CSU33031 Computer Networks

Assignment #1: Publish/Subscribe Protocol for Video, Audio and Text

Rhys Mac Giollabhuidhe

October 30, 2023

Contents

1 Introduction

The aim of this assignment is to look at protocol design; namely the design of headers, and the balancing of minimising header size while adding functionality that requires more information to be contained in the header. The particular context we were given to consider was providing a new protocol to be used by a content streaming service.

This report will lay out the specific design goals and constraints of the problem at hand, as well as my own background in this area of development and the technologies used in the development process to provide context for my process and the solution I ultimately arrived on. I will then discuss my initial approach to designing a solution, before going more in depth on the final implementation. The motivation for this being that an understanding of the design intentions that led to the final build will help to understand the particular functionality and merits thereof.

2 Problem Statement

This section will briefly lay out any design specifications of the project that were considered during the design and development process to give a better understanding of the reasoning behind the decisions I made throughout both.

2.1 Topology

The assignment descriptor specified a topology whereby a number of producers send content to a single broker. The function of the broker is to track incoming content and send that content on to a number of consumers that can request particular content streams from any producers currently running. Data sent between these various actors is to be done using User Datagram Protocol sockets.

2.2 Specifications of Header Design

The overhead for packets must be able to support the publication of content by a number of producers simultaneously, and must be able to identify particular content streams and the types of content in those streams. The same is also true for requests by consumers to place subscriptions for particular streams or types of content by a given producer.

Furthermore, producers must be identified by a unique 3 byte ID. Content streams must be identified by a 1 byte ID, as well as the ID of the producer they belong to.

3 Background

This section will detail my own understanding of the principles of protocol design prior to the beginning of the design and development of the assignment. I will also briefly outline the development tools and environments I ultimately decided on using, and how these tools impacted the development process to provide context for particular design and implementation decisions that will be explored in later sections.

3.1 Technical Background

I have never worked in the field of protocol design before, be it for a personal project or a college assignment. However, I felt the lectures on the course content left me in a good starting point from which to approach the problem at hand; though I'd never designed such a protocol before, I felt I understood what my design should aim to achieve.

The above, coupled with our freedom to choose a development environment I was already comfortable with meant that though I was inexperienced in the field, I was ultimately confident in my ability to learn from the process and deliver on what was required of me.

3.2 Technologies Used

I decided early on that I would develop the project in the Python IDE Pycharm, for several reasons. Chiefly, Python is a language I am very comfortable with that lends itself well to quick development. Considering that I would have to learn much about protocol design throughout the process, I reasoned that Python's many useful libraries for easy handling of video and audio files would allow me to focus more on protocol design, which was the express learning goal of this assignment.

I have never done work before with a simulated network, so I discussed potential options with some classmates who pointed me in the direction of Docker. I found Docker intuitive to use and suitable for my purposes, and so I decided to use it to simulate the network.

4 Design

This section will discuss my intentions for the high-level design of the completed project before beginning with the implementation thereof. The implementation section will make reference to the content discussed in this section, and understanding of the motivation behind particular design decisions will provide useful context for understanding particular details of the implementation.

4.1 Project Topology

The problem statement specified a topology wherein 1 to many producers stream content to a broker, which forwards that content to 1 to many consumers on request, or "subscription", by those consumers. Below is a diagram visualising this topology.

