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

CSU33031 Computer Networks Assignment #1: Protocols

Ilia Romanov, Std# 23356952

November 1, 2023

Contents

1 Introduction

This report describes my solution to the task of developing a protocol for supporting a mechanism for distributing video content for live-streaming. More specifically, the protocol is meant to enable a pub/sub type system and is to be built on top of the User Datagram Protocol (UDP). In this report I will begin by briefly discussing the course concepts relevant to my solution and the technologies I used for implementing my solution, followed by the implementation details of my solution such as the overall topology, header design, and details of specific components of my solution. Lastly, I will end the report with a discussion of the pros and cons of my solution and a reflection on my experience working on this assignment.

2 Technical Background

In this section I will discuss course concepts relevant to my solution and the technologies I used for implementing my solution.

2.1 User Datagram Protocol (UDP)

As discussed in lecture, UDP is a protocol which allows for sending and receiving packets over IP. UDP protocol already contains a header which is 8 bytes long. Hence, for my custom protocol I will be adding my own header in the body of UDP datagrams/packets.

2.2 Automatic Repeat Request (ARQ)

ARQ is a networking flow control scheme which has several different implementation such as Stop-and-Wait, Selective Repeat, and Go-Back-N. The concepts at the core of ARQ are ACKs - acknowledgement messages that are sent to confirm whether some frame was received or not - and timeouts - configured limits on the amount of time a sender will wait on a response ACK until it gives up and performes some alternative action. For more details on my adoption of these concepts in my solution, view the Flow and and ARQ Adaptation section in below.

2.3 Docker

Running my solution is made possible using Docker - a software that enables creation and execution of isolated containers communicating over network bridges. Each actor that will be seen in my overall topology in the implementation section below runs in its own isolated Docker container. All of the containers in my system run and communicate on the ”csnet” network bridge created through the steps shown in the Docker tutorial provided at the beginning of the term. The instructions for building the Docker image which all of the actor containers in my system are based on can be found in `Dockerfile`, and a shell script for creating a container for a new actor in my system, specifically a new producer or consumer, is titled `create_container.sh`. Both can be found in my attached code.

2.4 Python

The logic for the actors in my system and the communication actions between them are implemented using Python3. Python3 has a built-in module - `socket` - for creating UDP sockets and transporting packets over them. I wrote a module/class on top of the built-in `socket` module implementing the base functionality required for my custom protocol such as sending, receiving, packing, and unpacking packets and corresponding headers in my custom protocol, as well as setting timeouts on the UDP sockets which are used for my ARQ implementation. This class can be found in `ProtocolSocketBase.py` in my attached code.

3 Implementation

In this section I will discuss the specifics of the implementation of my solution. I will begin with the overall topology of my design, followed by the contents of my protocol's header, and my adaptation of ARQ flow concepts in my system. I will finish with a section for each of the actors in my solution - producer, consumer, and broker - with a quick overview of the logic behind each.

3.1 Overall Topology

In my solution there are three types of actors: broker, producer, and consumer. Each actor runs in its own separate Docker container.

