

Chat Administration Filtering

Network programming

Dr. Erdemir

Team:

Charmaine Adegbite

Richard Chimbara

Table of Contents

| | |
|--|----|
| Summary..... | 3 |
| Introduction..... | 4 |
| Objectives..... | 4 |
| Design..... | 5 |
| Server | 5 |
| Client..... | 5 |
| Figure 1: Flowchart | 6 |
| Components | 6 |
| Figure 2: Client input | 7 |
| Figure 3: Server receives message..... | 8 |
| Results..... | 9 |
| Figure 4: Connection establishment..... | 9 |
| Figure 5: Inappropriate word replaced with asterisks(*)..... | 9 |
| Figure 6: Conditional message decline | 9 |
| Conclusion..... | 10 |
| Appendix A - Server Code | 11 |
| Appendix B - Client Code | 14 |
| References | 17 |

Summary

This report demonstrates how we built a small-scale TCP/IP chat filtering administration programme in c#. The programme enables communication between clients by a server which acts as a chat administrator with filtering functionality to detect inappropriate or harmful words and censors them in real time. By using multithreading, the server can allow for communication to occur concurrently. The server uses a condition to decline or block messages that have many inappropriate words.

Introduction

Effective digital communication is essential in today's connected world, but it's equally important to maintain safe and respectful conversations which has proven to be an issue in the digital world (Gupta, 2017). In response to this need, we used our knowledge from our network programming lessons to design and implement a TCP/IP chat filtering and administration program developed in C# to detect inappropriate words. This report will show the design, implementation, and functionality of the program, highlighting its key features and mechanisms.

Objectives

- To create a multi thread chat
- To filter bad words from client messages
- To provide a conditional message decline

The goal of the chat administration filtering project is to create a multi thread chat. We will use TCP/IP to implement this. To ensure security, each client's message will be sent to the administrating server.

The administrating server will filter for specific words which will be censored but the message will be sent to the receiving client.

The administrating server will count how many inappropriate words are in a message and block the message if too many words are found.

Design

The chat administration program will do the following:

Server

Accepts client connections through ports

Assign a client with a client number

Receives and listens to messages from the connected clients

Reviews words in messages to identify inappropriate words against a txt file

Replaces inappropriate words with an asterisk (*)

Sends the messages received from each client to the other client connected to the server.

Blocks messages with 4 or more bad words

Client

Connect to the server through the server's IP address

Accept a client number from the server

Send a message to the server

Receive a message from the server

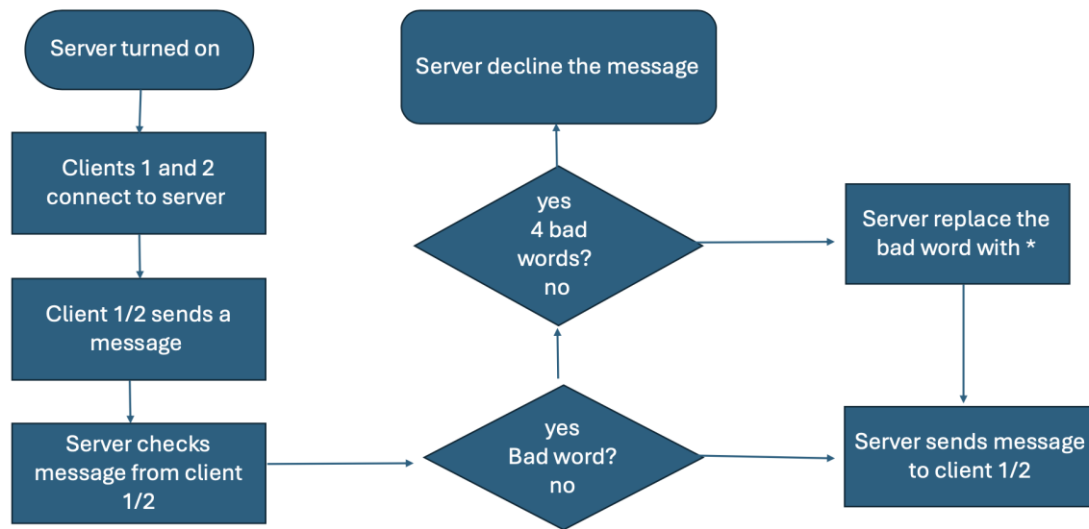


Figure 1: Flowchart

Components

The program uses a client-server architecture, where the server mediates communication between two clients over a TCP/IP network (Stevens, 1994).

