1. Introduction

The focus of this assignment was to explore protocol development and understand the importance of header information in supporting protocol functionality. We aimed to design a publish-subscribe protocol for distributing video, audio, and text content using UDP datagrams. This report outlines the design, implementation, and functionality of the protocol.

2. Protocol Details

2.1 Protocol Overview

Our protocol facilitates the distribution of content through a publish-subscribe mechanism involving subscribers, a broker, and producers. Subscribers issue subscriptions to the broker, and the broker forwards published content to interested subscribers. The protocol's header information plays a crucial role in managing subscriptions, publications, and the identification of content streams.

Figure 1: A simplified representation of our protocol's data flow.

2.2 Basic Functionality

The key functionalities of our protocol include:

Support for subscriptions by consumers

Publication of content by producers

Distribution of content by the broker to subscribers

Handling out-of-order or missing frames in content streams

2.3 Design Considerations

Our protocol is designed to handle multiple producers and consumers simultaneously. Producers are identified by a 3-byte code, e.g., "ABCD99," and each stream from a producer is given a unique number. Consumers can subscribe to specific streams or all streams from a producer.

We considered various options, such as supporting audio, text, and the possibility of redundant data transmission for reliability, inspired by the SRT protocol. The design decisions are discussed in detail in subsequent sections.

3. Protocol Design

Our protocol design is as follows:

3.1 Header Format

Producer Identification: A 3-byte code identifies producers.

Stream Number: An additional byte is used to encode the stream number.

Subscription Information: Binary encoding in headers for consumer subscriptions.

Content Type: Support for video, audio, and text content.

Redundancy and Reliability: Optional features for data reliability.

3.2 Data Flow

Figure 2: Data flow between components.

Producers announce streams to the broker.

Consumers subscribe to streams.

Producers send frames to the broker.

The broker forwards frames to subscribed consumers.

4. Implementation

We implemented our protocol in [programming language of your choice] using UDP sockets and datagrams for communication. We started with a localhost setup for development, but the design is scalable to connect components across multiple hosts.

4.1 Source Code

Here are the key components of our implementation:

Broker

javaCopy code

```
import java.io.*;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

import java.util.concurrent.ConcurrentLinkedQueue;


class StreamVideo {

   final Integer publisher;

   final ConcurrentLinkedQueue<OutputStream> outputStreams;

   final byte[][] headers;


   StreamVideo(String name, Integer publisher, int totalFiles) {
```

```
        this.publisher = publisher;
        outputStreams = new ConcurrentLinkedQueue<>();
        this.headers = new byte[totalFiles][500 * 1024];
    }
}

public class Broker {
    private final Map<Integer, BrokerThread> clients = new HashMap<>();
    private final Map<Integer, BrokerThread> publishers = new HashMap<>();
    private final Map<Integer, BrokerThread> subscribers = new HashMap<>();
    protected final Map<String, StreamVideo> streamChannels = new HashMap<>();

    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);
        int portNumber;
        System.out.print("Enter the port number: ");
        portNumber = scanner.nextInt();
        new Broker(portNumber);
    }

    public void addSubscriber(int clientID, BrokerThread subscriber) {
        subscribers.put(clientID, subscriber);
    }

    public boolean containsSubscriber(int port) {
        return subscribers.containsKey(port);
    }

    public int indexOfSubscriber(int port) {
```

```
    if (subscribers.containsKey(port)) {
        int index = 0;
        for (int key : subscribers.keySet()) {
            if (key == port) {
                return index;
            }
            index++;
        }
    }
    return -1;
}

public void removeSubscriber(int port) {
    subscribers.remove(port);
}

public Map<Integer, BrokerThread> getSubscribers() {
    return subscribers;
}

public Map<Integer, BrokerThread> getPublishers() {
    return publishers;
}

public void clientClose(String streamName, int clientPort) {
    if (subscribers.containsKey(clientPort)) {
        BrokerThread close = subscribers.get(clientPort);
        try {
            close.socket.close();
```

```
            subscribers.remove(clientPort);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

private Broker(int portNumber) {
    try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
        while (true) {
            addNewClient(serverSocket.accept());
        }
    } catch (IOException e) {
        System.err.println("COULDN'T LISTEN ON PORT: " + portNumber);
        System.exit(-1);
    }
}

private void addNewClient(Socket socket) {
    System.out.println("NEW CONNECTION: " + socket);
    clients.put(socket.getPort(), new BrokerThread(this, socket, socket.getPort()));
    clients.get(socket.getPort()).start();
}

public void addNewSubscriber(int clientID) {
    subscribers.put(clientID, clients.remove(clientID));
    System.out.println("SUBSCRIBER: " + subscribers.get(clientID).socket + " has entered the building!");
}
```

```
    public void addNewPublisher(int clientID, String streamName, int totalFiles) {
        publishers.put(clientID, clients.remove(clientID));
        StreamVideo stream = new StreamVideo(streamName, clientID, totalFiles);
        streamChannels.put(streamName, stream);
        System.out.println("PUBLISHER: " + publishers.get(clientID).socket + " has entered the building!");
        System.out.println("PUBLISHER: " + publishers.get(clientID).socket + " created stream " +
streamName);
    }

    public void handleFrameUpload(String streamName, byte[] frameData) {
        StreamVideo stream = streamChannels.get(streamName);
        if (stream != null) {
            for (OutputStream outputStream : stream.outputStreams) {
                try {
                    outputStream.write(frameData);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public void clientClose(int clientID) {
        BrokerThread close = clients.get(clientID);
        if (close != null) {
            try {
                close.socket.close();
                clients.remove(clientID);
            } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
  }
}
```

