SMS BASED REMOTE SERVER MONITORING SYSTEM

INTRODUCTION

The organization who is hosting the data server for a large number of clients faces the problem with the monitor of the up and down of the server using Remote Server Monitoring System. The problem is more on the night as at that time the number of network engineer is less. When downtime of the server happen, the organization should identify and remove the problem but in general, the client reports about the issue with the server when they try to access to the system and due to this the organization can cause the loss of the customer.

EXISTING SYSTEM

In the present Remote Server Monitoring System when the server failure happens then the network engineer makes the system working. But at the night time when there is less engineer available organization faces the problem, whereas the problem on the server is identified by the client and not by the organization. Even when the organization knows about the server problem, they have difficulty in removing them as the engineer at first needs to find out about the bad server. Whereas the problem can be temporarily, but it can occur multiple time because of this the engineer will face difficulty in the finding of the fault in the server.

PROPOSED SYSTEM

This Remote Server Monitoring System will monitor each server of the organization by sending a message to the servers. Then the server will respond to that message within the specified time, and client can send the information

to the server also, there can be two-way communication between the server and client. Also, the data transferred by the client to the server can be updated, deleted or inserted according to the instruction provided by the client. And the problem from the server side can be rectified easily by the client's instructions by using java socket programming.

JAVA SOCKET PROGRAMMING –

By definition, a *socket* is one endpoint of a two-way communication link between two programs running on different computers on a network. A socket is bound to a port number so that the transport layer can identify the application that data is destined to be sent to.

To connect to another machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

Socket socket = new Socket("local host", 1234)

- The first argument – **IP address of Server**. (IP address of localhost, where code will run on the single stand-alone machine), but as we are transferring the data from the same computer i.e local host is used as IP address.
- The second argument – **TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

To communicate over a socket connection, streams are used to both input and output the data.
In the program, the Client keeps reading input from a user and sends it to the server until "logout" is typed.
Server application makes a ServerSocket on a specific port which is 1234. This starts our Server listening for client requests coming in for port 1234.

Java provides a set of classes and interfaces, such as **Socket**, **ServerSocket**, **DatagramSocket**, and **DatagramPacket**, which can be used to create and

manage sockets, and exchange data between client and server applications. The client application initiates a connection to a server socket, and the server socket responds to the client's request by opening a new socket for communication. Once the connection is established, the client and server can exchange data in real-time.

PAKAGES USED IN THE CODE –

These are the packages, for which we need to make the following imports:

1. **import java.net.*;**
   The java.net package provides a set of classes and interfaces for working with network resources, including URLs, sockets, and datagrams.
   We need the *java.io* package, which gives us input and output streams to write to and read from while communicating. It is used in the given code to create a socket connection between the client and server.

2. **import java.io.*;**
   For the sake of simplicity, we'll run our client and server programs on the same computer. If we were to execute them on different networked computers, the only thing that would change is the IP address.
   The **java.io** package is used for performing input and output operations in Java. It provides classes and interfaces for reading and writing data streams, files, and other types of input/output sources.

3. The **java.nio.file** package provides a modern, object-oriented approach to working with files and directories in Java. Some of the key features of this package include:
   - ❖ Path API: The **Path** interface provides a platform-independent representation of a path to a file or directory. It can be used to construct and manipulate paths in a way that is independent of the underlying file system.
   - ❖ File I/O API: The **Files** class provides a set of methods for reading from and writing to files, creating and deleting directories, copying and moving files, and other file-related operations.
   - ❖ Watch Service API: The **WatchService** interface provides a way to monitor a directory for changes, such as the creation, deletion, or modification of files or directories within that directory.

Overall, the **java.nio.file** package provides a more robust and flexible way to work with files and directories than the older **java.io** package.

4. The **import java.util.*;** statement allows to use all the classes and interfaces defined in the **java.util** package without having to specify them individually. The **java.util** package provides a collection of utility classes such as **Scanner** for input processing.

Client class

The code is a Java program that implements a simple client that connects to a server using a socket connection.
The program uses the following classes and interfaces from the **java.io** package:

- **BufferedReader**: Used to read text from a character-input stream, such as the input stream of a socket.
- **InputStreamReader**: Used to bridge byte streams to character streams.
- **BufferedWriter**: Used to write text to a character-output stream, such as the output stream of a socket.
- **OutputStreamWriter**: Used to bridge character streams to byte streams.

