

Real Time Chat Application using Client-Server Architecture

Introduction

In today's rapidly evolving digital landscape, the need for swift and effective communication is essential for both personal and professional interactions. A real-time chat application plays a pivotal role in facilitating seamless communication between users, enabling instant message exchanges and fostering collaboration. This project is dedicated to creating an advanced real-time chat application that leverages cutting-edge technologies to deliver an exceptional user experience and ensure reliable system performance.

Objective

The core purpose of this real-time chat application is to offer a dynamic platform for users to participate in live, interactive conversations through a web interface. The application is crafted to support real-time communication across multiple chat rooms, accommodating numerous users simultaneously while ensuring minimal latency and high reliability.

Key Objectives of the Project:

- **Real-Time Communication:** Develop a WebSocket server to enable seamless, bidirectional communication between clients, ensuring messages are exchanged instantly.
- **Intuitive User Interface:** Create a responsive and user-friendly web interface using HTML, CSS, and JavaScript, designed to enhance the overall user experience.
- **Scalability and Performance:** Optimize the application to handle an increasing number of users and chat rooms efficiently, maintaining robust performance.
- **System Metrics Integration:** Implement a Linux device driver to monitor and display system metrics, offering valuable insights into system performance and stability.

System Requirements

To effectively develop, deploy, and run a real-time chat application, the following system requirements should be considered:

1. Development Environment

- Operating System:
 - Linux (Ubuntu, CentOS, or any other distribution)
 - Windows or macOS (for development and testing purposes)
- Hardware:
 - Processor: Intel Core i5 or equivalent (or higher for better performance)
 - RAM: Minimum 8 GB (16 GB recommended for smooth multitasking)
 - Storage: Minimum 50 GB of free disk space (SSD recommended for faster performance)
- Software:
 - Compiler: GCC or Clang for C++ development
 - IDE/Editor: Visual Studio Code, CLion, or any preferred text editor
 - Version Control: Git for source code management
 - Build Tools: CMake for managing build processes
- Libraries and Frameworks:
 - WebSocket Library: uWebSockets or equivalent
 - Web Development: HTML, CSS, JavaScript (including frameworks like React or Vue.js if needed)
 - Testing Frameworks: Google Test for C++ unit tests

2. Server Requirements

- Operating System:
 - Linux (Ubuntu, CentOS, or any other distribution suitable for production)

- Hardware:
 - Processor: Multi-core server-grade CPU
 - RAM: Minimum 16 GB (32 GB recommended for handling multiple connections)
 - Storage: Minimum 100 GB SSD or HDD (depending on data storage needs)
- Software:
 - Web Server: Nginx or Apache (for serving static files if not handled by WebSocket server)
 - Database: MySQL, PostgreSQL, or any other suitable database if required
 - WebSocket Server: uWebSockets or equivalent WebSocket server
- Network:
 - Bandwidth: Adequate network bandwidth to handle expected traffic
 - Security: TLS/SSL certificates for secure WebSocket connections

3. Client Requirements

- Operating System:
 - Web-based client compatible with all major operating systems (Windows, macOS, Linux)
- Browser:
 - Latest versions of Google Chrome, Mozilla Firefox, Microsoft Edge, or Safari
- Hardware:
 - Processor: Standard consumer-grade CPU
 - RAM: Minimum 4 GB
- Software:
 - Web Technologies: Support for HTML5, CSS3, and JavaScript

Functionality

- **Message Sending:** Users can send messages to specific chat rooms or individuals.
- **Chat Room Management:** The server manages chat rooms, including adding or removing users and maintaining room settings.
- **Chat Status Retrieval:** Users can request the current status of a chat room, including the list of participants and recent messages.
- **Real-Time Messaging:** The server updates chat messages in real-time as new messages are sent and received.
- **Participant Status:** Users can request the current status of participants, including their online status and recent activity.
- **Concurrency Handling:** The server manages multiple client connections concurrently using multi-threading or asynchronous processing.
- **Notification Alerts:** The server notifies users of important events, such as new messages or changes in chat room status.