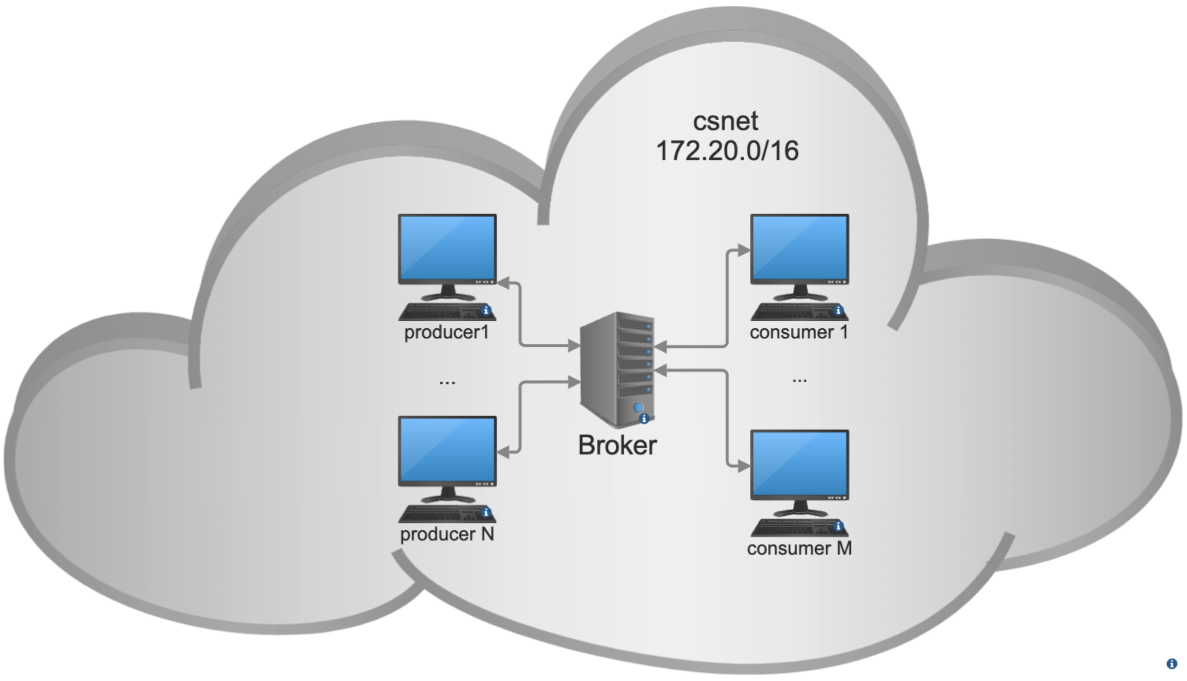


Figure 1: Diagram of my network topology

Producers announce new streams and publish frames/text for their streams by talking to a broker, consumers send subscription requests for either all streams or a single specific stream by any producer by talking to the broker, and the broker distributes published content to corresponding subscribed consumers. The broker also sends ACK replies in response to receiving certain producer and consumer packets as part of my ARQ flow adaptation which is why the communication arrows in Figure 1 go both ways between producer and broker as well as consumer and broker.

Lastly, note that all the actors my solution are connected on the Docker ”csnet” network bridge introduced in the Docker tutorial. This network allows for $2^{(32-16)} = 2^{16} = 65536$ total IP addresses. One IP address is required for the broker and the rest are available to be used by producer and consumer nodes. Hence, my solution allows for up to 65535 total active producers and consumers.

3.2 Header Design

In this section I will describe the fields and anatomy of my header design for my custom protocol. I will use screenshots of capture files being viewed through Wireshark as examples for use cases of the header fields.

Overall Anatomy:
My header consists of 13 bytes in total, broken down as shown in the example header in Figure 2 below:

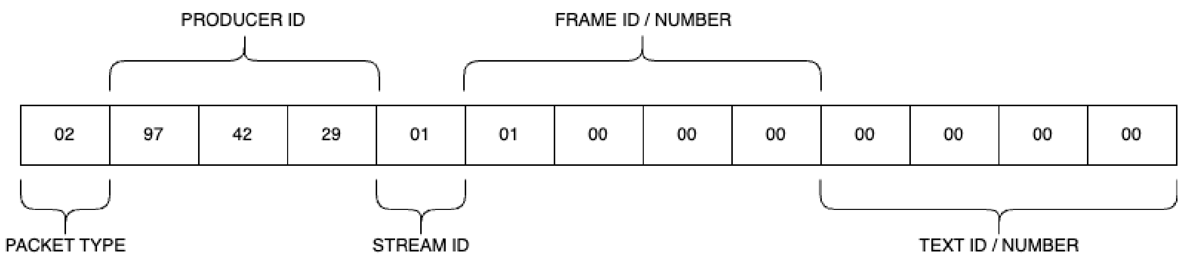


Figure 2: Example header for my protocol

As can be seen in Figure 2, the first byte of the header stores the Packet Type, the next three bytes are the Producer ID, the fifth byte is the Stream ID, the next four store the Frame ID and the final four store the Text ID.