Here, **socket.getInputStream()** returns an input stream connected to the server, and **socket.getOutputStream()** returns an output stream connected to the server. These streams are then wrapped in **BufferedReader** and **BufferedWriter** objects to provide buffering of data and ease of use.
The **java.net** package is also used to handle exceptions that may occur during socket communication. In the given code, the **try-catch** block is used to catch any **IOException** that may be thrown during socket communication.

String line = reader.readLine();
Reads a line of text from the server using the **BufferedReader** object.

After that by using while an infinite loop is started, if user types “exit” the loops breaks and the program ends.
String[] parts = input.split(",");
String action = parts[0];
Splits the user input into an array of strings based on the comma separator and assigns the first element to the **action** variable.

Based on the value of the **action** variable, the client sends a request to the server using the **BufferedWriter** object, reads the server response using the **BufferedReader** object, and prints it to the console.

CODE-

CLIENT.JAVA

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Client {
    public static void main(String[] args) {
    try (Socket socket = new Socket("localhost", 1234);
BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
BufferedWriter writer = new BufferedWriter
    (new OutputStreamWriter(socket.getOutputStream()));
            Scanner scanner = new Scanner(System.in)) {
           String line = reader.readLine();
           System.out.println(line);
           while (true) {
              System.out.print("> ");
              String input = scanner.nextLine();
              if (input.equals("exit")) {
                 break;
              }
              String[] parts = input.split(",");
              String action = parts[0];
              if (action.equals("login")&& parts.length >= 3) {
                 String username = parts[1];
                 String password = parts[2];
                 writer.write(input + "\n");
                 writer.flush();
                 line = reader.readLine();
                 System.out.println(line);
              } else if (action.equals("logout")) {
                 writer.write(input + "\n");
                 writer.flush();
                 line = reader.readLine();
                 System.out.println(line);
              } else if (action.equals("insert")) {
```

```
            writer.write(input + "\n");
            writer.flush();
            line = reader.readLine();
            System.out.println(line);
        } else if (action.equals("delete")) {
             writer.write(input + "\n");
             writer.flush();
            line = reader.readLine();
            System.out.println(line);
        } else if (action.equals("update")) {
            writer.write(input + "\n");
            writer.flush();
            line = reader.readLine();
            System.out.println(line);
        } else if (action.equals("get")) {
           writer.write(input + "\n");
           writer.flush();
          while ((line = reader.readLine()) != null) {
          System.out.println(line);
        }
        } else {
            System.out.println("Invalid action.");
         }
    }
    } catch (IOException e) {
      e.printStackTrace();
}}}
```

SERVER CLASS –

In the server class credentials are stored in the file and as the client logins different commands are executed which includes insertion, deletion, updating of the data stored in the server.

private static final String CREDENTIALS_FILE = "credentials.txt";

This creates a private, static, and final field named **CREDENTIALS_FILE** that contains the filename for the file that stores the user credentials.

private static final String DATA_FILE = "data.txt";

This creates a private, static, and final field named **DATA_FILE** that contains the filename for the file that stores the data.

private static List<String> authenticatedClients = new ArrayList<>();

This creates a private, static field named **authenticatedClients** that is of type **List<String>**. It is initialized with a new empty **ArrayList**.

The use of this variable is likely part of a larger authentication system implemented within the application, which may involve verifying the client's credentials and granting them a token or session ID that can be used for subsequent requests. By keeping track of authenticated clients in this way, the application can ensure that only authorized users are able to access sensitive or restricted functionality.

try (ServerSocket serverSocket = new ServerSocket(1234))

This creates a new **ServerSocket** instance that listens for incoming connections on port **1234**.

Socket socket = serverSocket.accept();

This waits for a client to connect to the server and accepts the connection, returning a new **Socket** instance to communicate with the client.

new Thread(new ClientHandler(socket)).start();

This creates a new **Thread** instance and passes a new **ClientHandler** instance that takes the **Socket** instance as an argument to the **Thread** constructor. It then starts the thread to handle the client's requests.

List<String> lines = Files.readAllLines(Paths.get(CREDENTIALS_FILE));

This reads all the lines from the **CREDENTIALS_FILE** file and stores them in a **List** of **String**s.

catch (IOException e) { e.printStackTrace(); }

This catches any **IOException** that occurs while reading the file and prints the error stack trace to the console.

Then separate methods are provided for the insertion, deletion and updating the data in data.txt file by the client.

private static class ClientHandler implements Runnable

This line defines the **ClientHandler** class as a private static inner class that implements the **Runnable** interface.

public ClientHandler(Socket socket) { this.socket = socket; }

This is the constructor of the **ClientHandler** class, which sets the value of the **socket** field.

After that if the action is "login" and there are at least three elements in the array, this code attempts to authenticate the user. If authentication is successful, the user's username is added to a list of authenticated clients, and a success message is sent to the client. If authentication fails, an error message is sent instead.

SERVER.JAVA -

