Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks

Assignment #1: Protocols

Dmitry Kryukov, 20336877

October 27, 2023

Contents

1 Introduction

This report describes how my implementation of the protocol assignment works, details and some problems with corresponding solutions. The task behind the assignment was to implement our own protocol for a live streaming service with producers, consumers, and the broker. Each possible producer can have multiple streams. Consumers can subscribe to those streams and receive files from the producer through the broker.

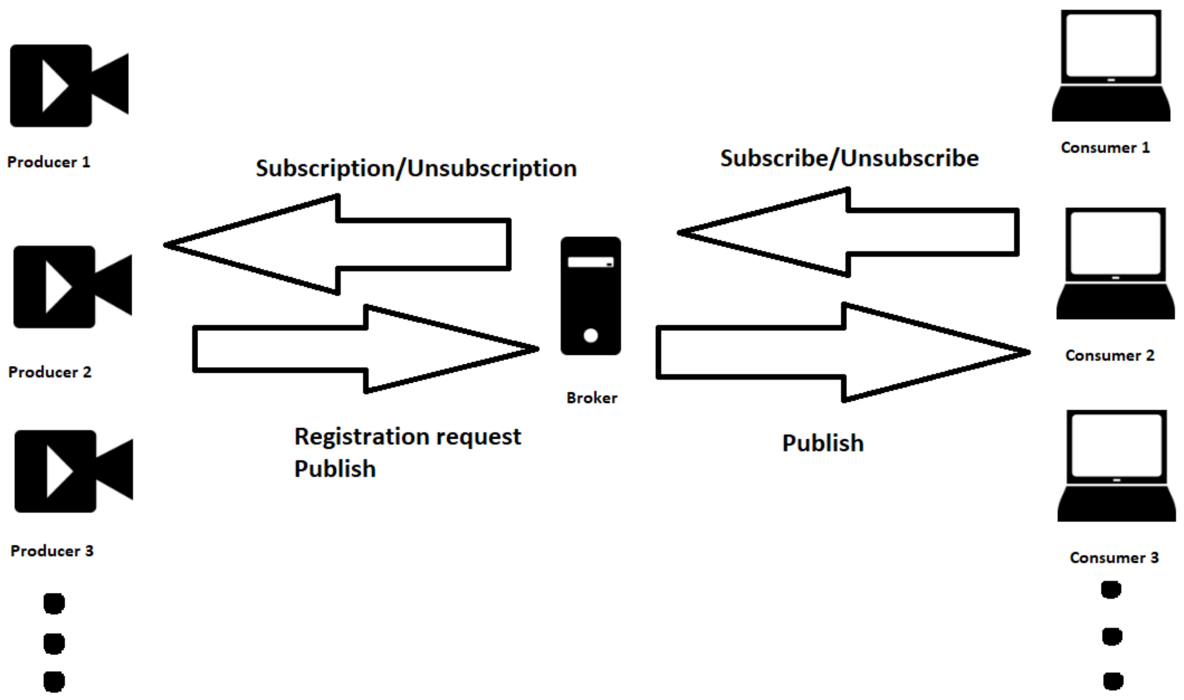
2 Background

The final project was tested on Docker using 3+ dockerfiles and Docker compose for faster startup. Captured traffic can be located inside the containers folder in the Docker. The main code for the protocol is written in Java using several libraries for sockets and datagram packets, which makes it possible to send information and files.

It is possible to test and run this assignment without using Docker, but for the sake of testing on one system, it is preferable. Java was chosen as the main language because of the familiarity with it and how different aspects of it work.

2.1 Topology

The structure of the project is essential for it to work. In the case of this assignment, there are 3 types of actors: Broker, Consumer, and Producer. Any number of consumers and producers can be added if necessary. All of them are connected to the broker, who is responsible for every request and communication between consumers and producers.