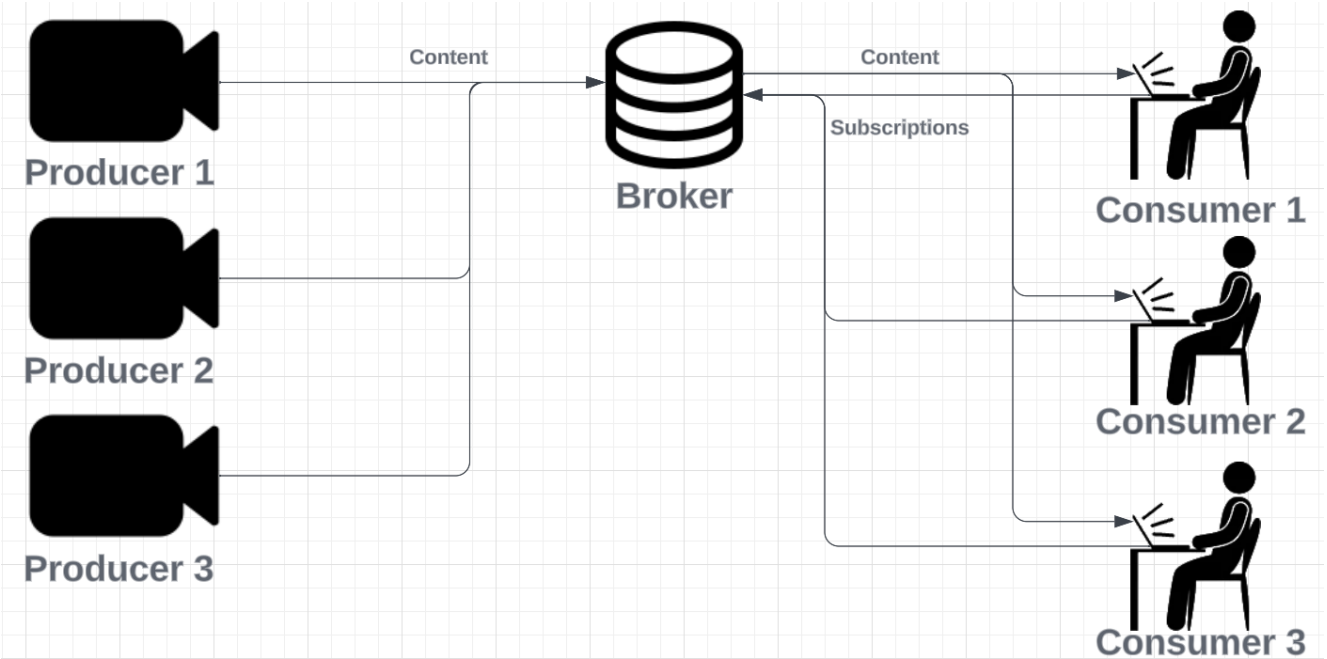

Fig 1: Specified topology of the project

As this is the topology laid out by the problem statement, the rest of the design process was done with the above in mind.

4.2 Overhead Design

The most basic functionality of the above described topology is the distribution of content packets by producers to a broker, and the forwarding of those packets by the broker to consumers. As such, the first aspect of header design I considered

was to facilitate the correct handling of such packets. Below are the details of my initial design for these purposes.

4.2.1 Producer Content Packet Overhead Design

I decided initially that my packet overhead would be 5 bytes in total. I have never worked with bit manipulation in Python before, but I quickly learned that bit manipulation in Python is quite limited, and so I reasoned that a 5 byte overhead would be the best I could do; per the specifications of the assignment, the ID of a producer had to be 3 bytes, and this would be necessary to include in the header. Of course, a stream's ID and type would also be necessary to include, and since bit manipulation is limited, I decided a separate byte would be necessary to indicate each. Hence, 5 bytes.

I decided that each producer's unique ID should start with the bits "1010". Since my design for the header of a stream data packet was such that it would start with the producer's ID, this way the broker would be able to identify an incoming stream data packet by looking at just the first 4 bits. From there, it could know how to handle the packet appropriately by examining the bits of the overhead allocated for indication of the stream's ID and content type.

Producer ID - bytes 1-3	**Stream ID - byte 4**	**Content Type - byte 5**
101000000000000000000001	11000110	11111111
0xA00001	C6	FF

Fig 2: Example of the overhead for a packet belonging to stream C6, or 198, by producer 1, consisting of audio content

4.2.2 Consumer Subscription Overhead Design

I decided that a consumer could indicate which stream it wanted to subscribe to in much the same way as a producer could indicate which stream a packet belonged to, as the same information is relevant to each.

The only difference being that in order for the broker to be able to differentiate between new content packets and subscription requests, a consumer's subscription request would change the ID of the desired producer to begin with the 4 bits "1001" instead of "1010", but otherwise remain the same.

Producer ID - bytes 1-3	**Stream ID - byte 4**	**Content Type - byte 5**
100100000000000000000010	01010001	10101010
0x900002	51	AA

Fig 3: Example of the overhead for a subscription packet for to stream 51, or 81, by producer 2, consisting of text content

4.3 Critical Packets

Though I had not initially planned for them, further development of the project necessitated that I consider certain packets to be “critical packets”, which is how I will refer to packets that must be received and handled properly in order for the system to function unimpeded. An example of such a packet would be a subscription request from a consumer to the broker.