```
import java.io.*;
import java.net.*;
import java.nio.file.*;
import java.util.*;

public class Server {

    private static final String CREDENTIALS_FILE = "credentials.txt";
    private static final String DATA_FILE = "data.txt";

    private static List<String> authenticatedClients = new ArrayList<>();

    public static void main(String[] args) {
                        System.out.println("Welcome to our server");
                        System.out.println("Client can login and send the commands");
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                Socket socket = serverSocket.accept();
                new Thread(new ClientHandler(socket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static boolean authenticate(String username, String password) {
        try {
            List<String> lines = Files.readAllLines(Paths.get(CREDENTIALS_FILE));
            for (String line : lines) {
                String[] parts = line.split(",");
                if (parts[0].equals(username) && parts[1].equals(password)) {
                    return true;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    private static boolean insertData(String data) {
```

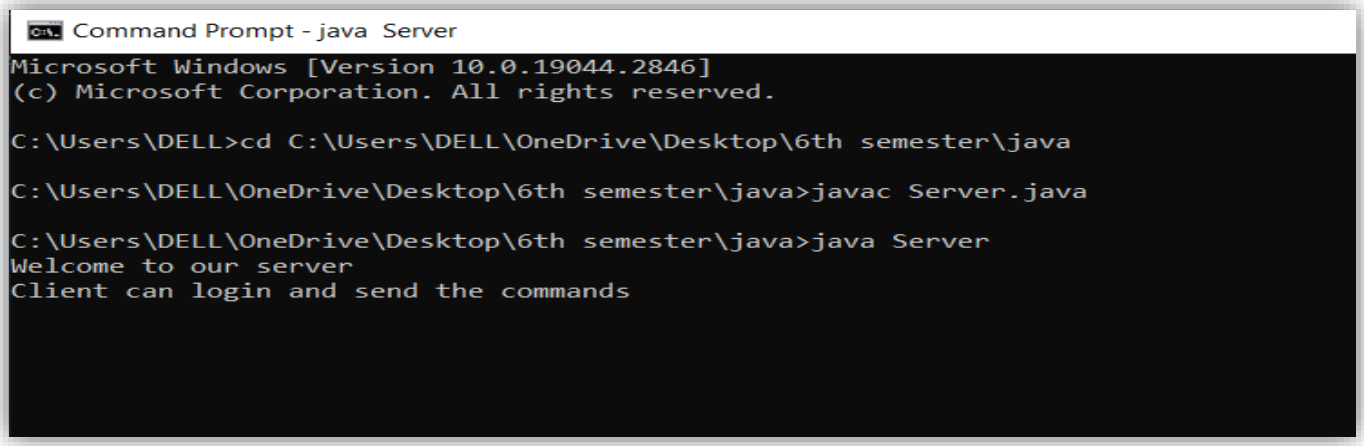
```
        try {
            Files.write(Paths.get(DATA_FILE), (data + "\n").getBytes(), StandardOpenOption.CREATE,
StandardOpenOption.APPEND);
            return true;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    private static boolean deleteData(String data) {
        try {
            List<String> lines = Files.readAllLines(Paths.get(DATA_FILE));
            List<String> newLines = new ArrayList<>();
            boolean found = false;
            for (String line : lines) {
                if (line.equals(data)) {
                    found = true;
                } else {
                    newLines.add(line);
                }
            }
            if (found) {
                Files.write(Paths.get(DATA_FILE), String.join("\n", newLines).getBytes(),
StandardOpenOption.TRUNCATE_EXISTING);
                return true;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    private static boolean updateData(String oldData, String newData) {
        try {
            List<String> lines = Files.readAllLines(Paths.get(DATA_FILE));
            List<String> newLines = new ArrayList<>();
            boolean found = false;
            for (String line : lines) {
                if (line.equals(oldData)) {
                    newLines.add(newData);
                    found = true;
                } else {
                    newLines.add(line);
                }
            }
            if (found) {
                Files.write(Paths.get(DATA_FILE), String.join("\n", newLines).getBytes(),
StandardOpenOption.TRUNCATE_EXISTING);
                return true;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
```