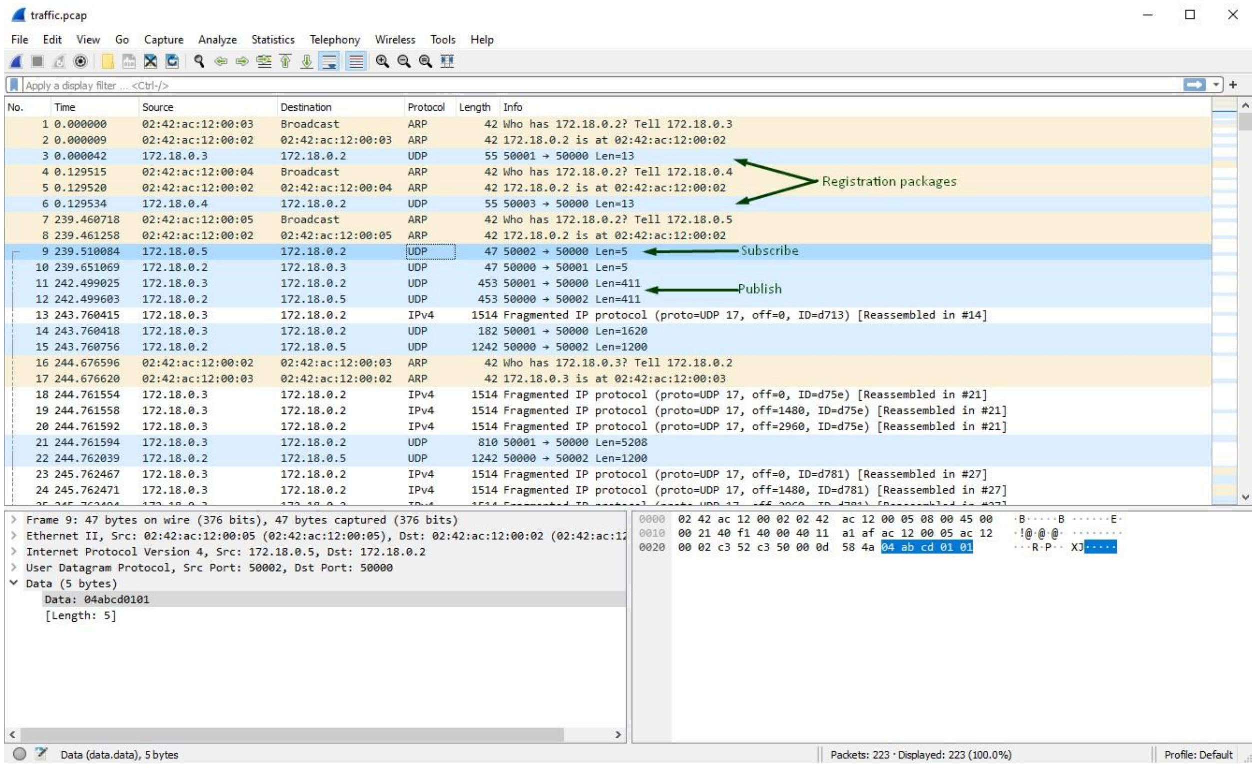


3 Problem Statement

The main task behind this assignment was to implement our own protocol for a live streaming service, which connects producers and consumers to the broker and allows them to communicate with each other in order to send and receive files.

4 implementation

The project works as intended, and every packet is sent to the correct address. The traffic can be seen in the figure below, with the necessary explanation.


traffic.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Registration packages
Subscribe
Publish

```
No. Time        Source             Destination        Protocol Length Info
  1 0.000000    02:42:ac:12:00:03  Broadcast          ARP        42 Who has 172.18.0.2? Tell 172.18.0.3
  2 0.000009    02:42:ac:12:00:02  02:42:ac:12:00:03  ARP        42 172.18.0.2 is at 02:42:ac:12:00:02
  3 0.000042    172.18.0.3         172.18.0.2         UDP        55 50001 → 50000 Len=13
  4 0.129515    02:42:ac:12:00:04  Broadcast          ARP        42 Who has 172.18.0.2? Tell 172.18.0.4
  5 0.129520    02:42:ac:12:00:02  02:42:ac:12:00:04  ARP        42 172.18.0.2 is at 02:42:ac:12:00:02
  6 0.129534    172.18.0.4         172.18.0.2         UDP        55 50003 → 50000 Len=13
  7 239.460718  02:42:ac:12:00:05  Broadcast          ARP        42 Who has 172.18.0.2? Tell 172.18.0.5
  8 239.461258  02:42:ac:12:00:02  02:42:ac:12:00:05  ARP        42 172.18.0.2 is at 02:42:ac:12:00:02
  9 239.510084  172.18.0.5         172.18.0.2         UDP        47 50002 → 50000 Len=5
 10 239.651069  172.18.0.2         172.18.0.3         UDP        47 50000 → 50001 Len=5
 11 242.499025  172.18.0.3         172.18.0.2         UDP       453 50001 → 50000 Len=411
 12 242.499603  172.18.0.2         172.18.0.5         UDP       453 50000 → 50002 Len=411
 13 243.760415  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=d713) [Reassembled in #14]
 14 243.760418  172.18.0.3         172.18.0.2         UDP       182 50001 → 50000 Len=1620
 15 243.760756  172.18.0.2         172.18.0.5         UDP      1242 50000 → 50002 Len=1200
 16 244.676596  02:42:ac:12:00:02  02:42:ac:12:00:03  ARP        42 Who has 172.18.0.3? Tell 172.18.0.2
 17 244.676620  02:42:ac:12:00:03  02:42:ac:12:00:02  ARP        42 172.18.0.3 is at 02:42:ac:12:00:03
 18 244.761554  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=d75e) [Reassembled in #21]
 19 244.761558  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=d75e) [Reassembled in #21]
 20 244.761592  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=d75e) [Reassembled in #21]
 21 244.761594  172.18.0.3         172.18.0.2         UDP       810 50001 → 50000 Len=5208
 22 244.762039  172.18.0.2         172.18.0.5         UDP      1242 50000 → 50002 Len=1200
 23 245.762467  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=d781) [Reassembled in #27]
 24 245.762471  172.18.0.3         172.18.0.2         IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=d781) [Reassembled in #27]
```

```
> Frame 9: 47 bytes on wire (376 bits), 47 bytes captured (376 bits)
> Ethernet II, Src: 02:42:ac:12:00:05 (02:42:ac:12:00:05), Dst: 02:42:ac:12:00:02 (02:42:ac:12
> Internet Protocol Version 4, Src: 172.18.0.5, Dst: 172.18.0.2
> User Datagram Protocol, Src Port: 50002, Dst Port: 50000
v Data (5 bytes)
    Data: 04abcd0101
    [Length: 5]
```