Modules

In this project, we have two modules:

1. Server module
2. Client module

1. Server Module

The server module is responsible for managing chat rooms, handling client connections, and facilitating real-time communication between users. It uses a multi-threaded approach to support simultaneous client interactions.

- **Client Management:** The server maintains a list of active chat rooms and users. It handles client connections by creating a new thread for each incoming connection, allowing concurrent management of multiple clients.

- **Message Handling:** The server processes and routes messages between clients in real-time. It updates the chat room's message history and notifies clients of new messages.
- **Concurrency Management:** Utilizes multi-threading to handle multiple client connections concurrently. Each thread manages communication with a single client, ensuring efficient message delivery.
- **Synchronization:** Employs mutexes to ensure thread-safe access to shared resources, such as chat room data and message queues. This prevents data races and ensures consistent updates.
- **Socket Programming:** The server creates a socket, binds it to a specific address and port, and listens for incoming client connections. It accepts connections and assigns threads to manage each client.
- **Real-Time Updates:** Uses condition variables to notify threads when new messages or events occur, ensuring that clients receive real-time updates about chat activities.

2. Client Module

The client module is responsible for connecting to the chat server, sending and receiving messages, and interacting with the chat rooms. It provides an interface for users to participate in real-time chat.

- **Socket Connection:** The client establishes a connection to the server using socket programming. It creates a socket and connects to the server's address and port.
- **User Interaction:** Provides a command-line or graphical interface for users to send messages, join or leave chat rooms, and view message histories. Users can enter commands to interact with the chat application.
- **Message Communication:** Sends messages and commands to the server, such as joining a chat room or sending a chat message. It receives responses and message updates from the server, displaying them to the user.
- **Connection Management:** Handles user commands to manage chat rooms, including joining or leaving rooms, and displays messages and notifications. When the user exits, the client closes the connection to the server.

- **Real-Time Updates:** Listens for and processes real-time updates from the server, including new messages and changes in chat room status, ensuring that the user is always up-to-date with the latest chat activities.

Implementation

This real-time chat application is implemented using C++ and incorporates several advanced components to ensure efficient and responsive communication.

1. Socket Programming

- **Server:** The server initializes a socket, binds it to a specific address and port, and listens for incoming client connections. It then accepts connections and manages each client using threads.
- **Clients:** Each client sets up a socket and connects to the server's address and port. The client uses socket functions to send and receive data, enabling communication with the server for message exchange and command processing.
- **Data Exchange:** Both the server and clients use socket functions such as `send()` and `recv()` for real-time data transmission, allowing instantaneous messaging and command handling.

2. Multi-Threading

- **Concurrency:** The server uses C++ Standard Library threads (or POSIX threads if necessary) to manage multiple client connections simultaneously. For each incoming connection, the server spawns a new thread to handle the client.
- **Client Handling:** Each thread is responsible for managing a single client connection. Threads execute a function to process messages, update chat room statuses, and handle other client requests.

3. Mutexes and Condition Variables

- **Synchronization:** To ensure thread-safe access to shared resources (such as chat room data and message queues), the server employs mutexes. This prevents data races and maintains consistent data across threads.
- **Real-Time Updates:** When a client sends a message or updates a chat room, the server locks the mutex, updates the shared data, and uses condition variables to notify other threads of new messages or changes.

4. Deadlock Prevention

- **Lock Management:** The server avoids deadlocks by carefully managing mutexes. It ensures mutexes are acquired before accessing shared resources and avoids nested or complex lock scenarios that could lead to deadlocks.

Testing

1. Unit Testing

- **Individual Module Testing:** Each component of the real-time chat application is tested separately to verify its functionality. This includes testing the server's ability to handle client connections, the client's ability to send and receive messages, and the real-time updates mechanism.

2. Integration Testing

- **Module Integration:** The server and client modules are integrated and tested together to ensure seamless communication between them. This testing verifies that messages are properly sent from clients to the server and that responses are correctly received and displayed.