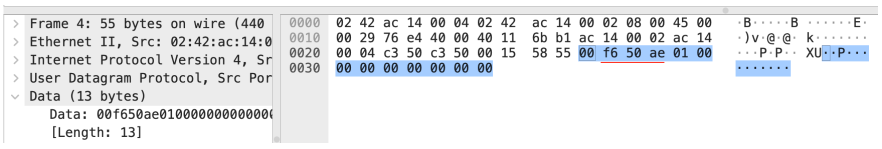
Field Details:

Packet Type: Needed within the code for each actor in my solution to determine how a received packet should be handled. The packet types I defined for my system are:

```
- ANNOUNCE_STREAM     = 0
# For when a producer sends a packet to the broker to announce a new stream

- ANNOUNCE_STREAM_ACK = 1
# For when a broker sends an ack that it registered the stream announcement by a producer

- SEND_FRAME          = 2
# For packets containing frame data in their body.

- SEND_TEXT           = 3
# For packets containing text data in their body.

- SUB_STREAM          = 4
# For when a consumer sends the broker a sub request for a single specific stream

- SUB_STREAM_ACK      = 5
# For when a broker sends an ack that it registered the sub request from a consumer

- UNSUB_STREAM        = 6
# For when a consumer sends the broker an unsub request to a single specific stream

- UNSUB_STREAM_ACK    = 7
# For when a broker sends an ack that it performed the unsub request from a consumer

- SUB_PRODUCER        = 8
# For when a consumer sends the broker a sub request for all streams belonging to a producer

- SUB_PRODUCER_ACK    = 9
# For when a broker sends an ack that it performed a consumer's sub request to a producer

- UNSUB_PRODUCER      = 10
# For when a consumer sends the broker an unsub request for all streams belonging to a
  producer

- UNSUB_PRODUCER_ACK  = 11
# For when a broker sends an ack that it performed a consumer's unsub request to a producer
```

Packet Type is a required field for any packet sent using my protocol so it can be seen in the Wireshark captures in the examples for the fields below.

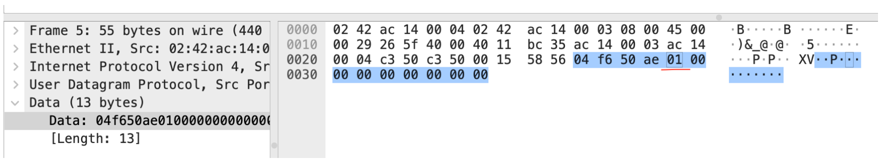
Producer ID: Needed to identify producers. For example when a producer sends a packet announcing a new stream to the broker (shown in Figure 3), or a consumer sends a packet requesting to subscribe to a stream of a specific producer this field would be required.



```
> Frame 4: 55 bytes on wire (440         0000  02 42 ac 14 00 04 02 42  ac 14 00 02 08 00 45 00   ·B·····B ······E·
> Ethernet II, Src: 02:42:ac:14:0        0010  00 29 76 e4 40 00 40 11  6b b1 ac 14 00 02 ac 14   ·)v·@·@· k·······
> Internet Protocol Version 4, Sr        0020  00 04 c3 50 c3 50 00 15  58 55 00 f6 50 ae 01 00   ···P·P·· XU··P···
> User Datagram Protocol, Src Por        0030  00 00 00 00 00 00 00                             ·······
v Data (13 bytes)
    Data: 00f650ae010000000000000
    [Length: 13]
```

Figure 3: Packet sent by a producer to the broker. Note the hex for the 3 byte producer ID - f6 50 ae. Also note that the packet type (first byte to the left of the producer ID bytes) is 00 which denotes announce stream.

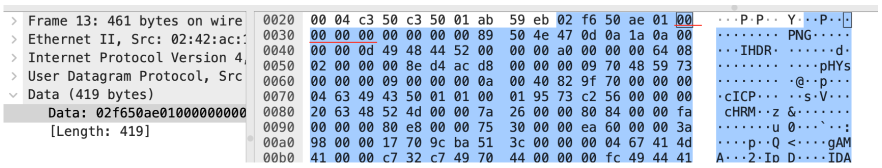
Stream ID: Needed to identify stream. For example when a consumer sends the broker a subscribe request to a specific stream (shown in Figure 4), or a producer publishes a frame of a stream, this field needs to be specified.



```
> Frame 5: 55 bytes on wire (440         0000  02 42 ac 14 00 04 02 42  ac 14 00 03 08 00 45 00   ·B·····B ······E·
> Ethernet II, Src: 02:42:ac:14:0        0010  00 29 26 5f 40 00 40 11  bc 35 ac 14 00 03 ac 14   ·)&_@·@· ·5······
> Internet Protocol Version 4, Sr        0020  00 04 c3 50 c3 50 00 15  58 56 04 f6 50 ae 01 00   ···P·P·· XV··P···
> User Datagram Protocol, Src Por        0030  00 00 00 00 00 00 00                             ·······
v Data (13 bytes)
    Data: 04f650ae010000000000000
    [Length: 13]
```

Figure 4: Packet sent by a consumer to the broker to request a subscription to stream id 01 from producer with hex form of ID being f6 50 ae.