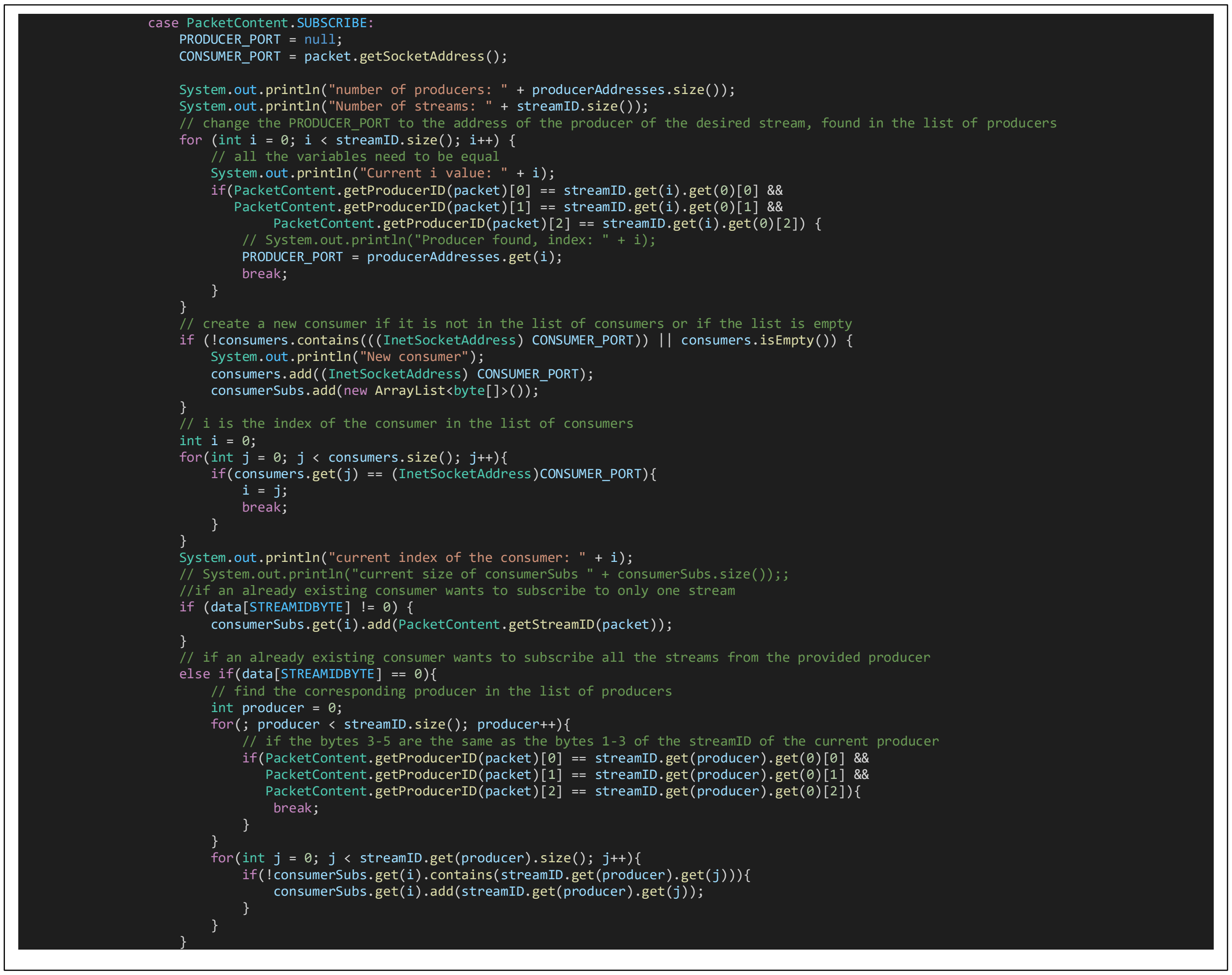
```
0000  02 42 ac 12 00 02 02 42  ac 12 00 05 08 00 45 00   ·B·····B ······E·
0010  00 21 40 f1 40 00 40 11  a1 af ac 12 00 05 ac 12   ·!@·@·@· ········
0020  00 02 c3 52 c3 50 00 0d  58 4a 04 ab cd 01 01      ···R·P·· XJ·····
```

Data (data.data), 5 bytes | Packets: 223 · Displayed: 223 (100.0%) | Profile: Default

The first two transfers are of type Register, so the broker can store streams and producers for further use. What comes next is a subscription request from one of the consumers. After the requested stream owner receives the packet, it starts publishing the stream. Fragmented transfers that can be seen in the screenshot above are because of the file size of the frames used for testing the performance of the system.