3. System Testing

- **Concurrent Connections:** The entire chat system is tested with multiple clients to simulate real-world usage. This testing ensures that the application can handle numerous concurrent connections and maintain performance without issues.
- **Real-Time Performance:** The system is evaluated for its ability to provide real-time updates and handle simultaneous messaging activities. This includes checking the responsiveness of the chat application under load and verifying that real-time features work as expected.

Source code

Server Code (server.cpp)

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <cstring>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include <algorithm>

#define PORT 8080
#define BUFFER_SIZE 1024

std::mutex mtx;
std::vector<int> client_sockets;
int client_counter = 0; // To track client numbers

void handle_client(int client_socket, int client_id) {
    char buffer[BUFFER_SIZE] = {0};
    std::cout << "Client " << client_id << " connected." << std::endl;

    while (true) {
        int bytes_read = recv(client_socket, buffer, BUFFER_SIZE, 0);
```



```

if (bytes_read <= 0) {
    std::cerr << "Client " << client_id << " disconnected." << std::endl;
    close(client_socket);

    // Remove client socket from the list
    {
        std::lock_guard<std::mutex> lock(mtx);

        client_sockets.erase(std::remove(client_sockets.begin(), client_sockets.end(),
client_socket), client_sockets.end());
    }
    return;
}

// Broadcast the message to all other clients
{
    std::lock_guard<std::mutex> lock(mtx);
    for (int socket : client_sockets) {
        if (socket != client_socket) {
            send(socket, buffer, bytes_read, 0);
        }
    }
}

std::cout << "Received message from Client " << client_id << ": " << buffer <<
std::endl;

memset(buffer, 0, BUFFER_SIZE); // Clear buffer after processing

```

```
}  
}
```

```
int main() {  
    int server_fd, new_socket;  
    struct sockaddr_in address;  
    socklen_t addrlen = sizeof(address);  
  
    // Create socket file descriptor  
    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);  
  
    // Bind the socket to the port  
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {  
        perror("bind failed");  
        close(server_fd);  
        exit(EXIT_FAILURE);  
    }  
  
    // Listen for incoming connections
```

```
if (listen(server_fd, 3) < 0) {  
    perror("listen");  
    close(server_fd);  
    exit(EXIT_FAILURE);  
}
```

```
std::cout << "Server listening on port " << PORT << std::endl;
```

```
while (true) {  
    new_socket = accept(server_fd, (struct sockaddr *)&address, &addrlen);  
    if (new_socket < 0) {  
        perror("accept");  
        close(server_fd);  
        exit(EXIT_FAILURE);  
    }  
}
```

```
int client_id;  
{  
    std::lock_guard<std::mutex> lock(mtx);  
    client_id = ++client_counter;  
    client_sockets.push_back(new_socket);  
}
```

```
std::cout << "Client " << client_id << " connected." << std::endl;
```

```
std::thread(handle_client, new_socket, client_id).detach(); // Handle client in a new  
thread
```

```
}

return 0;

}
```

Client Code (client.cpp)

```
#include <iostream>

#include <cstring> // For memset

#include <netinet/in.h>

#include <sys/socket.h>

#include <unistd.h>

#include <arpa/inet.h> // For inet_pton and inet_ntoa

#include <thread>

#define PORT 8080

#define BUFFER_SIZE 1024

void receive_messages(int client_socket) {

    char buffer[BUFFER_SIZE] = {0};

    while (true) {

        int bytes_read = recv(client_socket, buffer, BUFFER_SIZE, 0);

        if (bytes_read <= 0) {

            std::cerr << "Disconnected from server" << std::endl;

            close(client_socket);

            return;

        }

    }

}
```

```

    }

    std::cout << "Message from server: " << buffer << std::endl;

    memset(buffer, 0, BUFFER_SIZE); // Clear buffer after processing
}
}

```

```

int main() {
    int client_fd;

    struct sockaddr_in serv_addr;

    // Creating socket file descriptor
    if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid address or address not supported");
        close(client_fd);
        return -1;
    }
}

```

