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

CSU33031 Computer Networks Assignment #1: Publish/Subscribe Protocol for Audio/Video

Faith Olopade
October 19, 2023

1 Introduction

The rapidly evolving world of digital streaming, characterised by dynamic content creation and distribution, calls for robust protocols capable of addressing challenges in real-time data transfer. The assignment at hand delves into the intricacies of protocol design, emphasising the balance between enhanced functionality through enriched header information and the resultant overhead that such information introduces. Specifically, the assignment sets its focus on developing a Publish/Subscribe protocol tailored for seamless Audio/Video content distribution, employing a UDP-based publish-subscribe mechanism.

Within the confines of this design, a network of actors, consisting of subscribers, a broker, and producers, collaboratively contribute to the protocol's functionality. Envision a real-world scenario: In Real Life (IRL) streamers broadcasting live content. The content, generated by the streamer, travels to a designated broker and from there gets disseminated to a broad audience. Such a system draws parallels with existing solutions like the protocol employed by LiveU, where content traverses through multiple connections to a server before being unified into a singular stream for consumers.

The ensuing report encapsulates a comprehensive exploration of the protocol's design, emphasising its support for multiple publishers and consumers. In the forthcoming sections, I will elucidate the conceptual foundation that underpins the protocol, elaborate on its binary header design, and delineate the overarching communication dynamics of the entire system.

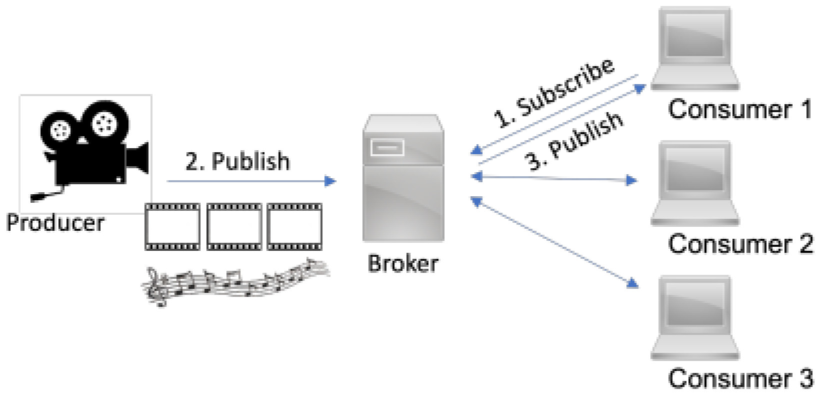


Figure 1: A high-level representation of the Publish/Subscribe protocol for digital streaming. A producer, symbolised by a camera or streaming device, transmits content to the central broker. The broker, in turn, disseminates this content to various consumers who have subscribed to the content. This depicts the fundamental flow of data from content creation to distribution in real-time streaming scenarios.

2 Background

The advancements in computer networks have revolutionised the manner in which content is distributed and consumed. A fundamental cornerstone in achieving this feat rests upon the design and implementation of robust and efficient networking protocols. These protocols, which define the rules for data exchange between computing entities, ensure that data is transmitted securely, reliably, and efficiently over a network. This section delves into the foundational concepts of networking and the protocols that facilitate the transfer of multimedia content, specifically focusing on the User Datagram Protocol (UDP) and its relevance in this assignment.

2.1 Technical Background

Networking and Protocols:
At its core, a computer network is a collection of interconnected computing devices that can communicate with one another. The ability to transmit data between devices in an understandable format is governed by protocols. A protocol, in the context of networking, is a set of standardised rules that dictate how data is formatted, transmitted, received, and processed. These rules ensure that devices, often with different architectures and operating systems, can communicate with one another effectively.

Protocols are layered in a hierarchical fashion, with each layer providing specific functionalities. These layers collectively form the protocol stack. A widely adopted model that explains this layered structure is the OSI (Open Systems Interconnection) model, which breaks down the networking process into seven layers, from the physical transmission of bits to the highest-level representation of data.

Basics of UDP Communication:
The User Datagram Protocol (UDP) is one of the core protocols of the Internet Protocol (IP) suite and operates at the transport layer of the OSI model. Unlike its counterpart, the Transmission Control Protocol (TCP), UDP is connectionless. This means that communication does not necessitate an established connection between the sender and receiver. As a result, UDP typically incurs less overhead, enabling faster data transmission rates.

A UDP datagram consists of a UDP header (8 bytes) followed by the data (payload). The UDP header is shown in Table 1.

Bytes Content	0	1	2	3	4	5	6	7
	Source port		Destination port		Length		Checksum	

Table 1: UDP Datagram Protocol Header. The source port refers to the port the datagram has been sent from. The destination port refers to the port the datagram is being sent to. The length refers to the length in bytes of the datagram as a whole (header and data). The checksum field may be used for error checking, although it is not in the solution discussed here.

Some key characteristics of UDP include:

1. **Connectionless Nature:** UDP does not establish a formal connection before data transmission, making it suitable for applications where speed is more critical than reliability.

2. **No Guarantee of Delivery:** UDP does not guarantee the delivery of datagrams. This means that datagrams may arrive out of order, be duplicated, or not arrive at all.
3. **Checksums for Data Integrity**: While UDP does not ensure delivery, it does use checksums to verify that the transmitted data has not been corrupted during transit.

Role of UDP in the Assignment:
The assignment emphasises the creation of a protocol that enables the distribution of video, audio, and text content using a publish-subscribe mechanism based on UDP datagrams. The inherent features of UDP make it an attractive choice for real-time applications, such as streaming, where timely delivery of data is crucial. For instance, in a live stream, a few missed frames due to dropped datagrams might be acceptable, but a delay in data delivery can lead to a compromised user experience. The protocol's design, as detailed in the assignment, will need to address UDP's shortcomings, like the potential out-of-order delivery of frames, while capitalising on its strengths to provide an efficient content distribution system.