One of the main problems was the subscription and stream ID system. In order to store the IDs of streams and producers, I used several ArrayLists. The same goes for consumers and their subscriptions. Whenever a consumer sends a subscription packet, it is remembered on the broker side and then sent further to the producer. When the subscription count of any stream goes above 0, it starts or continues. When it reaches 0 after several unsubscription packets, the stream pauses.

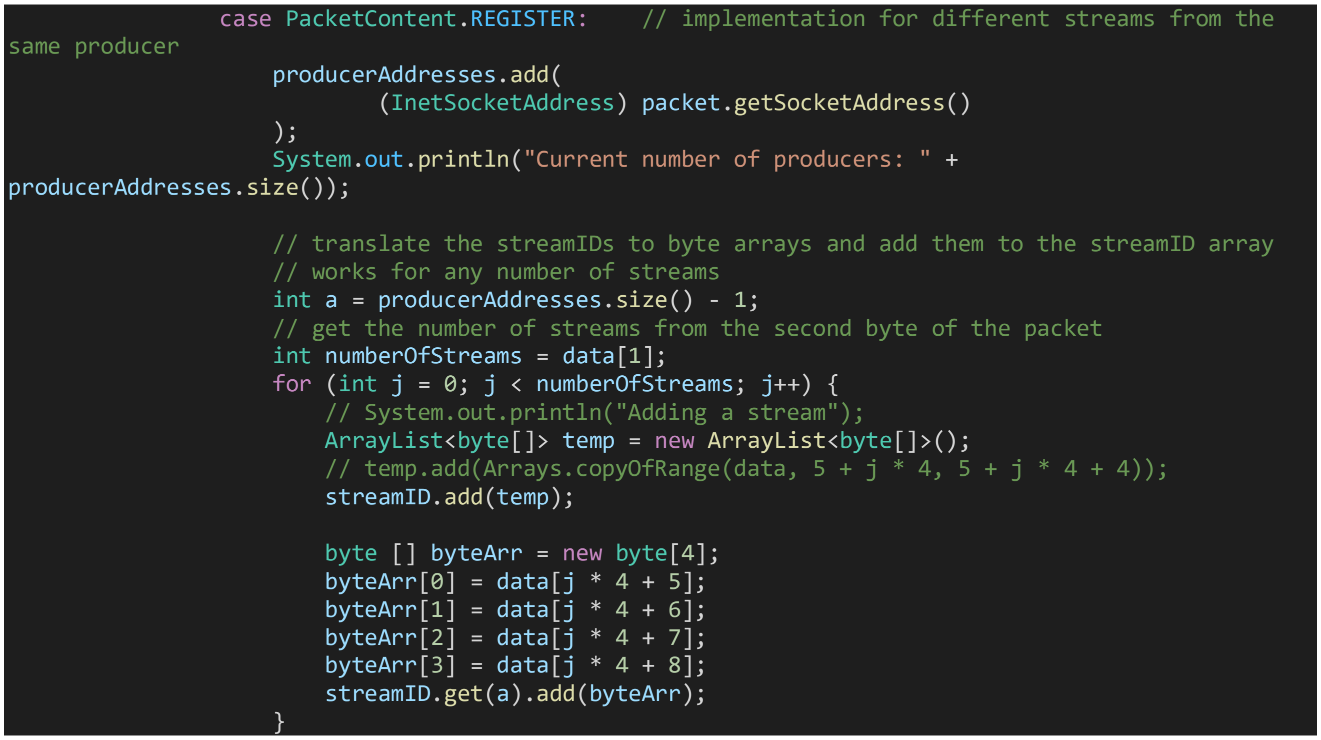
Code:



```
case PacketContent.SUBSCRIBE:
    PRODUCER_PORT = null;
    CONSUMER_PORT = packet.getSocketAddress();

    System.out.println("number of producers: " + producerAddresses.size());
    System.out.println("Number of streams: " + streamID.size());
    // change the PRODUCER_PORT to the address of the producer of the desired stream, found in the list of producers
    for (int i = 0; i < streamID.size(); i++) {
        // all the variables need to be equal
        System.out.println("Current i value: " + i);
        if(PacketContent.getProducerID(packet)[0] == streamID.get(i).get(0)[0] &&
           PacketContent.getProducerID(packet)[1] == streamID.get(i).get(0)[1] &&
                PacketContent.getProducerID(packet)[2] == streamID.get(i).get(0)[2]) {
            // System.out.println("Producer found, index: " + i);
            PRODUCER_PORT = producerAddresses.get(i);
            break;
        }
    }
    // create a new consumer if it is not in the list of consumers or if the list is empty
    if (!consumers.contains(((InetSocketAddress) CONSUMER_PORT)) || consumers.isEmpty()) {
        System.out.println("New consumer");
        consumers.add((InetSocketAddress) CONSUMER_PORT);
        consumerSubs.add(new ArrayList<byte[]>());
    }
    // i is the index of the consumer in the list of consumers
    int i = 0;
    for(int j = 0; j < consumers.size(); j++){
        if(consumers.get(j) == (InetSocketAddress)CONSUMER_PORT){
            i = j;
            break;
        }
    }
    System.out.println("current index of the consumer: " + i);
    // System.out.println("current size of consumerSubs " + consumerSubs.size());;
    //if an already existing consumer wants to subscribe to only one stream
    if (data[STREAMIDBYTE] != 0) {
        consumerSubs.get(i).add(PacketContent.getStreamID(packet));
    }
    // if an already existing consumer wants to subscribe all the streams from the provided producer
    else if(data[STREAMIDBYTE] == 0){
        // find the corresponding producer in the list of producers
        int producer = 0;
        for(; producer < streamID.size(); producer++){
            // if the bytes 3-5 are the same as the bytes 1-3 of the streamID of the current producer
            if(PacketContent.getProducerID(packet)[0] == streamID.get(producer).get(0)[0] &&
               PacketContent.getProducerID(packet)[1] == streamID.get(producer).get(0)[1] &&
               PacketContent.getProducerID(packet)[2] == streamID.get(producer).get(0)[2]){
                break;
            }
        }
        for(int j = 0; j < streamID.get(producer).size(); j++){
            if(!consumerSubs.get(i).contains(streamID.get(producer).get(j))){
                consumerSubs.get(i).add(streamID.get(producer).get(j));
            }
        }
    }
```