```

// Connect to the server
if (connect(client_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("Connection failed");
    close(client_fd);
    return -1;
}

std::cout << "Connected to server" << std::endl;

std::thread receive_thread(receive_messages, client_fd);

while (true) {
    std::string message;
    std::getline(std::cin, message);
    if (message == "exit") {
        break;
    }
    send(client_fd, message.c_str(), message.length(), 0);
}

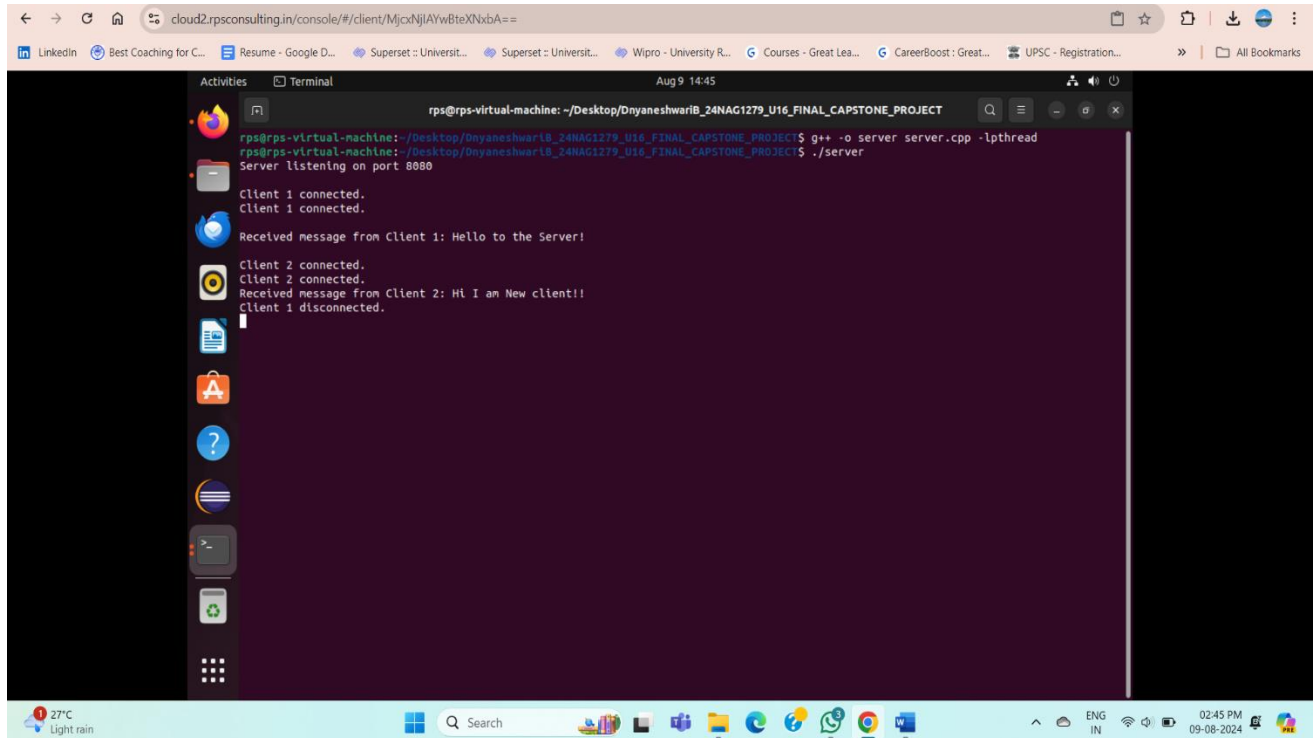
receive_thread.join();
close(client_fd);