I quickly realised that proper handling of critical packets would require confirmation from the recipient that the critical packet was received. As my original overhead design did not consider anything but stream content and subscription packets, I concluded that confirmation packets should be handled entirely differently to avoid a total redesign of the system I had already implemented. I was not entirely confident in this design, however given that such packets would only be handled highly contextually, I reasoned that it was at least functional.

With this in mind, the design I settled on would use a very small 1 byte overhead to indicate receipt of the critical packets. The byte in question would differ based on the packet being confirmed, to ensure that multiple confirmation packets being sent at once would not be confused. The below table indicates the 6 types of packets handled as critical packets by my implementation. These packets are each discussed further in the implementation section.

Context	**Sender->Recipient**
new producer notification	Producer->Broker
ID for new producer	Broker->Producer
request for list of running streams	Consumer->Broker
new subscription	Consumer->Broker
stream end	Producer->Broker
stream end	Broker->Consumer

Fig 4: Table of critical packets

5 Implementation

This section will describe the lower-level details of the functionality of the final solution, and how these details were arrived on from the previously discussed initial design.

Specifically, this section will touch on final header design, the functionality of the broker, producers and consumers, as well as giving an overview of the functionality of the system as a whole.

5.1 Implementation and Handling of Headers

The initial design for content packet and subscription headers proved effective throughout the development of the project and survived all the way from initial testing to the final build. Its robustness as well as its simplicity made the implementation of additional content types and multiple producers and consumers running/viewing multiple streams easy to implement once the basic framework was in place. Additionally, it being as minimal as it is reduces stress on the network.

However, as discussed in the design section, the handling of critical packets meant that I needed to adapt the system to also handle packets that communicate running information between actors, namely confirmation of the receipt of critical packets. I did ultimately stay with the single-byte overhead, however handling them introduced some difficulties that will be discussed further in the reflection

section. Below is some code depicting the basic skeleton of how I handled critical packets.

```
while True:
      local_socket.settimeout(1)
      local_socket.sendto(critical_packet, recipient_address_port)
      try:
            response = local_socket.recvfrom(buffer_size)
            header = response[0]
            if header == CRITICAL_PACKET_RECEIVED_CONFIRMATION:
                  print("critical packet was received!")
                  break
            else:
                  print("wrong packet. reattempting…")
      except socket.timeout:
            print("no response. reattempting…")
```

Fig 6: Basic skeleton of critical packet confirmation code, provided by ChatGPT.

This code sets the socket being used to send the critical packet to throw a timeout exception after 1 second has passed. It then sends the critical packet to the recipient and waits for a response. If a response is not received within 1 second, a timeout exception is thrown, and the while loop is repeated, sending the critical packet once again. If a response is received, it checks that the header indicates receipt of the sent critical packet. If not, the while loop is repeated, sending the critical packet once again. If the header does indicate receipt of the critical packet, we break from the while loop and continue. Below is a table indicating the various critical packet headers and what they indicate. Headers are indicated in hexadecimal for brevity's sake.

Header	**Sender->Recipient**	**Associated Critical Packet**
0x11	Broker->Producer	Confirm receipt of the producer's initialisation request. This process is discussed further under producer functionality.
0x22	Producer->Broker	Confirm receipt of the producer's ID. This process is discussed further under producer functionality
0x33	Broker->Consumer	Confirm receipt of the consumer's request to see the list of currently running streams.

0x44	Broker->Consumer	Confirm receipt of the consumer's new subscription.
0x66	Broker->Consumer	Indicates to the consumer that an attempted subscription has failed.
0x88	Consumer->Broker	Confirm receipt of the broker's notification to a consumer that a stream it was viewing has ended.
0x18	Broker->Producer	Confirm for the producer that the broker has successfully notified all relevant consumers that a stream has ended.

Fig 7: Table of headers for confirmation of all critical packets

5.2 Broker Functionality

The Broker uses a dictionary to keep track of producers, their streams, the content types of those streams, and all subscriptions to those streams.

While the system is running, the broker waits in a while true loop to receive packets from producers and consumers, and handles them accordingly. The below table indicates the types of non-contextual packets the broker receives, the headers that identify them and how the broker handles them. Headers are indicated in hexadecimal for brevity's sake. Xs, Ys, Zs, Vs and Ws are used to indicate variable bytes that affect handling.