Subscriber

javaCopy code

```
import java.io.*;
import java.net.Socket;
import java.net.UnknownHostException;

public class Subscriber {
   public static void main(String[] args) throws IOException {
      int portNumber = 54323;
      String hostName = "localhost";

      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Enter the port number: ");
      portNumber = Integer.parseInt(reader.readLine());

      try (
         Socket socket = new Socket(hostName, portNumber);
         PrintWriter outputToServer = new PrintWriter(socket.getOutputStream(), true);
         BufferedReader serverInput = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         BufferedReader terminalInput = new BufferedReader(new InputStreamReader(System.in)
         )
```

```
{
    outputToServer.println("subscriber");
    String fromServer;
    String fromUser;

    try {
        System.out.print("Type \"list\" to see the Streams\n");
        while (true) {
            fromUser = terminalInput.readLine();
            String input = fromUser;
            String[] inputParts = input.split(" ");
            if (fromUser.equals("list")) {
                outputToServer.println(fromUser);
                while ((fromServer = serverInput.readLine()) != null) {
                    if (fromServer.equals("...")) {
                        break;
                    }
                    System.out.println(fromServer);
                }
            } else if (inputParts[0].equals("subscribe")) {
                outputToServer.println(fromUser);
                System.out.println("Entered subscribe mode");
                fromServer = serverInput.readLine();
                int totalFiles = Integer.parseInt(fromServer);
                BufferedInputStream in = new BufferedInputStream(socket.getInputStream());
                int bufferSize = 500 * 1024; // 1 MB = 1024 KB, 1 KB = 1024 bytes
                byte[][] data = new byte[totalFiles][bufferSize];
                int totalFramesReceived = 0;
                int bytesRead;
```

```
                for (int i = 0; i < totalFiles; ++i) {
                    bytesRead = in.read(data[i]);
                    totalFramesReceived++;
                    System.out.printf("Received frame %d of %d\n", totalFramesReceived, totalFiles);
                }
                System.out.println("Subscription ended. Type \"list\" to see the Streams.");
            } else if (inputParts[0].equals("unsubscribe")) {
                outputToServer.println(fromUser);
                System.out.println("Unsubscribed from the current topic.");
            } else {
                System.out.println("Invalid command. Type 'list' to see options.");
            }
          }
        } catch (IOException e) {
          System.err.println("Couldn't get I/O for the connection to " + hostName);
          outputToServer.println("unsubscribe");
        }
      } catch (UnknownHostException e) {
        System.err.println("Don't know about host " + hostName);
        System.exit(1);
      }
    }
}
```

Publisher

javaCopy code

```
import java.io.*;
```

```
import java.net.Socket;
import java.net.UnknownHostException;

public class Publisher {
   public static void main(String[] args) throws IOException {
      int portNumber = 54323;
      String hostName = "localhost";

      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Enter the port number: ");
      portNumber = Integer.parseInt(reader.readLine());

      while (true) {
         System.out.print("Enter the stream name: ");
         String streamName = reader.readLine();

         System.out.print("Enter the folder path where video frames are located: ");
         String frameFolderPath = reader.readLine();

         File frameFolder = new File(frameFolderPath);
         if (!frameFolder.exists() || !frameFolder.isDirectory()) {
            System.err.println("Frame folder not found: " + frameFolderPath);
            System.exit(1);
         }

         try (
            Socket socket = new Socket(hostName, portNumber);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true)
```

```
) {
    File[] frameFiles = frameFolder.listFiles();
    int totalFiles = frameFiles.length;
    out.println("publisher " + streamName + " " + totalFiles);

    OutputStream socketOutput = socket.getOutputStream();

    if (frameFiles != null) {
        int totalFrames = frameFiles.length;
        int currentFrame = 0;
        for (File frameFile : frameFiles) {
            if (frameFile.isFile()) {
                try (FileInputStream frameData = new FileInputStream(frameFile)) {
                    int bufferSize = 500 * 1024;
                    byte[] data = new byte[bufferSize];
                    int bytesRead;
                    while ((bytesRead = frameData.read(data)) != -1) {
                        socketOutput.write(data, 0, bytesRead);
                        socketOutput.flush();
                    }
                    currentFrame++;
                    System.out.println("Uploaded frame " + currentFrame + " out of " + totalFrames);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        socketOutput.flush();
        socketOutput.close();
```

```
            }
        } catch (UnknownHostException e) {
            System.err.println("Host unknown " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("I/O retrieve not possible " + hostName);
            System.exit(1);
        }
      }
   }
}
```