This assignment's focal point is the development of a publish-subscribe protocol based on UDP datagrams for the distribution of video, audio, and text content. The publish-subscribe model is a messaging paradigm where senders ('publishers') categorise published messages into classes without knowing the recipients of the messages. Conversely, subscribers express interest in one or more classes and receive messages of interest without knowing the publishers. Leveraging UDP's characteristics, this model can be implemented to ensure quick, efficient data dissemination.

The technical implementation of this protocol was executed in Python, a high-level, versatile programming language renowned for its clarity and ease of use. Python provides a robust library for socket programming, facilitating the creation of networking applications. In this project, Python 3 was utilised, avoiding Python 2.7 due to its obsolete handling of Datagram sockets. Specifically, Python's socket library was employed to create UDP sockets, allowing for the sending and receiving of datagrams.

Furthermore, for this assignment, the code was developed and debugged using Visual Studio Code (VS Code). This popular Integrated Development Environment (IDE) offers a rich set of features, including syntax highlighting, debugging tools, and extensions that amplify productivity.

To examine the network traffic generated during the protocol's execution, Wireshark, a renowned network protocol analyzer, was utilised. Wireshark offers an intuitive interface to capture and interactively browse the traffic running on a computer network. By inspecting packets and their headers, one can discern valuable information regarding the data's nature and its path across the network.

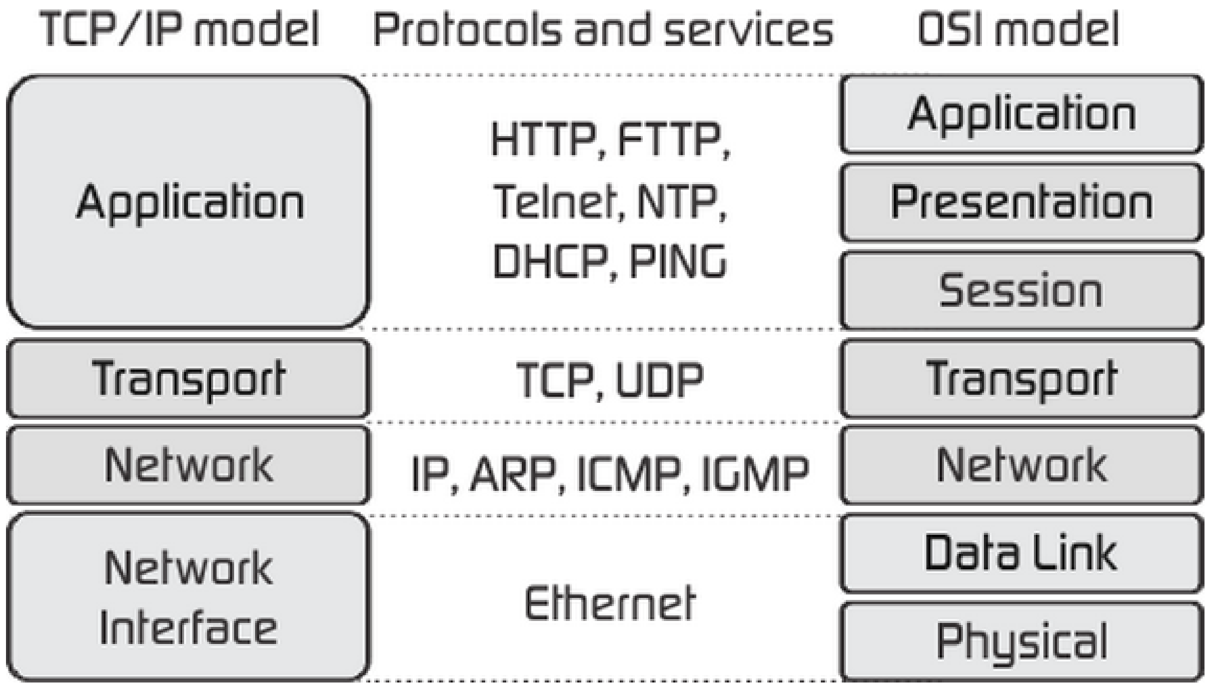


Figure 2: Diagrammatic representation of the OSI (Open Systems Interconnection) model, highlighting the position of the User Datagram Protocol (UDP) within the transport layer.

2.2 Closely-Related Projects

This section presents projects and concepts related to the design and implementation of the Publish/Subscribe protocol for Audio/Video assignment. While there exist many protocols in the realm of networking and multimedia distribution, I focus on aspects pertinent to the project, such as header design, content distribution mechanism, and real-time streaming protocols.

<u>Secure Reliable Transport (SRT)</u>
Secure Reliable Transport (SRT) is a streaming protocol that was developed to provide secure and reliable transport of video content over the internet. The key aspects of SRT include:

- **Header Design:** SRT uses a simple header design with a clear distinction between control and data packets. This separation allows for efficient routing and processing of the transmitted data, ensuring minimal latency during live streaming.

- **Error Recovery:** SRT implements a sophisticated error recovery mechanism. If a packet is not acknowledged within a certain timeframe, it's retransmitted, ensuring that viewers experience uninterrupted streaming.

- **Security:** The protocol offers end-to-end encryption, safeguarding the content from potential eavesdroppers.

Lessons from SRT:

A clear and straightforward header design can lead to efficient data processing and low-latency streaming. Furthermore, a robust error recovery mechanism is crucial for delivering high-quality streams over unreliable networks.

LiveU's Proprietary Protocol
LiveU's streaming solution involves distributing a single video stream over multiple connections. Notable aspects include:

- **Distributed Streaming:** The protocol splits the video stream and sends parts of it over different connections. This ensures a higher chance of successful delivery, even if one or more connections face disruptions.

- **Dynamic Adaptation:** Depending on the quality of the network connections, the protocol can adjust the bit rate and other streaming parameters to provide the best possible viewer experience.

Lessons from LiveU:
The distribution of video data over multiple channels or connections can enhance reliability and adaptability, ensuring consistent streaming quality in fluctuating network conditions.