Multithreading is used to handle concurrent client connections, where each client is assigned a dedicated thread. This configuration allows the server to manage multiple client connections simultaneously, increasing scalability and responsiveness.

The core of the program is the TCP/IP socket protocol, which provides reliable and stream-oriented communication between server and client. The server establishes the TCP socket and listens, waiting for incoming client connections. In networking, two-way data exchange takes place over a TCP/IP connection, enabling real-time communication between clients (Postel, 1981). The server initializes the socket using the Socket class, and configures it with the appropriate address family, socket type, and protocol type. `IPEndPoint` is configured to define the server endpoint, which specifies the server's IP address (localhost) and the specified port number (9050). The server socket is then bound to the endpoint and configured to listen for incoming connections with the specified backlog.

The design of the program is for it to only begin with the server being turned on and waiting for clients to connect. Following this, the clients connect to the server. Due to TCP being connection-oriented, a continuous connection is made between the server and each client for the duration of the entire communication session. This makes it easier to continuously exchange data without the need to establish connections repeatedly, improving efficiency.

Using TCP sockets, the program establishes persistent communication with each client, providing seamless communication sessions. The server handles each client connection in a separate thread to facilitate simultaneous communication with multiple clients. Built-in TCP channels for flow control, error detection, and obstacle avoidance ensure the accuracy and efficiency of data transmission, which is critical to maintaining a responsive conversation environment (Donahoo & Calvert, 2009).

Once the clients are connected to the server, a client can send a message to the server. User input is taken from the console using `Console.ReadLine()` in the main thread of the client program. The input message is then transferred to a byte array using ASCII encoding and sent to the server using the `Send` method of the client socket. This function allows the client to send arbitrary messages or commands to the server for processing.

```
while (true)
{
    input = Console.ReadLine();
    server.Send(Encoding.ASCII.GetBytes(input));
}
```

Figure 2: Client input

As mentioned earlier, the server utilises threads to manage communication with each client. In the `HandleClient` method, the server keeps listening for incoming messages from the client using the `Receive` method.

```

static void HandleClient(Socket clientSocket, Socket otherClientSocket)
{
    int recv;
    byte[] data = new byte[1024];

    while (true)
    {
        data = new byte[1024];
        recv = clientSocket.Receive(data);
        if (recv == 0)
            break;

        string receivedMessage = Encoding.ASCII.GetString(data, 0, recv);
        string filteredMessage = FilterBadWords(receivedMessage);
        Console.WriteLine(filteredMessage);

        // Check if the message is blocked due to containing four or more bad words
        bool messageBlocked = filteredMessage == "Bad words message has been blocked";

        // Send feedback to both clients if the message is blocked
        if (messageBlocked)
        {
            byte[] feedbackData = Encoding.ASCII.GetBytes(filteredMessage);
            clientSocket.Send(feedbackData, feedbackData.Length, SocketFlags.None);
            otherClientSocket.Send(feedbackData, feedbackData.Length, SocketFlags.None);
            Console.WriteLine("Feedback sent to both clients: " + filteredMessage);
        }
        else
        {
            // Forward message to the other client
            byte[] filteredData = Encoding.ASCII.GetBytes(filteredMessage);
            otherClientSocket.Send(filteredData, filteredData.Length, SocketFlags.None);
        }
    }

    IPEndPoint clientEndPoint = (IPEndPoint)clientSocket.RemoteEndPoint;
    Console.WriteLine("Disconnected from {0}", clientEndPoint.Address);
    clientSocket.Close();
}