Header	**Type**	**Handling**
0xA00000	New Producer Initialisation	The broker generates a new ID for the producer and records the new producer in its dict. Discussed further under producer functionality.
0x900000	Request for List of Running Streams	The broker sends an abbreviated dict back to the sender's IP address, containing a list of running streams but not the addresses of subscribed consumers. Discussed further under consumer functionality.
0x90000X-YZ-VW	Subscription by Consumer	The broker will add the consumer to the list of consumers subscribed to the stream of ID YZ from producer A0000X with content type VW by recording the sender's IP address. If no such stream exists, the broker will indicate as such to the sender.
0xA0000X-YZ-V	Stream	If the broker has not already recorded a stream of

W	Content Packet from Producer	ID YZ from producer A0000X with content type VW in its dict, it will do so. Regardless, it will then forward the packet to all subscribed consumers.
Other	Bad packet	Does nothing.

Fig 8: Table of non-contextual headers for packers received by the broker, and how it handles them

Once a producer has sent the final frame of a stream to the broker, it will then send a packet with the normal stream content overhead, but the content of the packet will indicate that the stream is now over. This packet is considered a critical packet. When the broker receives this packet, it will notify every consumer subscribed to the indicated stream that the stream is now over. Each of these notifications are also considered critical packets. Once each relevant consumer has been notified of the stream's end, the broker will remove that stream from its dict of running streams.

5.3 Producer Functionality

When a producer is initialised, it sends a packet to the broker with the base ID of 0xA00000. When the broker receives a packet from a producer with this ID, it assigns it a new ID counting up from 0xA00001, ie. the first producer will have ID 0xA00001, the second will have ID 0xA00002 and so on. The broker then records that producer and sends it back its new ID. This is how the broker ensures no 2 producers have the same ID.

Both the initial packet from the producer with the uninitialised ID and the response by the broker containing the producer's new ID are considered critical packets. In keeping with the handling of critical packets specified in the design section, the broker will continue to ask the producer for confirmation of receipt of its new ID until it receives this confirmation. In order to prevent a situation where the broker gets stuck waiting for this confirmation in the event that the conformation is lost while being sent over the network, the producer waits for 5 seconds after confirming receipt of its ID to make sure that the broker does not ask for confirmation again, indicating that confirmation of receipt of the ID was successfully received.

A producer tracks how many streams it has run itself and assigns a new ID to each stream as it starts them, counting up from 1. While it is initialising, it randomly decides how many streams it will run concurrently, and once all of its streams end, it starts equally many new streams. The specifics of this process are

ultimately superfluous though; it makes no odds whether or not it always starts the same number of new streams or when it starts the streams.

Each time a producer starts a new stream, it will send the broker a notification indicating the ID of that stream and the type(s) of content it will consist of. This gives the consumers an opportunity to subscribe to that stream before the first content packet is received. This notification is not considered a critical packet however, as the broker will handle a packet belonging to a stream it does not recognise without error.

The last byte of a producer's typical stream content packet is used to indicate the type of content being sent. The below table shows how each type of content is indicated. Bytes are shown in hexadecimal for brevity's sake.

Content Byte	**Indicated Type**
0x00	Video
0xFF	Audio
0xAA	Text
0x55	All of the above

Fig 9: table of bytes used to indicate content types

Note that all 3 types of content are never sent in one packet, but the 0x55 indicator is used by the producer when notifying the broker of a new stream to indicate that it will be sending video, audio and text packets that all belong to the same stream so that the broker will know to expect them. A consumer can also use this indicator to tell the broker that it wants to subscribe to all types of a given stream, rather than just one.

Before a producer starts sending content related to its streams, it measures the length of each stream in terms of the exact number of packets that will need to be sent before the stream ends. For streams with 1 content type, this process is a simple matter of counting the number of frames in the source file. However, for streams with multiple content types, the video, audio and text components may not all be the exact same length. For this reason, a producer considers the length of a stream to be the length of the shortest of its content types to avoid index out of bounds errors.

Once a producer determines that a stream is finished, it sends one final packet to the broker to indicate as such. This packet is considered a critical packet, as it is required so that the consumers will know not to expect further content in the current stream.

5.4 Consumer Functionality

As consumers do not have a unique ID like producers do, they do not need to have an initialisation process. Instead, on startup, a consumer simply chooses a number of streams to subscribe to, and sends the broker a packet querying it for a list of currently running streams. If there are no streams currently available, it will query the broker for a list repeatedly until there is at least one. This packet requesting a list of current streams is considered a critical packet, as consumers would not be able to subscribe to streams if it were not received.