```
    }

    private static class ClientHandler implements Runnable {
        private final Socket socket;

        public ClientHandler(Socket socket) {
            this.socket = socket;
        }

        //Override
        public void run() {
            try (BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                 BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream()))) {
                writer.write("Welcome to the chat server.\n");
                writer.flush();
                String username = null;
                while (true) {
                    String line = reader.readLine();
                    if (line == null) {
                        break;
                    }
                    String[] parts = line.split(",");
                    String action = parts[0];
                    if (action.equals("login")&& parts.length >= 3) {
                    username = parts[1];
                    String password = parts[2];
                    if (authenticate(username, password)) {
                    authenticatedClients.add(username);
                    writer.write("Login successful.\n");
                    writer.flush();
                   } else {
                     writer.write("Invalid username or password.\n");
                     writer.flush();
                   }
                   } else if (action.equals("logout")) {
                     authenticatedClients.remove(username);
                     writer.write("Logout successful.\n");
                     writer.flush();
                     break;
                  } else if (action.equals("insert")) {
                     if (authenticatedClients.contains(username)) {
                        String data = parts[1];
                        if (insertData(data)) {
                        writer.write("Data inserted successfully.\n");
                        writer.flush();
                   } else {
                   writer.write("Failed to insert data.\n");
                   writer.flush();
                   }
                   } else {
                   writer.write("You must be logged in to perform this action.\n");
                   writer.flush();
```

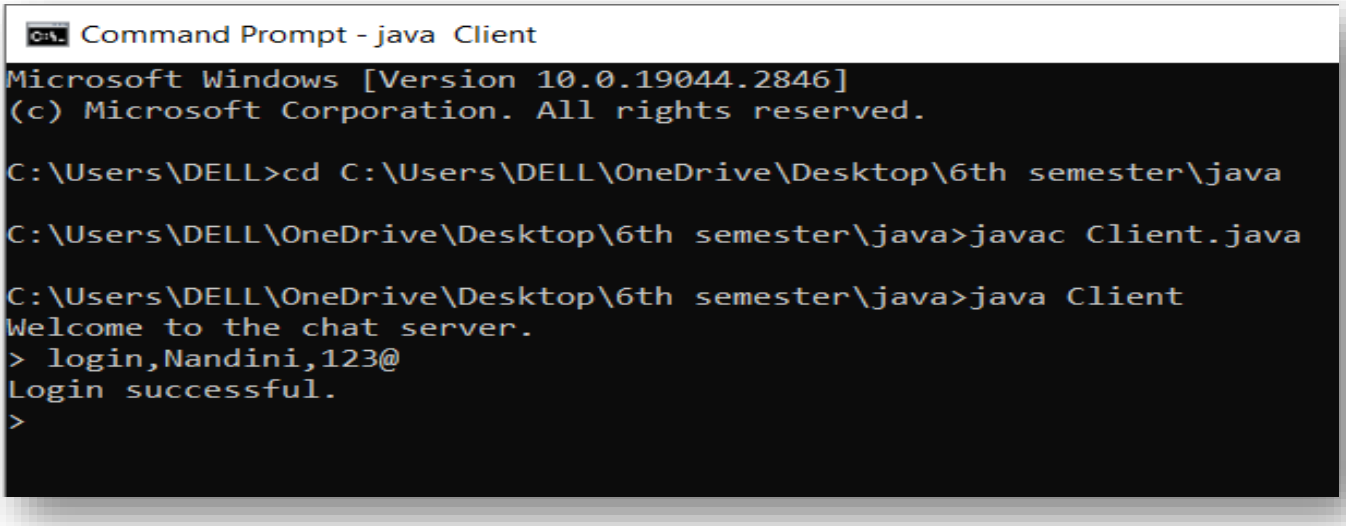
```
                    }
                    } else if (action.equals("delete")) {
                    if (authenticatedClients.contains(username)) {
                    String data = parts[1];
                    if (deleteData(data)) {
                    writer.write("Data deleted successfully.\n");
                    writer.flush();
                    } else {
                    writer.write("Failed to delete data.\n");
                    writer.flush();
                    }
                    } else {
                    writer.write("You must be logged in to perform this action.\n");
                    writer.flush();
