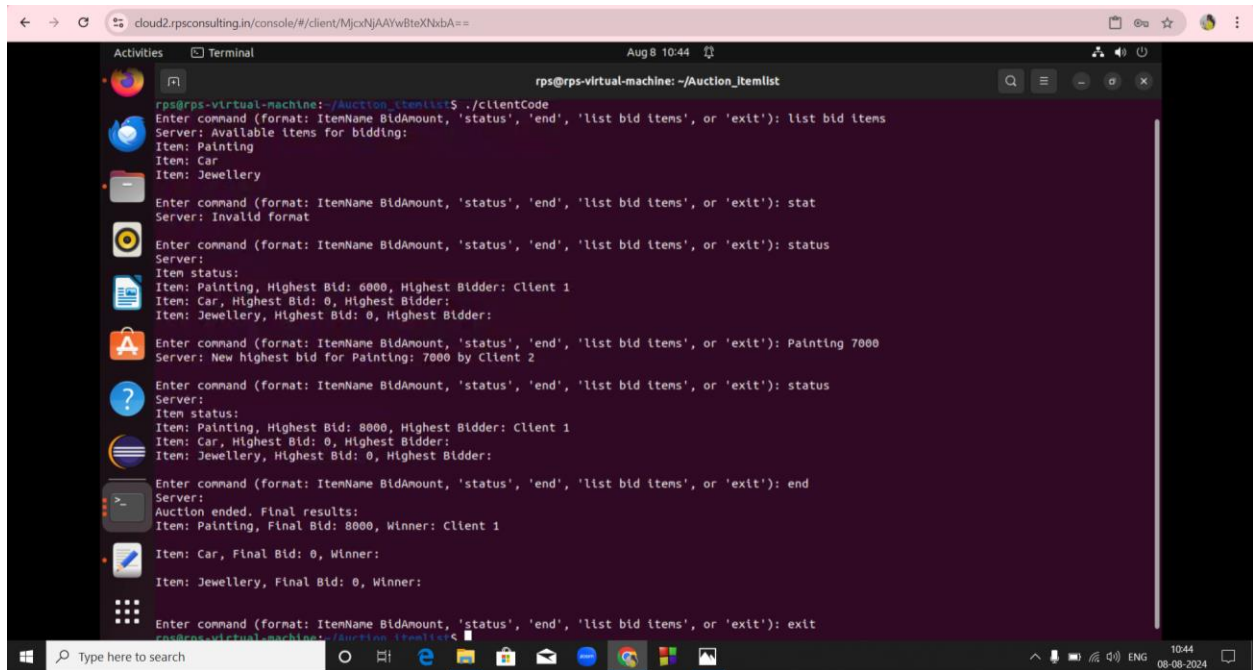
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): Painting 6000
Server: New highest bid for Painting: 6000 by Client 1

Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): status
Server:
Item status:
Item: Painting, Highest Bid: 7000, Highest Bidder: Client 2
Item: Car, Highest Bid: 0, Highest Bidder:
Item: Jewellery, Highest Bid: 0, Highest Bidder:

Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): Painting 8000
Server: New highest bid for Painting: 8000 by Client 1

Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): exit
rps@rps-virtual-machine:~/Auction_ItemList$
```

Client 1 terminal



```
rps@rps-virtual-machine: ~/Auction_ItemList
rps@rps-virtual-machine:~/Auction_ItemList$ ./clientCode
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): list bid items
Server: Available items for bidding:
Item: Painting
Item: Car
Item: Jewellery
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): stat
Server: Invalid format
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): status
Server:
Item status:
Item: Painting, Highest Bid: 6000, Highest Bidder: Client 1
Item: Car, Highest Bid: 0, Highest Bidder:
Item: Jewellery, Highest Bid: 0, Highest Bidder:
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): Painting 7000
Server: New highest bid for Painting: 7000 by Client 2
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): status
Server:
Item status:
Item: Painting, Highest Bid: 8000, Highest Bidder: Client 1
Item: Car, Highest Bid: 0, Highest Bidder:
Item: Jewellery, Highest Bid: 0, Highest Bidder:
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): end
Server:
Auction ended. Final results:
Item: Painting, Final Bid: 8000, Winner: Client 1
Item: Car, Final Bid: 0, Winner:
Item: Jewellery, Final Bid: 0, Winner:
Enter command (format: ItemName BidAmount, 'status', 'end', 'list bid items', or 'exit'): exit
```

Client 2 terminal

Conclusion

The auction system successfully demonstrates a implementation of a client-server application with concurrent bid handling and status reporting. By using multithreading and mutexes, the system can manage multiple clients and ensure data consistency. The system is tested using unit testing, integration testing, and system testing to ensure that it functions correctly. It has been tested using various scenarios and has proven to be robust and scalable.