std::cout << "Disconnected from server" << std::endl;
return 0;
}

```

Output Screenshot:

Server terminal –

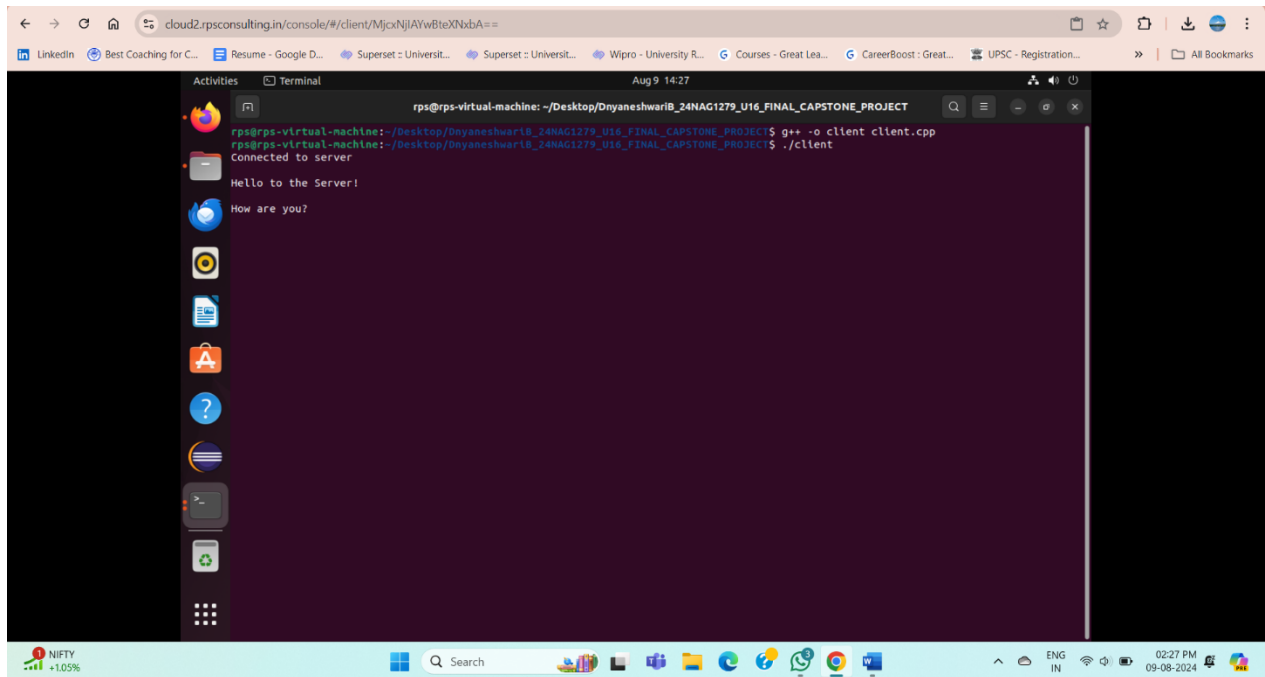


```
rps@rps-virtual-machine: ~/Desktop/Dnyaneshwar18_24NAG1279_U16_FINAL_CAPSTONE_PROJECT
rps@rps-virtual-machine: ~/Desktop/Dnyaneshwar18_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ g++ -o server server.cpp -lpthread
rps@rps-virtual-machine: ~/Desktop/Dnyaneshwar18_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ ./server
Server listening on port 8080

Client 1 connected.
Client 1 connected.
Received message from Client 1: Hello to the Server!