A consumer then randomly selects a number of streams from the available list to view, and sends a subscription request to the broker. The desired stream is indicated in the same way as was discussed in the design section; indicating the stream ID and type in the same way as a producer would when sending a stream content packet, but altering the ID of the desired producer to begin with the bits "1001" instead of "1010" so that the broker can recognise the packet as a subscription request rather than as a content packet. Additionally, instead of indicating a specific stream ID by a given producer, the consumer may use the byte 0x99 to indicate that it will subscribe to all streams of all types by that producer. Subscription request packets from consumers are considered critical packets for much the same reason as stream list requests.

Much of the rest of the consumer's functionality has to do with the frontend handling of the various types of content it can receive, however this is not important to the protocol design which is the primary learning goal of this project, and so I will not dwell on it here.

5.5 Overview of Whole Project

For a better insight into the running of the final project, see the demonstration video included with the submission of the final project. This video assumes the viewer has already read this report, and only serves to demonstrate the functionality described above. It is not an adequate substitute for the above explanation.

The pcap file submitted as part of this submission was run with only 1 consumer and 1 producer, and monitored from before any actors began running, to shortly after the end of stream 1 by producer 1.

6 Reflection

This section will assess the completed project on the basis of its suitability as a solution to the problem statement and reflect on potential improvements or shortcomings with the benefit of hindsight.

6.1 Suitability of Solution

All things considered, I am pleased with the standard to which I have provided a solution to the problem statement. The final build of the project works per the specifications of the assignment descriptor; the system supports 1 to many producers sending multiple streams of various content types to 1 broker that forwards that content to 1 to many consumers on request by those consumers.

6.2 Shortcomings and Potential Improvements

I think the biggest shortcoming in my solution was the handling of critical packets. My decision to handle them as contextually as I did resulted in 2 minor functionality issues. Firstly, while an actor waits for confirmation of receipt of a critical packet it sent, it is unable to properly handle other incoming packets. For example, while a consumer awaits confirmation of a sent subscription request, it is unable to properly handle incoming stream data. The second issue is that the particular approach I took to the confirmation of receipt of critical packets is not airtight. Because the handling of critical packets is done contextually, the recipient cannot always make sure its confirmation of receipt is received before continuing with handling the critical packet. For example, when the broker receives a stream end packet from a producer, it cannot afford to make sure its confirmation of receipt is received, as it must continue to notify the relevant consumers that the stream in question has ended.

The ramifications of these issues are never so catastrophic as to cause the system to fail, however had I chosen to go with a more comprehensive approach to the design and handling of headers, this issue would not be present at all.

Additionally, my decision to have streams with multiple types have all types run concurrently with a roughly equal number of packets meant that I had no proper

frontend implementation for the handling of audio content; video content is displayed in a window and text content is written to the command line, however audio content is not played or written as a file. I had attempted to write audio content to an output file as it came in, however because file writing takes as long as it does, and I have not implemented multithreading in this project, attempts to write audio content to a file frequently meant that much of the stream was missed in the time taken to complete the file write. The effects of this issue could have been reduced were I to send and write the audio in larger chunks, however, as above my design for streams with multiple types relied on each type consisting of roughly the same number of packets. This issue has only to do with the frontend however, which was not the focus of the assignment.

6.3 Personal Reflection

Despite the above shortcomings, I am still pleased with the standard of my solution. Especially considering my lack of experience in this area prior to this assignment. I feel I've learned a great deal about protocol design, and am happy that I was able to engage with the assignment and deliver on it to the standard that I did.

References

https://openai.com/blog/chatgpt

As discussed above, ChatGPT was used to generate the skeleton of the critical packet handling code. As this project involved the use of many Python libraries I was not already familiar with, I also had ChatGPT explain error messages I didn't understand so that I could more easily identify the cause of those errors. ChatGPT also recommended that I use the pydub library for handling audio data.

https://www.w3schools.com/python/python_dictionaries.asp

I referenced the above page a number of times for proper use of python dictionaries.

https://vinodhakumara2681997.medium.com/video-streaming-using-udp-5e3fde9142a0

The above article helped me to get familiar with UDP and cv2 during the early stages of the project.

Note: This pdf was exported from Google Docs, and so some of the formatting is slightly off. The below link is to the original document on Google Docs.

https://docs.google.com/document/d/1iTNmkNncaShjl1UKv5NYlpiGlrz2LmXXeiWu7Vxotek/edit?usp=sharing