This part is the subscription process on the broker’s end. Whenever a new consumer asks for a subscription, he is remembered, and the corresponding stream is added to the list of his subscriptions.

All the code in this part consists of going through lists and adding the required information to them.



```
                case PacketContent.REGISTER:    // implementation for different streams from the
same producer
                    producerAddresses.add(
                            (InetSocketAddress) packet.getSocketAddress()
                    );
                    System.out.println("Current number of producers: " +
producerAddresses.size());

                    // translate the streamIDs to byte arrays and add them to the streamID array
                    // works for any number of streams
                    int a = producerAddresses.size() - 1;
                    // get the number of streams from the second byte of the packet
                    int numberOfStreams = data[1];
                    for (int j = 0; j < numberOfStreams; j++) {
                        // System.out.println("Adding a stream");
                        ArrayList<byte[]> temp = new ArrayList<byte[]>();
                        // temp.add(Arrays.copyOfRange(data, 5 + j * 4, 5 + j * 4 + 4));
                        streamID.add(temp);

                        byte [] byteArr = new byte[4];
                        byteArr[0] = data[j * 4 + 5];
                        byteArr[1] = data[j * 4 + 6];
                        byteArr[2] = data[j * 4 + 7];
                        byteArr[3] = data[j * 4 + 8];
                        streamID.get(a).add(byteArr);
                    }
```

This code is for the registration of the new producer and his streams. Both the producer and streams are added to the corresponding ArrayLists. The number of streams is in the second byte of the header. The ID of the producer is at the beginning of the stream ID, so there is no need to remember it. The streams themselves are stored after the header, so any number of streams can be added as long as it fits the packet size.

The Second problem was implementing a universal header for all three types. The first 5 bytes of each packet were assigned as the header and used further on the receipt end. Each byte function is the following:

1. The first byte is used to save the type of packet being sent. There are several types of packets, for example “subscription” or “file send”.
2. Bytes 2-4 are allocated for the producer ID.
3. The first 2 of those 3 bytes are currently in the form of ABCD, but can be changed to whatever is needed.
4. Byte 4 is used as the position of the producer for easier allocation and search in the future.
5. The last byte is the number of the stream that the corresponding producer is issuing.