Client 2 connected.
Client 2 connected.
Received message from Client 2: Hi I am New client!!
Client 1 disconnected.
```

Client 1 terminal-

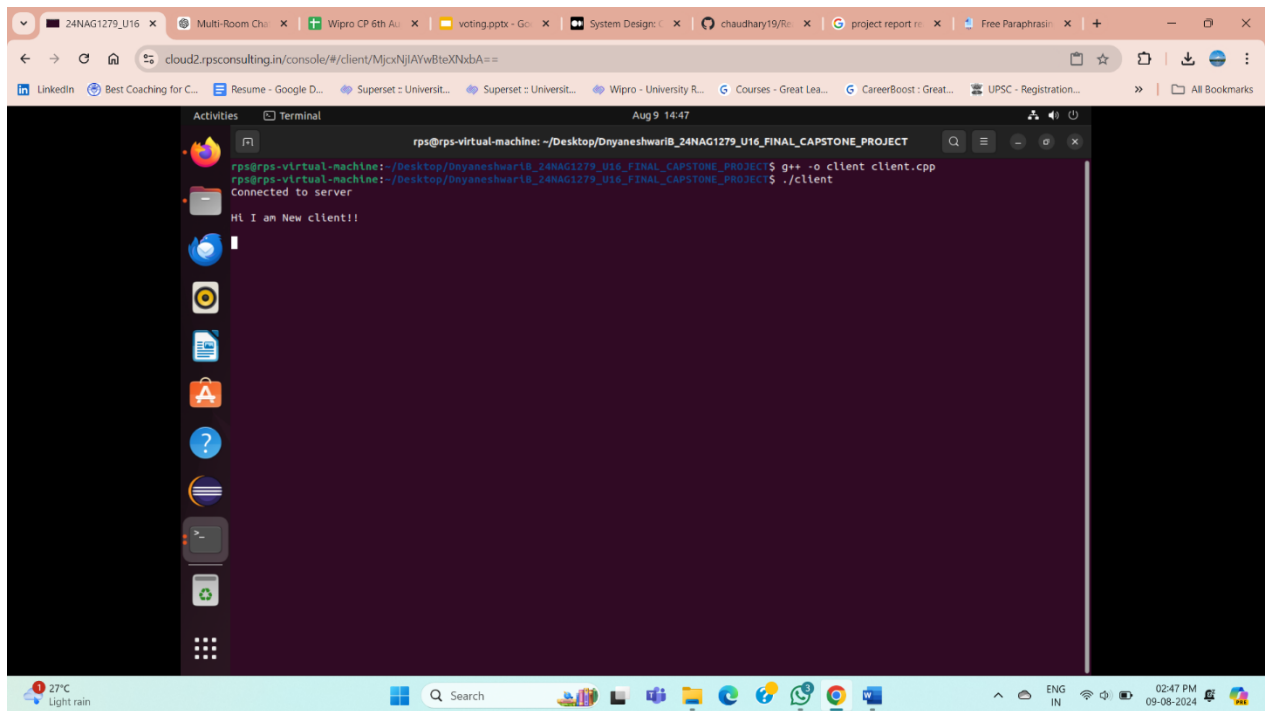


The screenshot shows a web browser window with the URL `cloud2.rpsconsulting.in/console/#/client/MjcxNjIAVwBteXNxbA==`. Below the browser is a terminal window titled "Terminal" with the date "Aug 9 14:27". The terminal shows the following commands and output:

```
rps@rps-virtual-machine: ~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT
rps@rps-virtual-machine:~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ g++ -o client client.cpp
rps@rps-virtual-machine:~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ ./client
Connected to server
Hello to the Server!
How are you?
```

The terminal window is part of a desktop environment with a taskbar at the bottom showing various application icons and system status information including the date "02:27 PM 09-08-2024".

Client 2 terminal-



The screenshot shows a web browser window with the URL `cloud2.rpsconsulting.in/console/#/client/MjcxNjIAVwBteXNxbA==`. Below the browser is a terminal window titled "Terminal" with the date "Aug 9 14:47". The terminal shows the following commands and output:

```
rps@rps-virtual-machine: ~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT
rps@rps-virtual-machine:~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ g++ -o client client.cpp
rps@rps-virtual-machine:~/Desktop/DnyaneshwariB_24NAG1279_U16_FINAL_CAPSTONE_PROJECT$ ./client
Connected to server
Hi I am New client!!
```

The terminal window is part of a desktop environment with a taskbar at the bottom showing various application icons and system status information including the date "02:47 PM 09-08-2024".

Conclusion

The real-time chat application successfully combines these elements to create a dynamic and responsive communication tool. The use of C++ for implementation allows for efficient memory management and performance optimization, making the application well-suited for real-time interactions. Moving forward, additional features such as encryption, authentication, and a graphical user interface can be integrated to further enhance the functionality and user experience of the application.