```

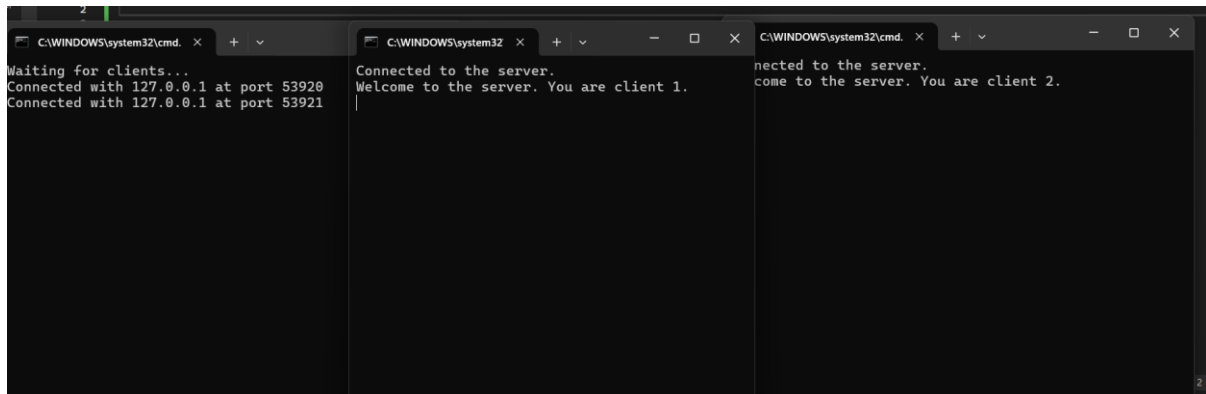
Figure 3: Server receives message

The received data is converted from bytes to strings using ASCII encoding and passed to the `FilterBadWords` method for content processing. Inappropriate words are replaced with asterisks (*), and messages are evaluated for blocking based on the predefined criteria of 4 or more inappropriate words.

The `FilterBadWords` method reads a list of bad words from a txt file and counts the occurrences in retrieved messages using regular expressions. Messages containing four or more negative words are blocked, and the response is sent to all recipients. Otherwise, the message is updated with asterisks to replace the inappropriate words before being sent to the receiving client.

Results

Figure 4 illustrates 2 clients using ports to connect to a server and are assigned a client number upon connection.



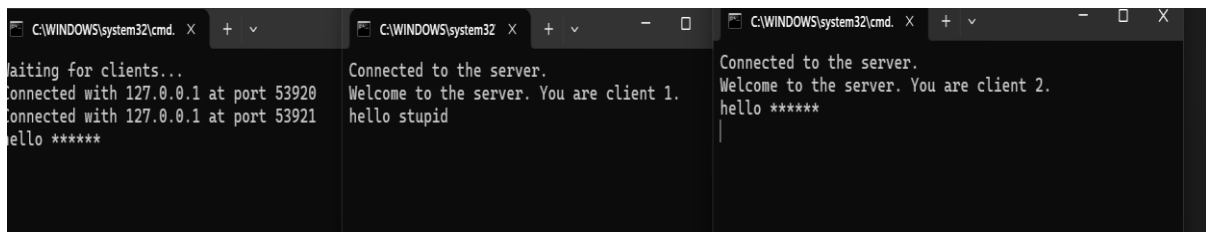
```
C:\WINDOWS\system32\cmd. x + v
Waiting for clients...
Connected with 127.0.0.1 at port 53920
Connected with 127.0.0.1 at port 53921

C:\WINDOWS\system32 x + v - □ X
Connected to the server.
Welcome to the server. You are client 1.

C:\WINDOWS\system32\cmd. x + v - □ X
Connected to the server.
Welcome to the server. You are client 2.
```

Figure 4: Connection establishment

Figure 5 illustrates the inappropriate words sent from a client to a server. The server listens and identifies the inappropriate word, replaces the word with asterisks and forwards the message to the recipient client.