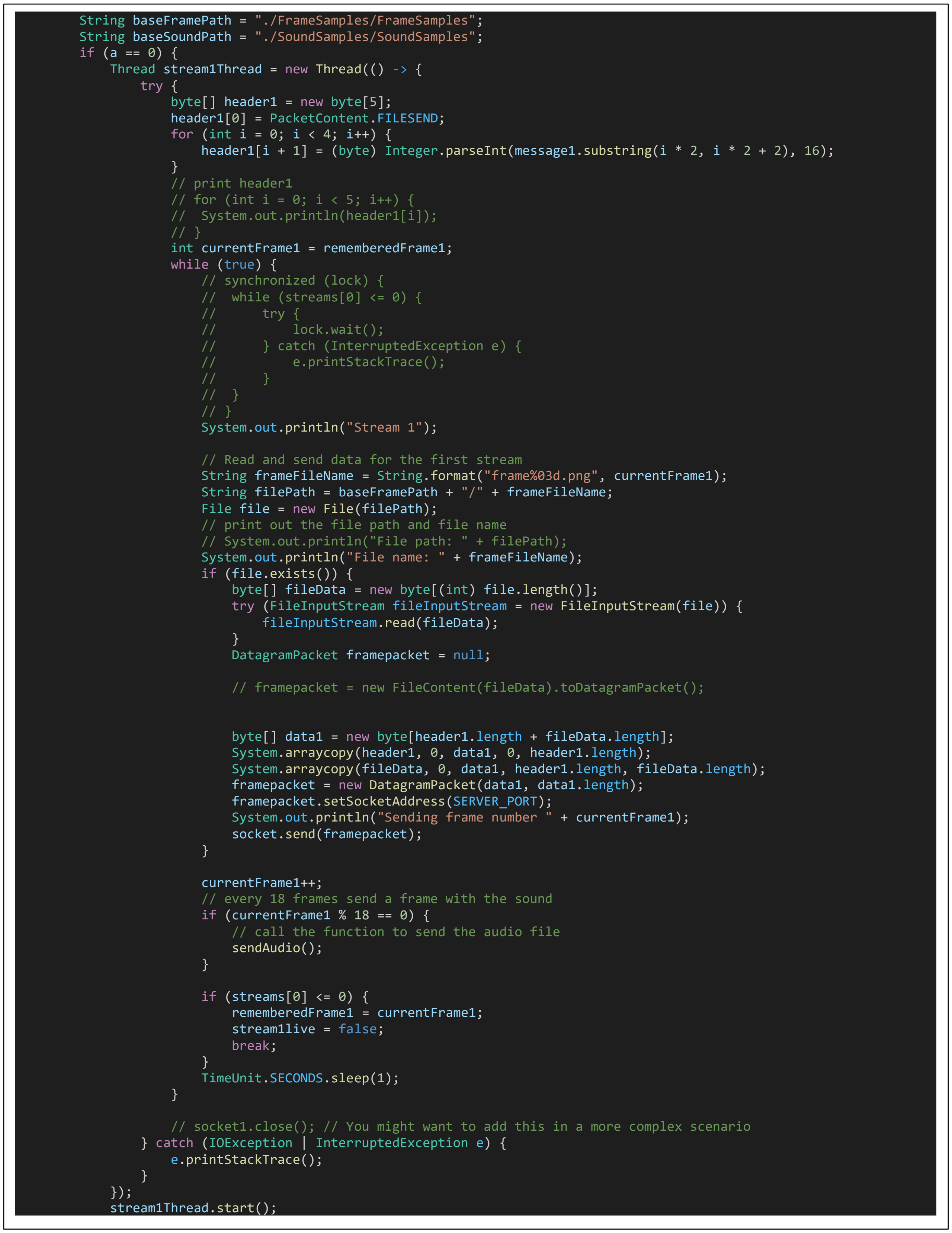
Code:

```
        // put all the streams and their IDs in the packet
        // stream IDs are in the form of ABCDxxyy where the xx is the
producers ID and the yy is the stream number/ID
        // for example ABCD0101 is the first stream of the first producer
        // ABCD0201 is the first stream of the second producer
        // ABCD0102 is the second stream of the first producer
        byte[] header = new byte[5];
        header[0] = PacketContent.REGISTER;
        byte numberOfStreams = 2;
        header[1] = numberOfStreams;
        // the rest bytes are all 0
        for (int i = 2; i < 5; i++) {
            header[i] = 0;
        }

        // Create a byte array to hold the header and the message
        byte[] data = new byte[5 + 8];

        // Copy the header to the beginning of the array
        System.arraycopy(header, 0, data, 0, 5);

        // Copy the encoded string to the array after the header
        for (int i = 0; i < 8; i++) {
            if (i < 4) {
                data[i + 5] = (byte) Integer.parseInt(message1.substring(i *
2, i * 2 + 2), 16);
            } else {
                data[i + 5] = (byte) Integer.parseInt(message2.substring((i -
4) * 2, (i - 4) * 2 + 2), 16);
            }
        }
```

This code is from the start() method of the producer. message1 and message2 are the IDs of both streams from that producer. They are parsed into integer and added to the data[] array, which is later put into the packet. A simillar process happens on the side of the consumer when a subscription or unsubscription is required.

The third problem was to properly encode and send the actual required data from the producer. This was done by simply taking the file from the folder where all the files are sorted in order. Streams themselves are located inside the threads. Threads for several streams differ only in the type of file they are sending, for example .png, which can be quickly changed for any other type of file.