Local Protocol Implementation
In the project's context, the following are the key aspects:

- **Header Design:** The header in my protocol comprises a packet type identifier, producer ID, stream number, frame number, payload length, and content type. This design supports easy identification of content and aids in its distribution to the relevant subscribers.

- **Content Distribution:** Based on the content type—video, text, or audio—the respective content is processed and sent. The broker efficiently routes this content to subscribers based on their preferences.

- **UDP-based Mechanism:** Utilising the User Datagram Protocol (UDP) ensures a lightweight and fast mechanism, suitable for real-time content distribution, although it requires careful management to handle out-of-order or lost packets.

Lessons from the Local Protocol:
A well-structured header aids in efficient content distribution. Using UDP can achieve real-time performance but requires careful error-handling mechanisms to tackle its inherent unreliability.

2.2.1 Summary

The table below provides a summarised view of the related projects and their key takeaways:

Project/Protocol	**Key Aspects**	**Lessons Learned**

Secure Reliable Transport (SRT)	Clear Header Design, Error Recovery, Security	Importance of header clarity and error mechanisms
LiveU's Proprietary Protocol	Distributed Streaming, Dynamic Adaptation	Multi-channel streaming enhances reliability
Local Protocol Implementation	Structured Header, Content Distribution, UDP-based	Clear headers aid efficient routing; manage UDP's unreliability

3 Problem Statement

During my studies and analysis of protocols, particularly in the domain of audio, video, and text content distribution using a publish-subscribe mechanism, I noticed several challenges and shortcomings in the existing approaches. The CSU33031 assignment's primary aim was to develop a protocol that ensures efficient distribution of such content based on UDP datagrams while maintaining the binary encoding of header information. This exercise entailed managing multiple actors, including subscribers, a broker, and several producers.

One of the prominent problems to address was the fragmentation and ordering of frames sent by the producer. Since video streams comprise individual frames sent by a producer to a broker, maintaining a smooth user experience becomes paramount. The challenge lies in ensuring that older frames do not display after newer ones, even if they arrive out of order, as it degrades the viewing experience significantly. Furthermore, considering the scenario where multiple producers are involved and each producer could have one or more streams, the management of these streams becomes crucial. The protocol should effectively handle the scenario where multiple consumers wish to subscribe either to specific streams or all streams from a given producer. Thus, the solution needs to take into account the complexities introduced by numerous participants in the network.

Taking inspiration from proprietary solutions such as LiveU, which combines traffic from various connections into a singular stream for consumers, I evaluated options for functionalities to be implemented. One interesting dilemma was deciding on the depth and breadth of functionalities - whether a producer should only send out frames or also integrate audio, or whether producers and consumers should exchange text related to a stream.

In my implementation, my protocol solution manages distinct content types, such as video, text, and audio. The producer script prompts for the content type, encoding the relevant frame, and sending it to the broker. On the other side, the broker handles the incoming data, interpreting subscription requests, and appropriately forwarding content to the relevant subscribers.

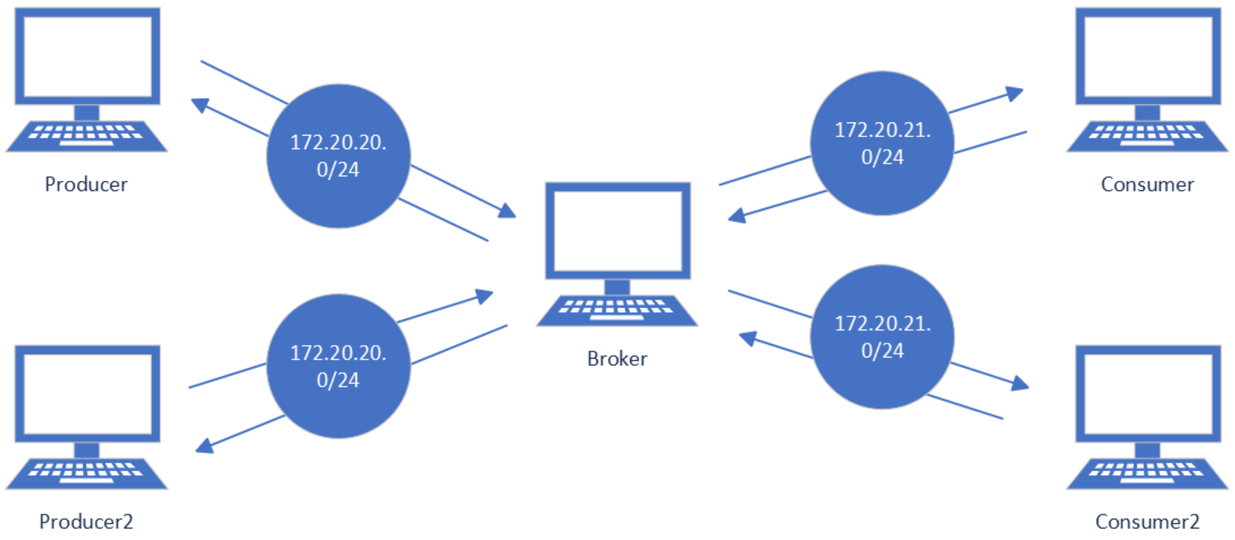


Figure 3: Architectural representation of the protocol's topology, depicting the interaction between Producers, the Broker, and Consumers across distinct IP networks, showcasing the importance of efficient routing and content distribution.