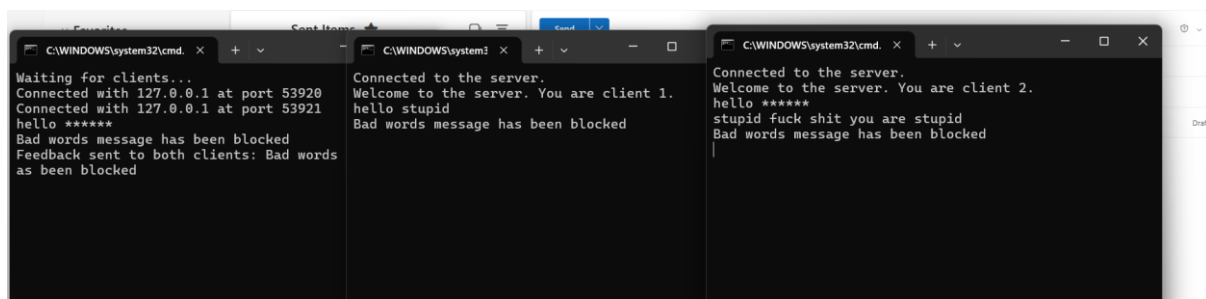
```
C:\WINDOWS\system32\cmd. x + v
Waiting for clients...
Connected with 127.0.0.1 at port 53920
Connected with 127.0.0.1 at port 53921
hello *****

C:\WINDOWS\system32 x + v - □ X
Connected to the server.
Welcome to the server. You are client 1.
hello stupid

C:\WINDOWS\system32\cmd. x + v - □ X
Connected to the server.
Welcome to the server. You are client 2.
hello *****
```

Figure 5: Inappropriate word replaced with asterisks(*)

Figure 6 illustrates the server checking for a condition of 4 or more inappropriate words and blocking the message from being sent to the recipient client.



```
C:\WINDOWS\system32\cmd. x + v
Waiting for clients...
Connected with 127.0.0.1 at port 53920
Connected with 127.0.0.1 at port 53921
hello *****
Bad words message has been blocked
Feedback sent to both clients: Bad words
as been blocked

C:\WINDOWS\system32 x + v - □ X
Connected to the server.
Welcome to the server. You are client 1.
hello stupid
Bad words message has been blocked

C:\WINDOWS\system32\cmd. x + v - □ X
Connected to the server.
Welcome to the server. You are client 2.
hello *****
stupid fuck shit you are stupid
Bad words message has been blocked
```

Figure 6: Conditional message decline

Conclusion

In conclusion, our chat filtering program was successful. The server efficiently receives and processes messages from clients in the context of TCP/IP connections. By using socket programming and multithreading, the server enforced information handling protocols to ensure simultaneous communication with multiple clients. Additionally, the server correctly identified inappropriate words and replaced them with asterisks(*) and forwards the message, with the inappropriate words replaced, to the recipient. The addition of the conditional message decline is useful especially in the context of the digital community. We considered how to improve the program. Restricting the chat to only a limited number of clients would provide additional security. Furthermore, future work could extend the conditional message blocking feature to file sharing. This would be useful in the context of cybersecurity.

Appendix A - Server Code

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;

class SimpleTcpSrvr
{
    static Socket clientSocket1;
    static Socket clientSocket2;

    public static void Main()
    {
        // Set up server socket
        Socket serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
        IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 9050);

        serverSocket.Bind(endPoint);
        serverSocket.Listen(10);
        Console.WriteLine("Waiting for clients...");
        // Accept first client
        clientSocket1 = serverSocket.Accept();
        IPEndPoint clientEndPoint1 = (IPEndPoint)clientSocket1.RemoteEndPoint;
        Console.WriteLine("Connected with {0} at port {1}", clientEndPoint1.Address,
        clientEndPoint1.Port);

        // Send welcome message to client 1
        string welcomeMessage1 = "Welcome to the server. You are client 1.";
        byte[] welcomeData1 = Encoding.ASCII.GetBytes(welcomeMessage1);
        clientSocket1.Send(welcomeData1, welcomeData1.Length, SocketFlags.None);

        // Accept second client
        clientSocket2 = serverSocket.Accept();
        IPEndPoint clientEndPoint2 = (IPEndPoint)clientSocket2.RemoteEndPoint;
        Console.WriteLine("Connected with {0} at port {1}", clientEndPoint2.Address,
        clientEndPoint2.Port);
    }
}
```