The code is on the following page.



```
        String baseFramePath = "./FrameSamples/FrameSamples";
        String baseSoundPath = "./SoundSamples/SoundSamples";
        if (a == 0) {
            Thread stream1Thread = new Thread(() -> {
                try {
                    byte[] header1 = new byte[5];
                    header1[0] = PacketContent.FILESEND;
                    for (int i = 0; i < 4; i++) {
                        header1[i + 1] = (byte) Integer.parseInt(message1.substring(i * 2, i * 2 + 2), 16);
                    }
                    // print header1
                    // for (int i = 0; i < 5; i++) {
                    //  System.out.println(header1[i]);
                    // }
                    int currentFrame1 = rememberedFrame1;
                    while (true) {
                        // synchronized (lock) {
                        //  while (streams[0] <= 0) {
                        //      try {
                        //          lock.wait();
                        //      } catch (InterruptedException e) {
                        //          e.printStackTrace();
                        //      }
                        //  }
                        // }
                        System.out.println("Stream 1");

                        // Read and send data for the first stream
                        String frameFileName = String.format("frame%03d.png", currentFrame1);
                        String filePath = baseFramePath + "/" + frameFileName;
                        File file = new File(filePath);
                        // print out the file path and file name
                        // System.out.println("File path: " + filePath);
                        System.out.println("File name: " + frameFileName);
                        if (file.exists()) {
                            byte[] fileData = new byte[(int) file.length()];
                            try (FileInputStream fileInputStream = new FileInputStream(file)) {
                                fileInputStream.read(fileData);
                            }
                            DatagramPacket framepacket = null;

                            // framepacket = new FileContent(fileData).toDatagramPacket();


                            byte[] data1 = new byte[header1.length + fileData.length];
                            System.arraycopy(header1, 0, data1, 0, header1.length);
                            System.arraycopy(fileData, 0, data1, header1.length, fileData.length);
                            framepacket = new DatagramPacket(data1, data1.length);
                            framepacket.setSocketAddress(SERVER_PORT);
                            System.out.println("Sending frame number " + currentFrame1);
                            socket.send(framepacket);
                        }

                        currentFrame1++;
                        // every 18 frames send a frame with the sound
                        if (currentFrame1 % 18 == 0) {
                            // call the function to send the audio file
                            sendAudio();
                        }

                        if (streams[0] <= 0) {
                            rememberedFrame1 = currentFrame1;
                            stream1live = false;
                            break;
                        }
                        TimeUnit.SECONDS.sleep(1);
                    }

                    // socket1.close(); // You might want to add this in a more complex scenario
                } catch (IOException | InterruptedException e) {
                    e.printStackTrace();
                }
            });
            stream1Thread.start();
```

The explanation is on the next page.

In this picture is one of the threads of the producer. This thread is created after the number of subscribers to the corresponding stream reaches a number above 0. It takes the file from the source, which is stated before the thread, and translates it into a byte[] array. This array is combined with the header array and put into the packet. The sendAudio() method is called every 18 frames to send the matching audio file. That method is almost the same as the thread a bove.

The rest of the design is packet encoding using Java libraries for sockets and datagram packets. A big part of the code and inspiration weres taken from the Java example from the Blackboard.

5 Discussion

Overall, the solution provided above covers everything required by the assignment.

The main disadvantage of it is the usage of ArrayLists. Because of them, every time a broker requires access to subscriptions, IDs, or streams, it will have to go through the whole ArrayList in order to find the correct item. It does the job on a smaller scale, but as the number of consumers, producers, and streams grows, so will ArrayLists. That might lead to heavily increased time consumption. One of the possible options to fix this issue is to use a hash map.

Another small disadvantage might be the difficulty of adapting the producer to new requirements. It covers everything needed, but if some other functionality might be wanted, for example, another stream, it will require a several changes.

The main advantage of the stated solution is its simplicity and potential for additional functionality. The code structure is basic, allowing for the incorporation of extra features if needed.

The given solution works for multiple producers and consumers. New consumers can be started using the same dockerfile and code. As stated before, for new producers, it will require a few changes, for example, another stream name/ID or the type of file.

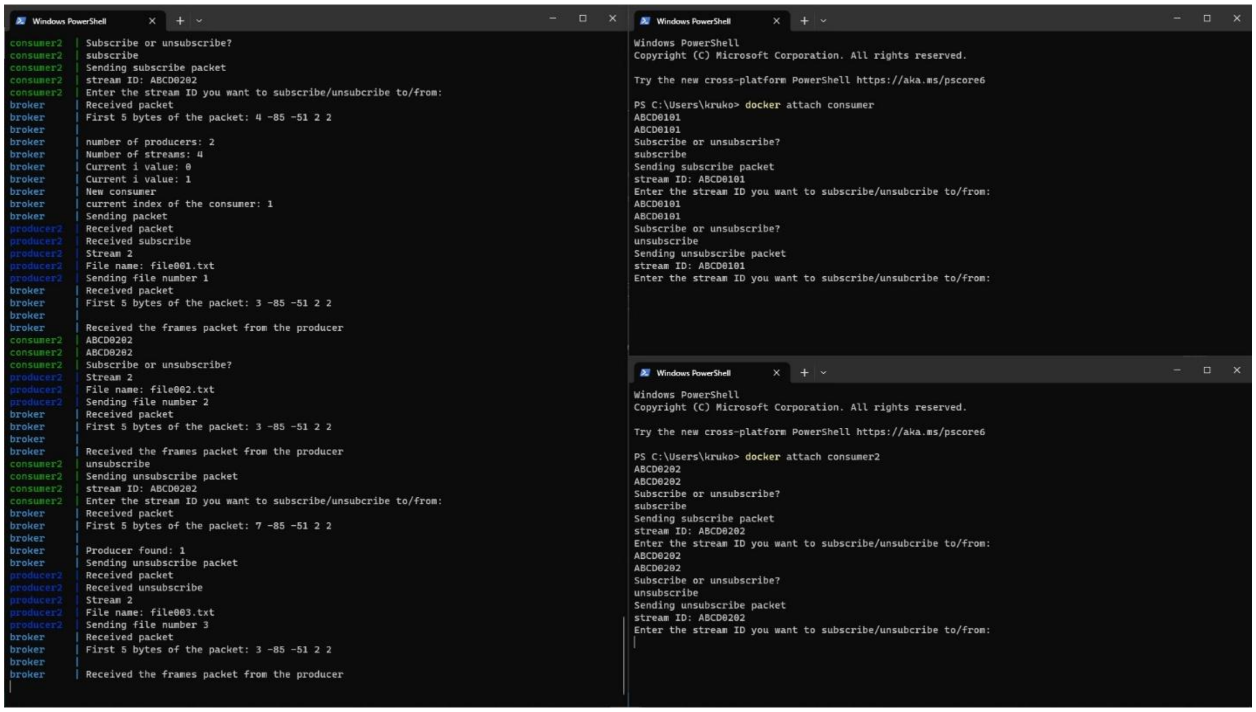
For the sake of testing the project, everything was in one network. This was done in order to have only 1 file with captured traffic instead of 2 or more. This can be changed if needed.