4 Design

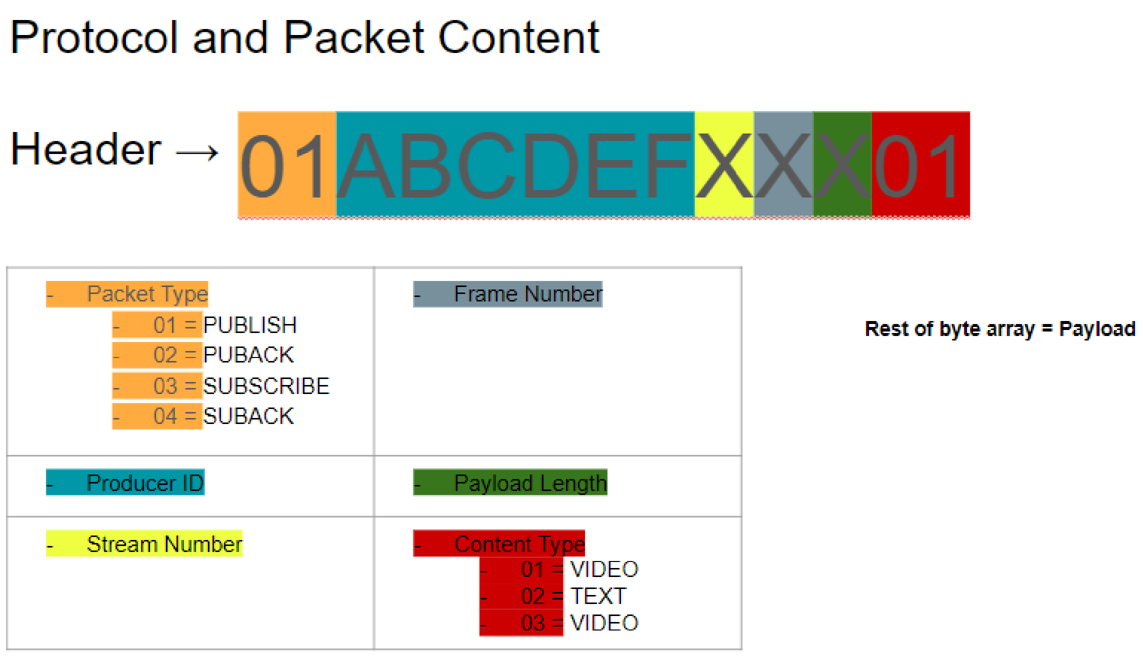
Within this section, I will provide an insightful, high-level design discussion on my solution for the Publish/Subscribe Protocol for Audio/Video assignment. My primary motivation for this design approach arises from the challenge of ensuring timely and efficient delivery of multimedia content, particularly under the conditions of UDP where packet delivery is not guaranteed and packets can arrive out of order.

4.1 Protocol Header Design

In the design of my protocol, the header is crucial for efficiently identifying, routing, and managing packets. Given the assignment's focus on binary encoding, the header for my packets was designed as follows:

1. **Packet Type (1 byte):** This helps determine if the packet is a publication, subscription, or acknowledgment.
2. **Producer ID (6 bytes):** Encoded in ASCII, this uniquely identifies the producer, like "ABCD99".
3. **Stream Number (1 byte):** Specifies the stream from a given producer, which could range from 0-255.
4. **Frame Number (4 bytes):** Indicates the sequence of frames ensuring proper order at the consumer's end.
5. **Payload Length (4 bytes):** This determines the size of the accompanying payload.
6. **Content Type (1 byte):** Identifies if the payload is video, text, or audio.

This compact header design balances functionality with minimal overhead. It facilitates identifying subscriptions and managing content dissemination by the broker while also supporting the unique needs of multimedia streams.



- Packet Type - 01 = PUBLISH - 02 = PUBACK - 03 = SUBSCRIBE - 04 = SUBACK	- Frame Number
- Producer ID	- Payload Length
- Stream Number	- Content Type - 01 = VIDEO - 02 = TEXT - 03 = VIDEO

Figure 4: Packet structure

4.2 Content Distribution Mechanism

The system comprises multiple actors: subscribers, a broker, and producers. A high-level operation is as follows:

- **Producers:** They create content and use the aforementioned header structure to send packets to the broker. For example, the Producer.py code allows for different content types, each assigned a unique byte code (VIDEO=1, TEXT=2, AUDIO=3). For video and audio content, existing files are utilised, while text content is input by the user.

- **Broker:** The Broker.py code is responsible for forwarding packets from producers to subscribers. It maintains a dynamic subscription list and uses the header's Producer ID and Stream Number to determine the appropriate subscribers. The broker ensures that packets, particularly video frames, are relayed in sequence, maintaining the viewing experience's quality.

- **Subscribers:** They issue subscription requests to the broker, specifying the desired producer and potentially a specific stream. Once subscribed, they receive content relevant to their subscriptions.

```
Producer.py
|
|  [Packet Header]
|  +---------------------+
|  | Packet Type        |
|  +---------------------+
|  | Producer ID        |
|  +---------------------+
|  | Stream Number      |
|  +---------------------+
|  | Frame Number       |
|  +---------------------+
|  | Payload Length     |
|  +---------------------+
|  | Content Type       |
|  +---------------------+
|
V
Broker.py
|
|  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
|
|  (Maintains Subscription List)
|  (Routes packets using Producer ID & Stream Number)
|  (Ensures Packet Order)
|  (Sends Acknowledgment)
|
V
Subscriber.py
   |
   |  [Subscription Request]
   |  (Specific Producer & Stream)
   |
   +--------------------< Content Reception (Video, Text, Audio)

Key:
(|) represents the flow of information from one entity to another
(-) represents processes or actions within the given entity
(V) shows the direction of the flow
```

Figure 5: Flow Chart detailing the Publish/Subscribe Protocol's design, showcasing packet header structure, and the interaction between system entities for efficient multimedia content delivery

4.3 Handling Out-of-Order Frames

Given that frames can arrive out of order in UDP, my solution emphasises the frame number in the packet header. By tracking this at the consumer's end, the protocol can ensure that older frames aren't rendered after newer ones, preventing any unexpected viewing experiences.

4.4 Scalability and Multimodal Support

The protocol supports multiple producers and consumers concurrently. By uniquely identifying producers and streams in the header, it can efficiently handle scenarios like three producers each having two consumers.

Moreover, with built-in support for video, text, and audio content, it offers flexibility in multimedia distribution. This feature might be especially appealing for applications like streaming platforms where different content modalities coexist.