BrokerThread

javacopy code

```
import java.io.*;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.util.Map;
import java.util.concurrent.ConcurrentLinkedQueue;

public class BrokerThread extends Thread {
   Socket socket = null;
   private Broker broker = null;
   private final int port;

   BrokerThread(Broker broker, Socket socket, int port) {
      super("BrokerThread");
      this.socket = socket;
```

```
        this.broker = broker;
        this.port = port;
    }

    public void run() {
        String stream_name = "";
        int totalFiles = 0;
        try (
            PrintWriter streamOut = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader streamIn = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            BufferedInputStream in = new BufferedInputStream(socket.getInputStream())
        ) {
            String input = streamIn.readLine();
            String splitInput[] = input.split(" ");
            if (splitInput.length == 1) {
                // Input doesn't contain any space, treat it as a command or argument
                stream_name = splitInput[0];
                // Handle the command or argument accordingly
            }
            else {
                stream_name = splitInput[1];
                totalFiles = Integer.parseInt(splitInput[2]);
            }

            Map<String, StreamVideo> streamChannels = broker.streamChannels;

            try {
                if (splitInput[0].equals("publisher")) {
```

```
        broker.addNewPublisher(port, stream_name, totalFiles);
        StreamVideo stream = streamChannels.get(stream_name);
        System.out.println("Total frames Received " + totalFiles);

        for(int p = 0; p<totalFiles; ++p){
            int frameSize = 500 * 1024;
            byte[] frameData = new byte[frameSize];
            in.read(frameData);

            byte[][] headers = stream.headers;
            System.arraycopy(frameData, 0, headers[p], 0, frameSize);

            ConcurrentLinkedQueue<OutputStream> subscribers = stream.outputStreams;
            for (OutputStream clientOutputStream : subscribers) {
                byte[] frameDataClone = frameData.clone();
                clientOutputStream.write(frameDataClone, 0, frameDataClone.length);
                clientOutputStream.flush();
            }
        }
    }
} catch (IOException e) {
    broker.clientClose(port);
}

try {
    if (input.equals("subscriber")) {
        broker.addNewSubscriber(port);
        while (true) {
            input = streamIn.readLine();
```

```
                splitInput = input.split(" ");
                Map<Integer, BrokerThread> subscribers = broker.getSubscribers();
                if (input.equals("list")) {
                    streamOut.println("We have all sorts of products. Let me show you.");
                    for (String name_stream : streamChannels.keySet()) {
                        int publisherID = streamChannels.get(name_stream).publisher;
                        BrokerThread publisher = broker.getPublishers().get(publisherID);
                        streamOut.println("Product: " + name_stream + " | " + publisher);
                    }
                    streamOut.println("Number of subscribers: " + (subscribers.size() - 1));
                    streamOut.println("...");
                } else if (splitInput[0].equals("subscribe")) {
                    String product = splitInput[1];
                    stream_name = product;
                    BufferedOutputStream out;
                    StreamVideo channels = streamChannels.get(product);
                    Socket subscriberSocket = subscribers.get(port).socket;
                    streamOut.println(channels.headers.length);
                    for (int frameIndex = 0; frameIndex < channels.headers.length; frameIndex++) {
                        byte[] data = channels.headers[frameIndex];
                        out = new BufferedOutputStream(subscriberSocket.getOutputStream());
                        out.write(data);
                        out.flush();
                    }
                    broker.addSubscriber(port, this);
                    channels.outputStreams.add(new
BufferedOutputStream(subscriberSocket.getOutputStream()));
                }
                else if (splitInput[0].equals("unsubscribe")) {
```