```

// Send welcome message to client 2
string welcomeMessage2 = "Welcome to the server. You are client 2.";
byte[] welcomeData2 = Encoding.ASCII.GetBytes(welcomeMessage2);
clientSocket2.Send(welcomeData2, welcomeData2.Length, SocketFlags.None);

// Handle clients in separate threads
Thread clientThread1 = new Thread(() => HandleClient(clientSocket1, clientSocket2));
clientThread1.Start();

Thread clientThread2 = new Thread(() => HandleClient(clientSocket2, clientSocket1));
clientThread2.Start();
}

static void HandleClient(Socket clientSocket, Socket otherClientSocket)
{
    int rcv;
    byte[] data = new byte[1024];

    while (true)
    {
        data = new byte[1024];
        rcv = clientSocket.Receive(data);
        if (rcv == 0)
            break;

        string receivedMessage = Encoding.ASCII.GetString(data, 0, rcv);
        string filteredMessage = FilterBadWords(receivedMessage);
        Console.WriteLine(filteredMessage);

        // Check if the message is blocked due to containing four or more bad words
        bool messageBlocked = filteredMessage == "Bad words message has been blocked";

        // Send feedback to both clients if the message is blocked
        if (messageBlocked)
        {
            byte[] feedbackData = Encoding.ASCII.GetBytes(filteredMessage);
            clientSocket.Send(feedbackData, feedbackData.Length, SocketFlags.None);
            otherClientSocket.Send(feedbackData, feedbackData.Length, SocketFlags.None);
            Console.WriteLine("Feedback sent to both clients: " + filteredMessage);
        }
        else
        {
            // Forward message to the other client
            byte[] filteredData = Encoding.ASCII.GetBytes(filteredMessage);

```

```

        otherClientSocket.Send(filteredData, filteredData.Length, SocketFlags.None);
    }
}

IPEndPoint clientEndPoint = (IPEndPoint)clientSocket.RemoteEndPoint;
Console.WriteLine("Disconnected from {0}", clientEndPoint.Address);
clientSocket.Close();
}

static string FilterBadWords(string message)
{
    // Dictionary to store the count of each bad word
    Dictionary<string, int> badWordCounts = new Dictionary<string, int>();

    // Read bad words from file
    string filePath = "badwords.txt";
    string[] badWords = File.ReadAllLines(filePath);

    foreach (string badWord in badWords)
    {
        // Count occurrences of each bad word in the message
        int count = Regex.Matches(message, "\\b" + badWord + "\\b").Count;
        badWordCounts[badWord] = count;
    }

    foreach (var entry in badWordCounts)
    {
        string badWord = entry.Key;
        int count = entry.Value;

        // If any bad word occurs four or more times, block the message
        if (count >= 4)
        {
            return "Bad words message has been blocked";
        }

        // Replace bad words with asterisks
        string filteredWord = new string('*', badWord.Length);
        message = message.Replace(badWord, filteredWord);
    }

    return message;
}
}

```

Appendix B - Client Code

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

class SimpleTcpClient
{
    static Socket server;
    static byte[] data = new byte[1024];
    static string input;

    public static void Main()
    {
        IPEndPoint ipep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
        server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

        try
        {
            server.Connect(ipep);
            Console.WriteLine("Connected to the server.");
        }
        catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server.");
            Console.WriteLine(e.ToString());
            return;
        }

        // Receive welcome message from server
        int recv = server.Receive(data);
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        // Start thread for receiving messages
        Thread receiveThread = new Thread(ReceiveMessage);
        receiveThread.Start();

        // Main thread for sending messages
        while (true)
        {
            input = Console.ReadLine();
            server.Send(Encoding.ASCII.GetBytes(input));
        }
    }
}

```

```
static void ReceiveMessage()
{
    while (true)
    {
        int recv = server.Receive(data);
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
    }
}
```


References

Gupta, Shubhankar. (2017). Detection and Elimination of Censor Words on Online Social Media.

Stevens, W. R. (1994). TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley.

Postel, J. (1981). Transmission Control Protocol. RFC 793. Internet Engineering Task Force says about the TCP/IP socket protocol.

Donahoo, M. J., & Calvert, K. L. (2009). TCP/IP Sockets in C: Practical Guide for Programmers. Morgan Kaufmann.