4.5 Acknowledgments

To ensure reliability over UDP, the protocol includes acknowledgment mechanisms. After a producer sends a packet to the broker or when a consumer subscribes to content, they await an acknowledgment. This verifies that the broker has received the content or the subscription request, respectively.

4.6 Future Enhancements

Considering LiveU's proprietary protocol that uses multiple connections for redundancy, a future design iteration could incorporate multi-path content delivery. By sending different parts of content over different routes, this could enhance reliability and performance, especially in network conditions where certain paths might be congested or unreliable.

5 Implementation

In the process of designing a protocol to support the publish/subscribe mechanism for audio, video, and text content over UDP datagrams, several distinct components were created. The core components of this solution are the Producer, which generates and sends content, and the Broker, responsible for managing subscriptions and forwarding content. The following sections delve into the specific implementation details of these components, and provide code snippets essential to the understanding of the designed protocol.

5.1 Producer

The Producer serves as the entity generating content (audio, video, text) and forwarding it to the Broker. Implemented in Python, the Producer was structured around a loop where the type of content to send is input by the user.

Based on the content type, appropriate frame data is fetched. Video frames are extracted from an image file, audio frames from an audio file segment, and text directly from the user's input.

Each content type carries a specific byte identifier (VIDEO, TEXT, AUDIO) which is embedded into the header. The header, crafted in a binary format, encodes crucial metadata: producer's ID, stream number, current frame number, payload length, and the content type.

```
header = b'\x01' + producer_id.encode() + stream_no.to_bytes(1,
byteorder='big') + current_frame_no.to_bytes(4, byteorder='big') +
payload_length + content_type_byte
UDPClientSocket.sendto(header + payload_bytes, brokerAddressPort)
```

Listing 1: Key segment of the Producer's code showing header construction and data transmission to the Broker.

5.2 Broker

The Broker is the centrepiece of the solution, receiving content from the Producers, managing subscriptions, and subsequently forwarding content to the subscribed Consumers. The Broker listens indefinitely for incoming UDP datagrams.

When data arrives, it is parsed to determine its type (subscription request or content data). In case of a subscription request, the Broker updates its subscription list. For content data, the Broker forwards it to the appropriate subscribers based on the embedded metadata.

```
if packet_type == 3:  # Subscription packet
    ...
    # Update subscriptions list
    ...
    UDPServerSocket.sendto(b'\x04', addr)  # Send subscribe ack to
the consumer
```

Listing 2: Segment from the Broker's code that handles subscription requests and acknowledges them.

On receiving content from a producer, the Broker checks its subscription list to determine which Consumers should receive the content. This determination is based on the stream's unique identifier (a combination of the producer's ID and the stream number). The content, without any alteration, is then forwarded to the appropriate subscribers.

```
for sub_stream_id in [stream_id, producer_id + "ALL"]:
    if sub_stream_id in subscriptions:
        for subscriber in subscriptions[sub_stream_id]:
            UDPServerSocket.sendto(data, subscriber)
```

Listing 3: Forwarding received content to the appropriate subscribers.

This implementation ensures that content is routed efficiently from Producers to Consumers through the Broker, adhering to the publish/subscribe model. The binary format of headers ensures minimal overhead, promoting effective content distribution even in real-time streaming scenarios.

5.3 Consumer

The consumer, as the endpoint in this setup, subscribes to specific streams of content provided by the producers. The consumer specifies which producer's stream it wishes to subscribe to and sends a subscription request to the broker. Once subscribed, it starts receiving content (video, audio, or text) from the broker, which it can then process. The consumer also has the flexibility to subscribe to all streams from a specific producer by using the special stream number 255.

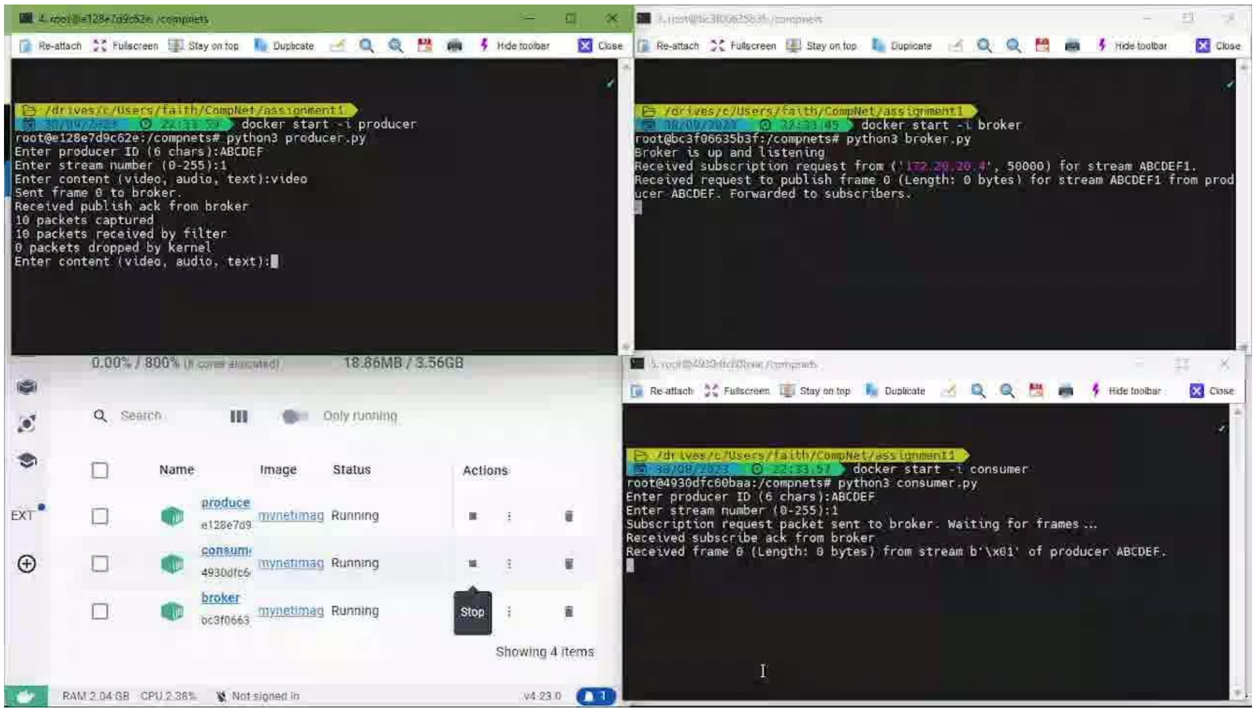
```
# Snippet from Consumer.py indicating subscription and data reception
header = b'\x03' + producer_id.encode() + stream_no.to_bytes(1,
byteorder='big')
UDPClientSocket.sendto(header, brokerAddressPort)
data, _ = UDPClientSocket.recvfrom(bufferSize)
```