6 Summary

This project covers the most basic functionality of live streaming protocols, making sure that the video can be sent to anyone who is subscribed to the stream. Every packet goes through the broker, which also handles everything related to the subscription and communication between consumers and producers. In this report you can find all crucial information needed to replicate the project including screenshots of the most important parts of the code and an explanation of them.

7 Reflection

This assignment took significantly more time than I thought it would. Lack of any previous knowledge on the matter and the amount of work that it required slowed the progress several times. Watching a few introduction videos online and doing some research helped a lot at the start. If I had more time, I would try to change the implementation of subscriptions and stream IDs storage system. The ArrayList variant works and does the job, but it can be done better. Next time, I would try to start a week or two earlier, so I would have enough time to make some major changes to the code if needed. A lot of time was also wasted on running the containers. This might be due to the incorrect settings or just the power of the machine I was testing everything on.



```
consumer2  | Subscribe or unsubscribe?
consumer2  | subscribe
consumer2  | Sending subscribe packet
consumer2  | stream ID: ABCD0202
consumer2  | Enter the stream ID you want to subscribe/unsubcribe to/from:
broker     | Received packet
broker     | First 5 bytes of the packet: 4 -85 -51 2 2
broker     |
broker     | number of producers: 2
broker     | Number of streams: 4
broker     | Current i value: 0
broker     | Current i value: 1
broker     | New consumer
broker     | current index of the consumer: 1
broker     | Sending packet
producer2  | Received packet
producer2  | Received subscribe
producer2  | Stream 2
producer2  | File name: file001.txt
producer2  | Sending file number 1
broker     | Received packet
broker     | First 5 bytes of the packet: 3 -85 -51 2 2
broker     |
broker     | Received the frames packet from the producer
consumer2  | ABCD0202
consumer2  | ABCD0202
consumer2  | Subscribe or unsubscribe?
producer2  | Stream 2
producer2  | File name: file002.txt
producer2  | Sending file number 2
broker     | Received packet
broker     | First 5 bytes of the packet: 3 -85 -51 2 2
broker     |
broker     | Received the frames packet from the producer
consumer2  | unsubscribe
consumer2  | Sending unsubscribe packet
consumer2  | stream ID: ABCD0202
consumer2  | Enter the stream ID you want to subscribe/unsubcribe to/from:
broker     | Received packet
broker     | First 5 bytes of the packet: 7 -85 -51 2 2
broker     |
broker     | Producer found: 1
broker     | Sending unsubscribe packet
producer2  | Received packet
producer2  | Received unsubscribe
producer2  | Stream 2
producer2  | File name: file003.txt
producer2  | Sending file number 3
broker     | Received packet
broker     | First 5 bytes of the packet: 3 -85 -51 2 2
broker     |
broker     | Received the frames packet from the producer
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\kruko> docker attach consumer
ABCD0101
ABCD0101
Subscribe or unsubscribe?
subscribe
Sending subscribe packet
stream ID: ABCD0101
Enter the stream ID you want to subscribe/unsubcribe to/from:
ABCD0101
ABCD0101
Subscribe or unsubscribe?
unsubscribe
Sending unsubscribe packet
stream ID: ABCD0101
Enter the stream ID you want to subscribe/unsubcribe to/from:
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\kruko> docker attach consumer2
ABCD0202
ABCD0202
Subscribe or unsubscribe?
subscribe
Sending subscribe packet
stream ID: ABCD0202
Enter the stream ID you want to subscribe/unsubcribe to/from:
ABCD0202
ABCD0202
Subscribe or unsubscribe?
unsubscribe
Sending unsubscribe packet
stream ID: ABCD0202
Enter the stream ID you want to subscribe/unsubcribe to/from:
```

The setup that was used for testing. A broker on the left side of the screenshot, and two consumers on the right.