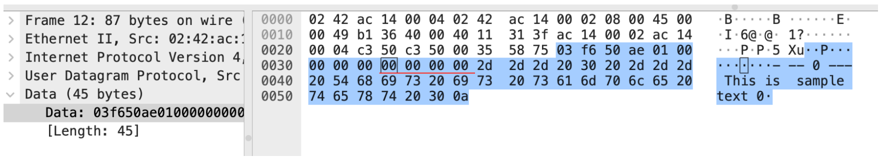
Frame ID: Needed to identify the count of the frame being sent (is it the first frame, second frame, etc). For example when a broker sends frame of a stream to a subscribed consumer (shown in Figure 5), or a producer publishes a new frame for a stream this field needs to be specified.



```
> Frame 13: 461 bytes on wire   0020  00 04 c3 50 c3 50 01 ab  59 eb 02 f6 50 ae 01 00   ···P·P·· Y···P···
> Ethernet II, Src: 02:42:ac:1  0030  00 00 00 00 00 00 00 89  50 4e 47 0d 0a 1a 0a 00   ········ PNG·····
> Internet Protocol Version 4,  0040  00 00 0d 49 48 44 52 00  00 00 a0 00 00 00 64 08   ···IHDR· ······d·
> User Datagram Protocol, Src   0050  02 00 00 00 8e d4 ac d8  00 00 00 09 70 48 59 73   ········ ····pHYs
v Data (419 bytes)              0060  00 00 00 09 00 00 00 0a  00 40 82 9f 70 00 00 00   ········ ·@··p···
    Data: 02f650ae01000000000   0070  04 63 49 43 50 01 01 00  01 95 73 c2 56 00 00 00   ·cICP··· ··s·V···
    [Length: 419]               0080  20 63 48 52 4d 00 00 7a  26 00 00 80 84 00 00 fa    cHRM··z &·······
                                0090  00 00 00 80 e8 00 00 75  30 00 00 ea 60 00 00 3a   ·······u 0···`··:
                                00a0  98 00 00 17 70 9c ba 51  3c 00 00 00 04 67 41 4d   ····p··Q <····gAM
                                00b0  41 00 00 c7 32 c7 49 70  44 00 00 00 fc 49 44 41   A···2·Ip D····IDA
```

Figure 5: Packet sent by the broker to a consumer to froward a frame 0 for stream 1 from producer with hex form of producer id being f6 50 ae.

Text ID: Needed to identify count of the text being sent. For example when a producer sends a piece of text for one of its streams (shown in Figure 6), or a broker sends a subscribed consumer a piece of published text this field would be used.



```
> Frame 12: 87 bytes on wire (
> Ethernet II, Src: 02:42:ac:1
> Internet Protocol Version 4,
> User Datagram Protocol, Src
v Data (45 bytes)
    Data: 03f650ae01000000000
    [Length: 45]
```

```
0000  02 42 ac 14 00 04 02 42  ac 14 00 02 08 00 45 00   ·B····B ······E·
0010  00 49 b1 36 40 00 40 11  31 3f ac 14 00 02 ac 14   ·I·6@·@· 1?······
0020  00 04 c3 50 c3 50 00 35  58 75 03 f6 50 ae 01 00   ···P·P·5 Xu··P···
0030  00 00 00 00 00 00 00 2d  2d 2d 20 30 20 2d 2d 2d   ·······- -- 0 ---
0040  20 54 68 69 73 20 69 73  20 73 61 6d 70 6c 65 20    This is  sample 
0050  74 65 78 74 20 30 0a                               text 0·
```

Figure 6: Packet sent by a producer to the broker to stream a piece of text number 0 for stream 1.