Each content type (video, audio, or text) received by the consumer is displayed in bytes but in future implementations could be enhanced. For instance, video frames can be displayed, audio frames played, and text frames printed or logged.

5.4 Evolution of the Implementation

My solution evolved through several distinct phases, each building upon the last and adding new features and capabilities. To showcase this progression, I've provided a visual representation of the assignment's three parts:

Part 1: Initial Design
In the first iteration, the system was simplified, encompassing just one producer, one broker, and one consumer. This version was limited to sending text content.



```
/drives/c/Users/faith/CompNet/assignment1
30/09/2023  22:33:59  docker start -i producer
root@e128e7d9c62e:/compnets# python3 producer.py
Enter producer ID (6 chars):ABCDEF
Enter stream number (0-255):1
Enter content (video, audio, text):video
Sent frame 0 to broker.
Received publish ack from broker
10 packets captured
10 packets received by filter
0 packets dropped by kernel
Enter content (video, audio, text):
```

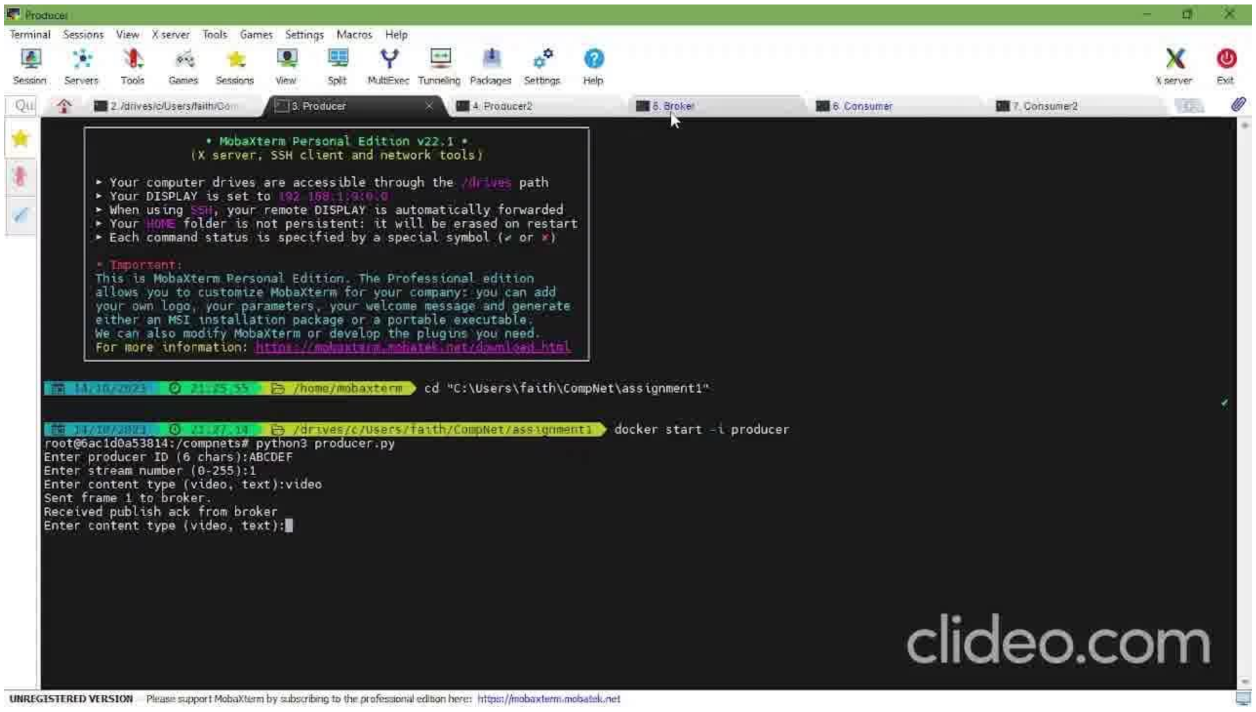
```
/drives/c/Users/faith/CompNet/assignment1
30/09/2023  22:33:45  docker start -i broker
root@bc3f06635b3f:/compnets# python3 broker.py
Broker is up and listening
Received subscription request from ('172.20.20.4', 50000) for stream ABCDEF1.
Received request to publish frame 0 (Length: 0 bytes) for stream ABCDEF1 from prod
ucer ABCDEF. Forwarded to subscribers.
```

```
/drives/c/Users/faith/CompNet/assignment1
30/09/2023  22:33:57  docker start -i consumer
root@4930dfc60baa:/compnets# python3 consumer.py
Enter producer ID (6 chars):ABCDEF
Enter stream number (0-255):1
Subscription request packet sent to broker. Waiting for frames ...
Received subscribe ack from broker
Received frame 0 (Length: 0 bytes) from stream b'\x01' of producer ABCDEF.
```

