Project Report

On

NETWORK CHAT APPLICATION WITH UDP

Submitted in partial fulfillment of the requirements for the award of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

(Artificial Intelligence & Machine Learning)

by

Ms. L SHARANYA (22WH1A6610)

Ms. L VARSHA (22WH1A6612)

Ms. P KRITIKA (22WH1A6625)

Ms. O V CHANDRIKA (22WH1A6627)

Under the esteemed guidance of

Ms. P Anusha

Assistant Professor, CSE(AI&ML)

Department of Computer Science & Engineering

(Artificial Intelligence & Machine Learning)

BVRIT HYDERABAD COLLEGE OF ENGINEERING FOR WOMEN

(AUTONOMOUS)

(Approved by AICTE, New Delhi and Affiliated to JNTUH, Hyderabad)

Accredited by NBA and NAAC with A Grade

Bachupally, Hyderabad – 500090

2024-25

ABSTRACT

This paper presents the design and implementation of a network chat application using User Datagram Protocol (UDP), a connectionless transport protocol. Unlike TCP, UDP offers lower latency and faster transmission, making it well-suited for real-time communication applications where speed is crucial and some degree of data loss is acceptable. This chat application allows users to send and receive text-based messages over a network, utilizing a client-server model where multiple clients can communicate with each other through a central server. In the application, the server acts as a central point that listens for incoming messages from clients and forwards those messages to other connected clients.

PROBLEM STATEMENT

Design and implement a simple network chat application using the User Datagram Protocol (UDP). The application should allow two or more users to communicate with each other in real-time over a network. The key challenge is to implement the application in such a way that it handles network communication without the reliability mechanisms provided by protocols like TCP. The task is to create a simple network chat application using the User Datagram Protocol (UDP), allowing two or more users to communicate with each other in real-time. However, since UDP is connectionless and does not provide reliability mechanisms such as guaranteed delivery, packet ordering, or retransmission, we need to account for these limitations in our design. Below is a detailed explanation of the design and implementation steps for the chat application

FUNCTIONAL REQUIREMENTS

1. User Registration and Authentication:

- Users should be able to register and authenticate (optional for a simple application) before starting to chat.
- A unique username is required for each user to identify them in the chat system

2. Sending and Receiving Messages:

- Users can send text-based messages to other users or groups.
- Messages sent from a client should be broadcast to all other connected clients by the server.

3. Multicast Communication:

- The server should broadcast messages using UDP multicast to all clients.
- Clients should be able to join multicast groups to receive messages sent by the server or other clients.

4. Message Display:

- The application should display received messages in the client interface in a readable format.
- Each message should show the sender's username and the content of the message.

5. Real-Time Updates:

- Messages should be displayed on all clients in real time as they are received.

6. Support for Multiple clients:

- The server should support multiple clients connecting and exchanging messages simultaneously.

7. **Server Management:**

The server should handle multiple clients simultaneously, using non-blocking I/O and efficient message forwarding to avoid lag and ensure the system is responsive.

NON-FUNCTIONAL REQUIREMENTS

1. **Performance:**
    - The system should provide low-latency message delivery to ensure real-time communication.
    - The server must be able to handle at least 100 simultaneous client connections without significant degradation in performance.
2. **Scalability:**
    - The system should be scalable to support a growing number of clients, especially in the case of an increasing number of users over time.
    - The server should be able to dynamically handle new clients joining the chat with minimal delay.
3. **Availability:**
    - The system must be available 24/7, ensuring continuous monitoring of the network.
    - It should include failover mechanisms to ensure high availability in case of server or component failure.
4. **Security:**
    - The system should use encryption for all data transmission, including monitoring data and alert messages.
    - It should include strong authentication and authorization mechanisms to protect against unauthorized access.
5. **Usability:**
    - The system should have an intuitive user interface that allows network administrators to configure and monitor the network easily.
    - The dashboard should be customizable, providing both high-level overviews and detailed information on demand.
6. **Interoperability:**
    - The system should support integration with a wide range of network devices, operating systems, and third-party applications.
    - It should support standard protocols such as SNMP, NetFlow, and syslog.

7. **Data Integrity:**
    - The system must ensure that the collected network data is accurate and reliable.
    - It should perform periodic integrity checks and maintain consistency in data storage.
8. **Maintainability:**
    - The system should be designed to facilitate easy updates, bug fixes, and enhancements.
    - It should have clear documentation for system maintenance and troubleshooting.
9. **Extensibility:**
    - The system should be easily extensible to support future features and integrations, such as additional protocols, security measures, or cloud-based services.

SOURCE CODE

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

#define PORT 8080
#define MAX_BUFFER 1024

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char buffer[MAX_BUFFER];

    // Create UDP socket
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Fill server information
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket to the port
    if (bind(sockfd, (const struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
```

```
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    while (1) {
        // Receive message from client
        int n = recvfrom(sockfd, (char *)buffer, MAX_BUFFER, 0, (struct sockaddr
*)&client_addr, &addr_len);
        buffer[n] = '\0';
        printf("Client: %s\n", buffer);

        // Send a response back to the client
        const char *response = "Message received";
        sendto(sockfd, (const char *)response, strlen(response), 0, (const struct sockaddr
*)&client_addr, addr_len);
    }

    close(sockfd);
    return 0;
}
```

//client.c

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

```
#define PORT 8080
#define MAX_BUFFER 1024

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[MAX_BUFFER];

    // Create UDP socket
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Fill server information
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    while (1) {
        printf("Enter message: ");
        fgets(buffer, MAX_BUFFER, stdin);
        buffer[strcspn(buffer, "\n")] = '\0';  // Remove newline character

        // Send message to server
        sendto(sockfd, (const char *)buffer, strlen(buffer), 0, (const struct sockaddr
*)&server_addr, sizeof(server_addr));
```

```
        // Receive response from server
        int n = recvfrom(sockfd, (char *)buffer, MAX_BUFFER, 0, NULL, NULL);
        buffer[n] = '\0';
        printf("Server: %s\n", buffer);
    }

    close(sockfd);
    return 0;
}
```

COMPILATION AND EXECUTION

Compile :

gcc -o network_monitor network_monitor.c

Run the executable code:

./network_monitor

OUTPUT

```
Server listening on port 8080...
```

```
Enter message: Hello we are from BVRITH
Server: Message received
Enter message:
```

```
Server listening on port 8080...
Client: Hello we are from BVRITH
```

```
Enter message: Hello we are from BVRITH
Server: Message received
Enter message: This is our Computer Networks project
```

```
Enter message: Hello we are from BVRITH
Server: Message received
Enter message: This is our Computer Networks project
Server: Message received
Enter message:
```

```
Server listening on port 8080...
Client: Hello we are from BVRITH
Client: This is our Computer Networks project
```

```
Enter message: Hello we are from BVRITH
Server: Message received
Enter message: This is our Computer Networks project
Server: Message received
Enter message: End
Server: Message received
Enter message:
```

```
Server listening on port 8080...
Client: Hello we are from BVRITH
Client: This is our Computer Networks project
Client: End
```