Note that Frame ID and Text ID are needed and used in my solution to have a way of ignoring duplicate frames/text packets or frames/text packets that go back in time form what has already been processed by a consumer. The reason for requiring both of these fields rather than just one of them is for when the broker needs to notify a newly subscribed consumer of both the highest Frame ID and the highest Text ID that has been seen for its subscribed to stream (more on this in the broker subsection below).

3.3 Flow Control and ARQ Adaptation

A generic flow in my solution can be seen in Figure 7. The producer "ABCD99" announces a new stream with Stream ID 01, receives an ACK back from the broker. Then the consumer sends the broker a subscription request for Stream 01 published by Producer "ABCD99" and receives back an ACK from the broker. The producer then publishes frames to the broker and the broker forwards them to the subscribed consumer, ignoring frames that are duplicates or that precede the highest seen frame by a consumer for the given stream.

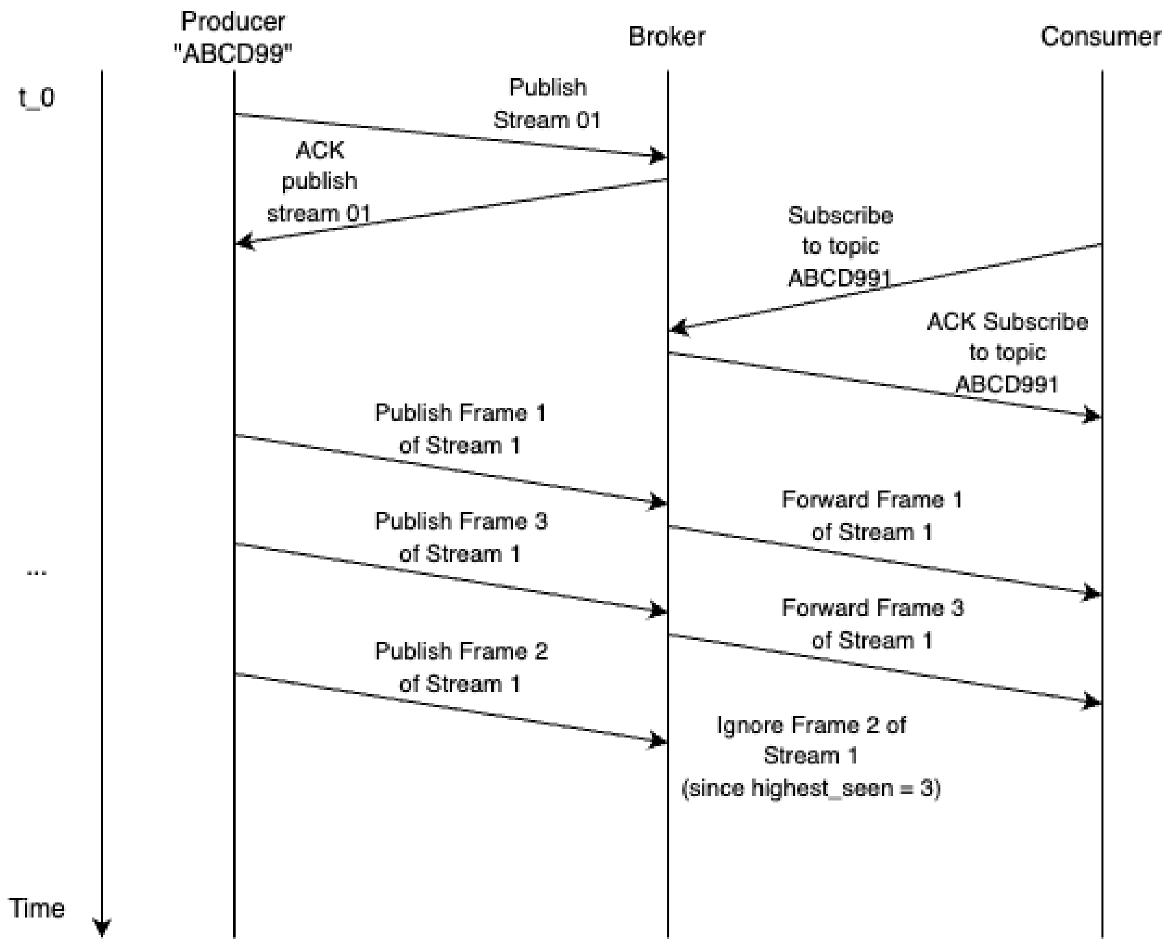

Producer "ABCD99"
Broker