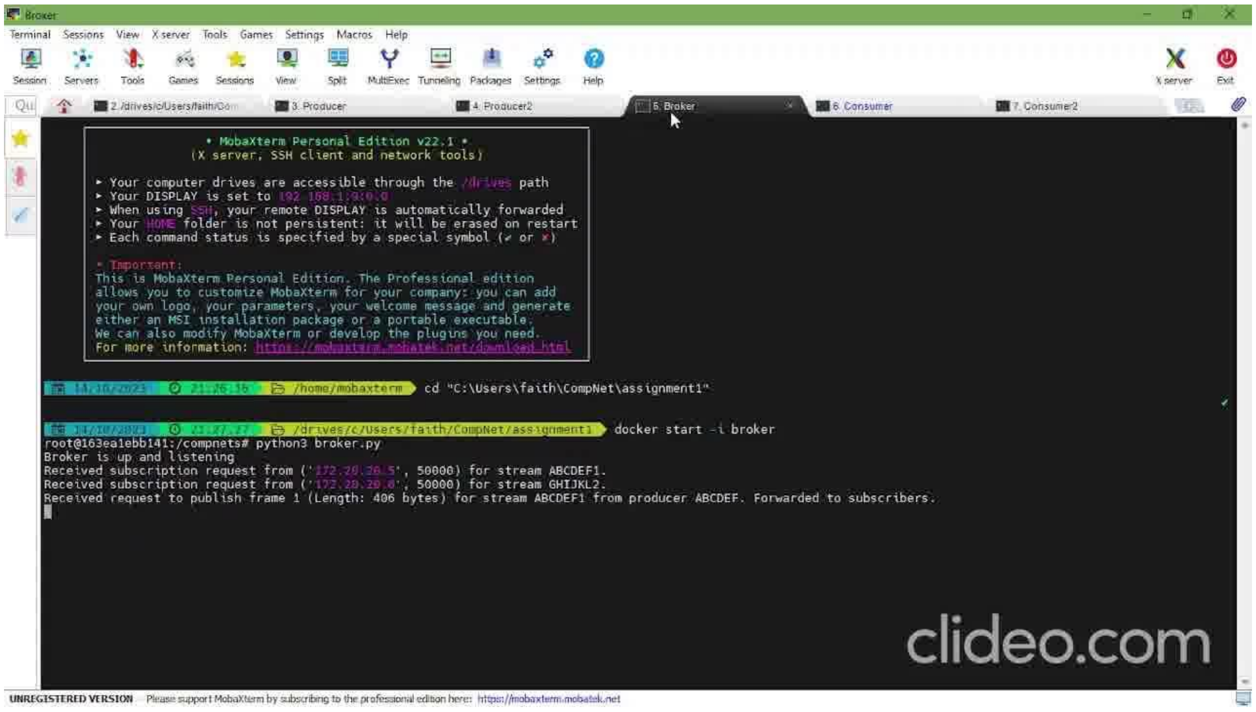
Name	Image	Status
produce e128e7d9	mynetimag	Running
consum 4930dfc6	mynetimag	Running
broker bc3f0663	mynetimag	Running

Initial Design with Docker Containers and PCAP file
Figure 6: The architecture of Part 1, showing a singular producer, broker, and consumer, with Docker containers running in the background. This setup is exclusively geared towards text content.