                   }
                   } else if (action.equals("update")) {
                   if (authenticatedClients.contains(username)) {
                   String oldData = parts[1];
                   String newData = parts[2];
                   if (updateData(oldData, newData)) {
                   writer.write("Data updated successfully.\n");
                   writer.flush();
                   } else {
                   writer.write("Failed to update data.\n");
                   writer.flush();
                    }
                   } else {
                   writer.write("You must be logged in to perform this action.\n");
                   writer.flush();
                  }
                  } else if (action.equals("get")) {
                  List<String> lines = Files.readAllLines(Paths.get(DATA_FILE));
                  for (String dataline : lines) {
                  writer.write(dataline + "\n");
                  writer.flush();
                  }
                  } else {
                  writer.write("Invalid action.\n");
                  writer.flush();
                  }
              }
             } catch (IOException e) {
               e.printStackTrace();
              }
             }
            }
}
```

OUTPUT-

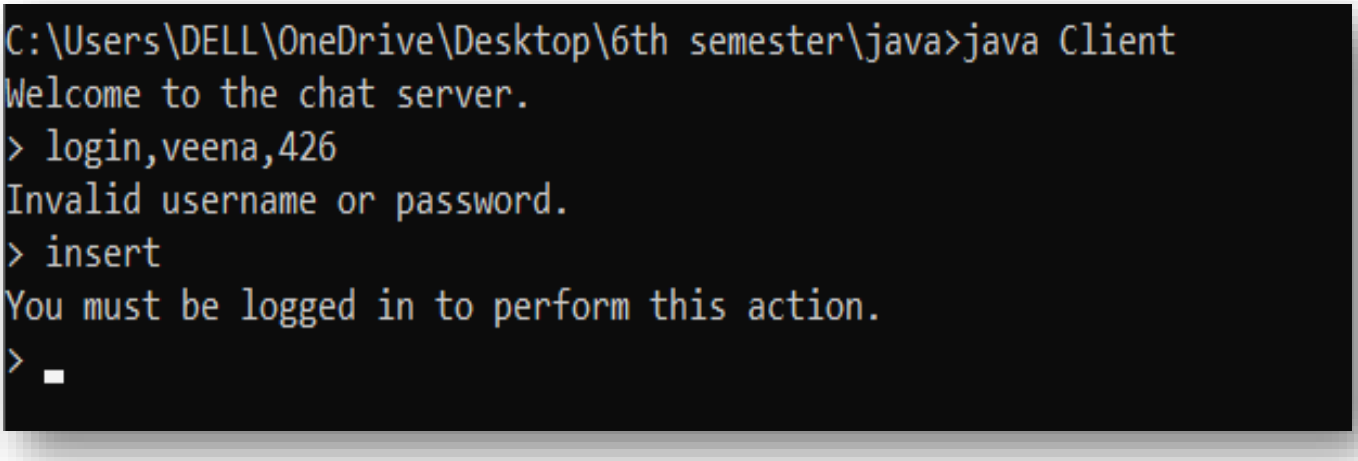
Here the server is giving the output as shown below after the execution of the program-



In the client side the user can login by using the credentials –



If the username or password will be wrong and still after that client want to give commands to server it will say to login again as shown below



After that various command are given and the server is implementing on them and giving the successful message as shown below,

```
C:\Users\DELL\OneDrive\Desktop\6th semester\java>javac Client.java

C:\Users\DELL\OneDrive\Desktop\6th semester\java>java Client
Welcome to the chat server.
> login,Nandini,123@
Login successful.
> insert,priya
Data inserted successfully.
> update,453#,253@
Failed to update data.
> update,Nandini,nandini
Data updated successfully.
> delete,priya
Data deleted successfully.
> logout
Logout successful.
>
```

By the get command one can see the data stored in the data.txt file.

```
C:\Users\DELL\OneDrive\Desktop\6th semester\java>java Client
Welcome to the chat server.
> login,veena,426
Invalid username or password.
> insert
You must be logged in to perform this action.
> login,neeraj,456
Login successful.
> get
nandini
123@
veena
453#
```

CONCLUSION-

The project can be enhanced by using GUI and making the interfaces by using swings, but here we have used the java socket programming which is helping to communicate between the client and server, and having the login credentials for the client after which it can give various command to server for insertion, deletion, and updating the data in on the server side by using the file handling.