Consumer
t_0
Publish Stream 01
ACK publish stream 01
Subscribe to topic ABCD991
ACK Subscribe to topic ABCD991
Publish Frame 1 of Stream 1
Forward Frame 1 of Stream 1
Publish Frame 3 of Stream 1
Forward Frame 3 of Stream 1
...
Publish Frame 2 of Stream 1
Ignore Frame 2 of Stream 1 (since highest_seen = 3)
Time

Figure 7: Generic flow in my solution.

The usage of ACK responses as seen in Figure 7 above is the aspect of the ARQ flow that I decided to adopt into my solution. I use these ACKs to ensure that important actions within my solution such as announcing a new stream or performing a subscription/unsubscription are more likely to be processed in case of an unprecedented malfunction in communication. Furthermore, observe in Figure 8 how I use the ARQ flow concept of timeouts and for re-sending packets that request important events to occur in the case of some such communication malfunction.

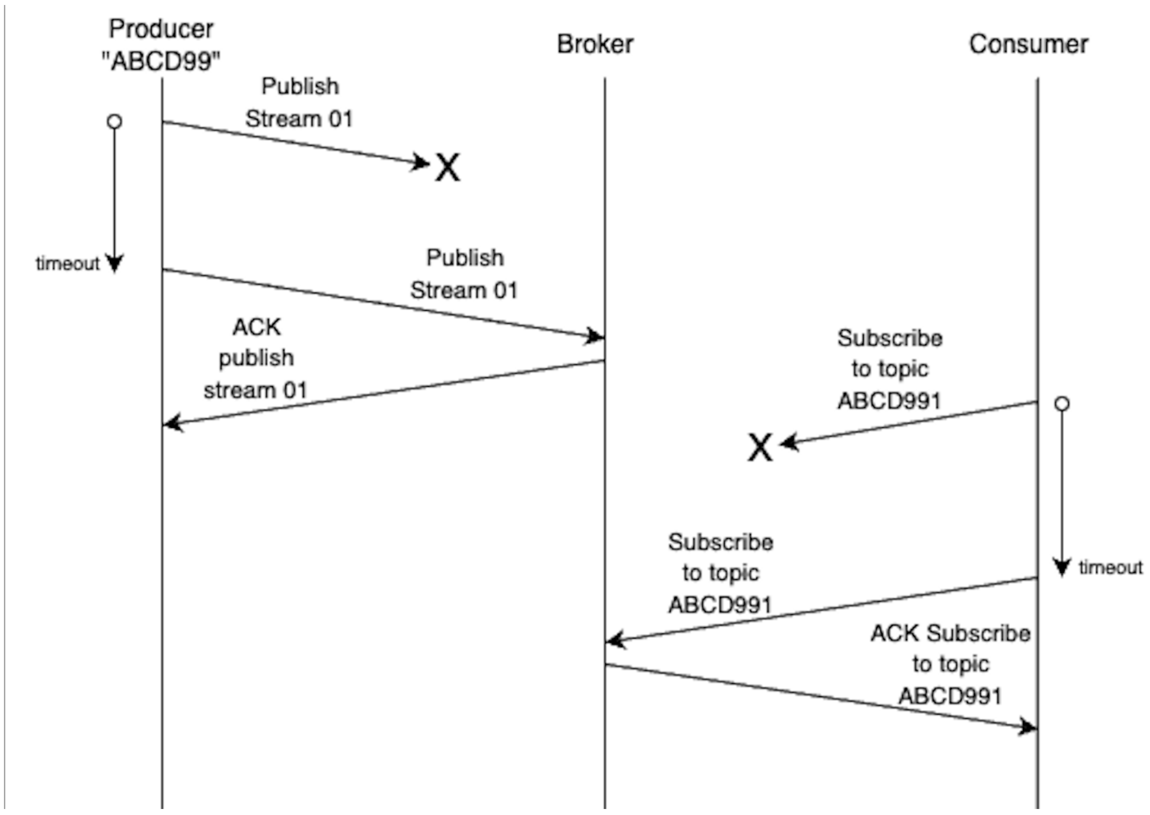


Figure 8: Flow with communication malfunction in my solution. Note how once a timeout is reached, the stream publish request is sent again and same idea for the subscribe request.