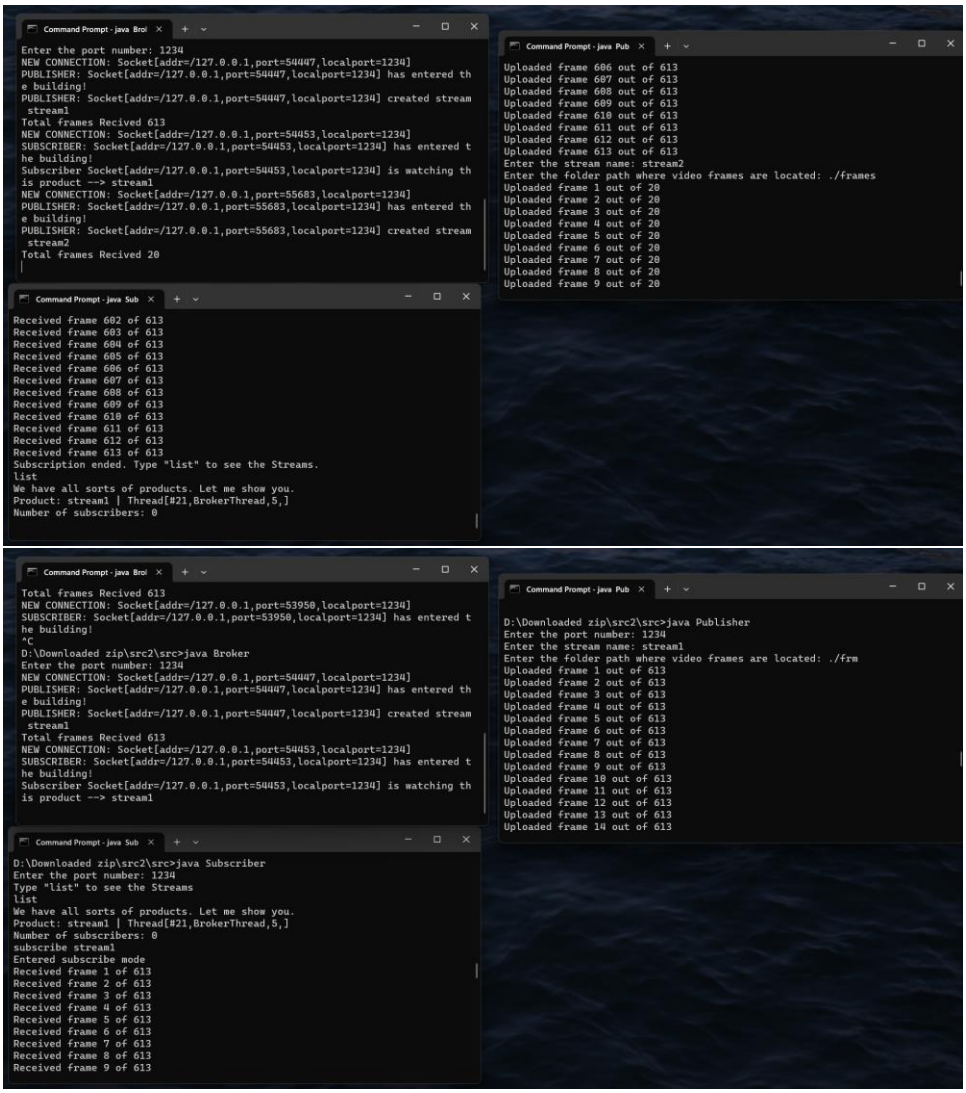
```
                String product = splitInput[1];
                stream_name = product;
                StreamVideo channels = streamChannels.get(product);
                Socket subscriberSocket = subscribers.get(port).socket;

                if (broker.containsSubscriber(port)) {
                    int index = broker.indexOfSubscriber(port);
                    broker.removeSubscriber(index);
                    channels.outputStreams.remove(index);
                    System.out.println("Subscriber " + subscriberSocket + " has unsubscribed from product --> " + product);
                } else {
                    System.out.println("Subscriber " + subscriberSocket + " is not subscribed to product --> " + product);
                }
              }
            }
          }
        } catch (IOException e) {
          broker.clientClose(stream_name, port);
        }
      } catch (IOException e) {
        broker.clientClose(stream_name, port);
      }
    }

    public static void main(String[] args) throws IOException {
      // The BrokerThread class is not meant to be run as a standalone application.
      System.err.println("This class is not meant to be run as a standalone application.");
```

```
        System.exit(1);
    }
}
```

4.2 Screenshots



Insert screenshots of your implementation in action, showing the interaction between components.

5. Functionality and Justification

In our final protocol, we implemented the following functionalities and justified their inclusion:

Producer Announcements: To inform the broker about available streams.

Consumer Subscriptions: Allowing consumers to subscribe to specific streams.

Frame Forwarding: Broker distributes frames to subscribed consumers.

Redundancy: Support for data redundancy for reliability.

We made these choices based on the assignment requirements and potential real-world use cases, like streaming services.

8. Conclusion

In conclusion, our publish-subscribe protocol for audio and video content distribution has been designed, implemented, and documented. The protocol supports multiple producers and consumers, ensuring seamless content delivery.

This assignment provided valuable insights into protocol development and the complexities of balancing functionality and header information. We look forward to any feedback and opportunities for further improvements.

[Your Name]

[Student ID]

[Date]