The number of allowed retries and timeout length can be tweaked as an input for my solution.

Lastly, note that while my diagrams above show the flow for only a single producer and consumer, my solution also handles multiple consumers and producers which operate with this same idea for the flow; each producer and consumer would individually communicate with the single broker as shown in the diagrams above.

3.4 Producer

The two files implementing the producers in my attached code are `producer_main.py` which implements the command line interface for it and `Producer.py` which has the main logic for producer actions. The functionality allowed through my CLI interface which uses an instance of the producer class includes: the publishing of streams and the streaming of either image frames or text frames for a given announced stream.

In my implementation, every producer instance has a running counter for the number of the highest image frame produced and the highest text frame produced which is initialized to zero for every new announced stream. These values are what is being put in the Frame ID and Text ID fields of my protocols header when the producer sends a packet for streaming some data.

Lastly, my producer implementation allows for multiple producers because the broker IP in my solution stays constant and each producer simply needs to communicate with the single known broker container which then talks to consumers as required and direct communication between producers and consumer is thus not needed. Once a producer sends the broker a packet that requires an ACK, the broker knows the producers IP through my ProtocolSocketBase class implementation from which all the actors in my system inherit, including producers and brokers.

3.5 Consumer

The two files implementing the consumers in my attached code are `consumer_main.py` which implements the command line interface for it and `Consumer.py` which has the main logic for consumer actions. Similar to the broker, the functionality allowed through my consumer CLI interface which uses an instance of the consumer class includes: subscribing to a specific stream or to all streams from a producer, and unsubscribing to either option as well.

I use multithreading to allow for listening of incoming packets to the consumer on a separate port; a separate thread runs on my consumer for receiving UDP packets on a specific port dedicated for this purpose. The main thread is used for the consumer to take in user input and send sub/unsub packets and receive ACK responses for them on a different port.

My implementation of consumers uses hashmaps for keeping track of the highest seen frame ID and text ID for all of its subscribed to streams in order to avoid processing duplicates or frames that precede the highest seen frame for a given stream. This is done in case a similar mechanism fails within the broker.

Multiple consumers are possible for the same reasoning as described in the producer subsection above.

3.6 Broker

My implementation of the broker in my attached code are `broker_main.py` which implements the command line interface for it and `Broker.py` which implements the main logic for the broker. The broker does not take any user input commands as its only purpose is to facilitate communication between the producers and consumer in my system. I keeps hashmaps for tracking the current subscriptions of consumers and for tracking the highest processed frame and text counts for each announced topic by producers. It uses these hashmaps for forwarding produced content to subscribed consumers as well as ensuring no duplicates or back in time frames/text.

4 Discussion

I believe that my solution has a lot of good strengths. For example important actions such as announcing a new stream, and subscription/unsubscription requests, have ensured stability using ARQ concepts of ACKs and timeouts before re-sending of packets, while sending frames/text does not require ACKs for efficiency reasons. My solutions also ensure that no duplicate frames are shown and that frames are not presented out of order. The last big strength of my solution, is that it also allows for multiple types of data to be transmitted over the protocol, namely video in the form of frames, and text.
I'm proud also of the extendability of my code since I wrote a separate base class for my protocol which can be modified to enable all the actors inheriting from it to have more functionality.
A big area where my solution may lack is efficiency in highly concurrent or simply very large task loads. Since my broker only runs a single thread for all of its requests, I think it would possible experience some heavy lag in a case where there are tens of thousands of producers and consumers sending it requests at once.

5 Reflection

Overall, I really enjoyed working on this assignment and I feel that I know have a deeper understanding of concepts such as UDP and ARQ networking flow as well as how to design my own header for specific protocol requirements. I would say I spent about 60 to 80 hours of active work on this project and most of it went to trying to make my code as readable and extendable as possible. Thus, if I were to do something different for this assignment, I would focus more on the specifics of the task at hand rather than trying to solver larger scale ideas out of the scope of the requirements of the assignment.