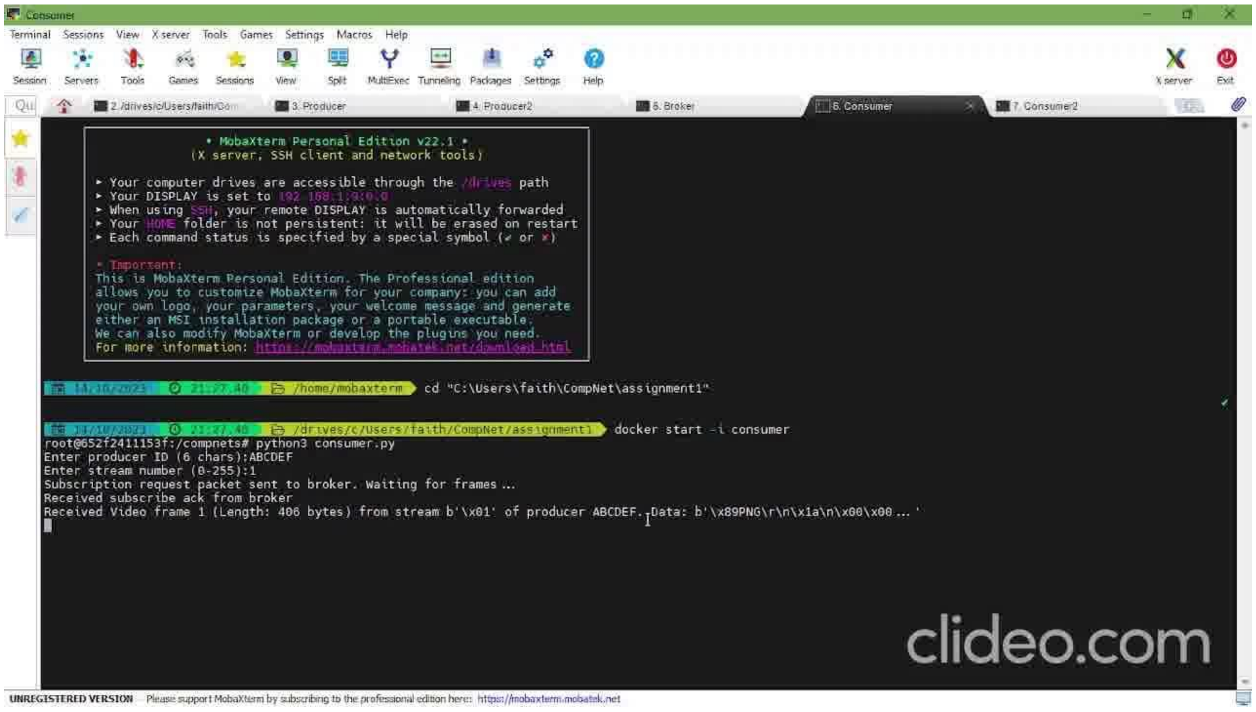
Part 2: Video Integration and Scaling
My second iteration introduced the capability to send video content. Additionally, the system was scaled to accommodate multiple producers and consumers, making it more robust.



```
• MobaXterm Personal Edition v22.1 •
(X server, SSH client and network tools)

► Your computer drives are accessible through the /drives path
► Your DISPLAY is set to 192.168.1.9:0.0
► When using SSH, your remote DISPLAY is automatically forwarded
► Your HOME folder is not persistent: it will be erased on restart
► Each command status is specified by a special symbol (✓ or ✗)

• Important:
This is MobaXterm Personal Edition. The Professional edition
allows you to customize MobaXterm for your company: you can add
your own logo, your parameters, your welcome message and generate
either an MSI installation package or a portable executable.
We can also modify MobaXterm or develop the plugins you need.
For more information: https://mobaxterm.mobatek.net/download.html

14/10/2023  21:25:55  /home/mobaxterm  cd "C:\Users\faith\CompNet\assignment1"

14/10/2023  21:27:14  /drives/c/Users/faith/CompNet/assignment1  docker start -i producer
root@6ac1d0a53814:/compnets# python3 producer.py
Enter producer ID (6 chars):ABCDEF
Enter stream number (0-255):1
Enter content type (video, text):video
Sent frame 1 to broker.
Received publish ack from broker
Enter content type (video, text):
```



```
• MobaXterm Personal Edition v22.1 •
(X server, SSH client and network tools)

► Your computer drives are accessible through the /drives path
► Your DISPLAY is set to 192.168.1.9:0.0
► When using SSH, your remote DISPLAY is automatically forwarded
► Your HOME folder is not persistent: it will be erased on restart
► Each command status is specified by a special symbol (✓ or ✗)

• Important:
This is MobaXterm Personal Edition. The Professional edition
allows you to customize MobaXterm for your company: you can add
your own logo, your parameters, your welcome message and generate
either an MSI installation package or a portable executable.
We can also modify MobaXterm or develop the plugins you need.
For more information: https://mobaxterm.mobatek.net/download.html

14/10/2023  21:26:16  /home/mobaxterm  cd "C:\Users\faith\CompNet\assignment1"

14/10/2023  21:27:27  /drives/c/Users/faith/CompNet/assignment1  docker start -i broker
root@163ea1ebb141:/compnets# python3 broker.py
Broker is up and listening
Received subscription request from ('172.20.20.5', 50000) for stream ABCDEF1.
Received subscription request from ('172.20.20.6', 50000) for stream GHIJKL2.
Received request to publish frame 1 (Length: 406 bytes) for stream ABCDEF1 from producer ABCDEF. Forwarded to subscribers.
```



```
        • MobaXterm Personal Edition v22.1 •
      (X server, SSH client and network tools)

 ► Your computer drives are accessible through the /drives path
 ► Your DISPLAY is set to 192.168.1.9:0.0
 ► When using SSH, your remote DISPLAY is automatically forwarded
 ► Your HOME folder is not persistent: it will be erased on restart
 ► Each command status is specified by a special symbol (✔ or ✘)

 • Important:
 This is MobaXterm Personal Edition. The Professional edition
 allows you to customize MobaXterm for your company: you can add
 your own logo, your parameters, your welcome message and generate
 either an MSI installation package or a portable executable.
 We can also modify MobaXterm or develop the plugins you need.
 For more information: https://mobaxterm.mobatek.net/download.html

 14/10/2023  21:27.40  /home/mobaxterm  cd "C:\Users\faith\CompNet\assignment1"

 14/10/2023  21:27.40  /drives/c/Users/faith/CompNet/assignment1  docker start -i consumer
root@652f2411153f:/compnets# python3 consumer.py
Enter producer ID (6 chars):ABCDEF
Enter stream number (0-255):1
Subscription request packet sent to broker. Waiting for frames ...
Received subscribe ack from broker
Received Video frame 1 (Length: 406 bytes) from stream b'\x01' of producer ABCDEF. Data: b'\x89PNG\r\n\x1a\n\x00\x00 ... '
```

Consumer
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
2. /drives/c/Users/faith/Com
3. Producer
4. Producer2
5. Broker
6. Consumer
7. Consumer2
clideo.com
UNREGISTERED VERSION Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net

Enhanced Design with Multiple Producers and Consumers
Figure 7,8,9: The architecture of Part 2, illustrating the expanded capabilities to handle video/audio content and multiple producers and consumers.

6 Discussion

In this section, I provide a critical evaluation of my publish-subscribe protocol designed for audio/video content based on UDP datagrams. By delving into the nuances of the protocol, its appropriateness to address the assignment's objectives is analysed, and its strengths and drawbacks are also weighed.

- **Suitability to the Assignment's Objective:**
  The protocol was fundamentally aimed to create a publish-subscribe mechanism for various content types, particularly video, audio, and text, keeping the format binary. Given the requirement to use UDP datagrams, it is evident from the codebase of Producer.py, Broker.py, and Consumer.py that this objective was accomplished. The protocol architecture involves producers, brokers, and consumers, as highlighted in the assignment, and the Python files faithfully execute their respective roles.

- **Advantages:**

1. **Flexibility:** The protocol provides flexibility in terms of content type, allowing for the transmission of video, audio, and text. This flexibility is apparent in the Producer.py where users are prompted to input the content type they wish to send.

2. **Binary Header Design:** Adhering to the assignment's requirement, the protocol utilises a binary format for its headers. This approach aids in compactness and speed when transmitting data over the network. The detailed use of binary conversions in the protocol, such as to_bytes, confirms the implementation of this feature.
3. **Error Handling:** While the protocol is based on UDP, which doesn't guarantee delivery, the producer waits for an acknowledgment after sending each frame. This ensures that the broker has successfully received the frame and potentially reduces the loss of frames.

4. **Scalability:** The broker is designed to manage multiple subscriptions efficiently. This is evident from the Broker.py code, where subscriptions are managed using a dictionary, facilitating quick lookups and modifications.

5. **Order Management:** As video streams can't afford to display older frames after newer ones, the protocol incorporates frame numbers, allowing the consumer to process frames in their correct order.

- **Disadvantages:**

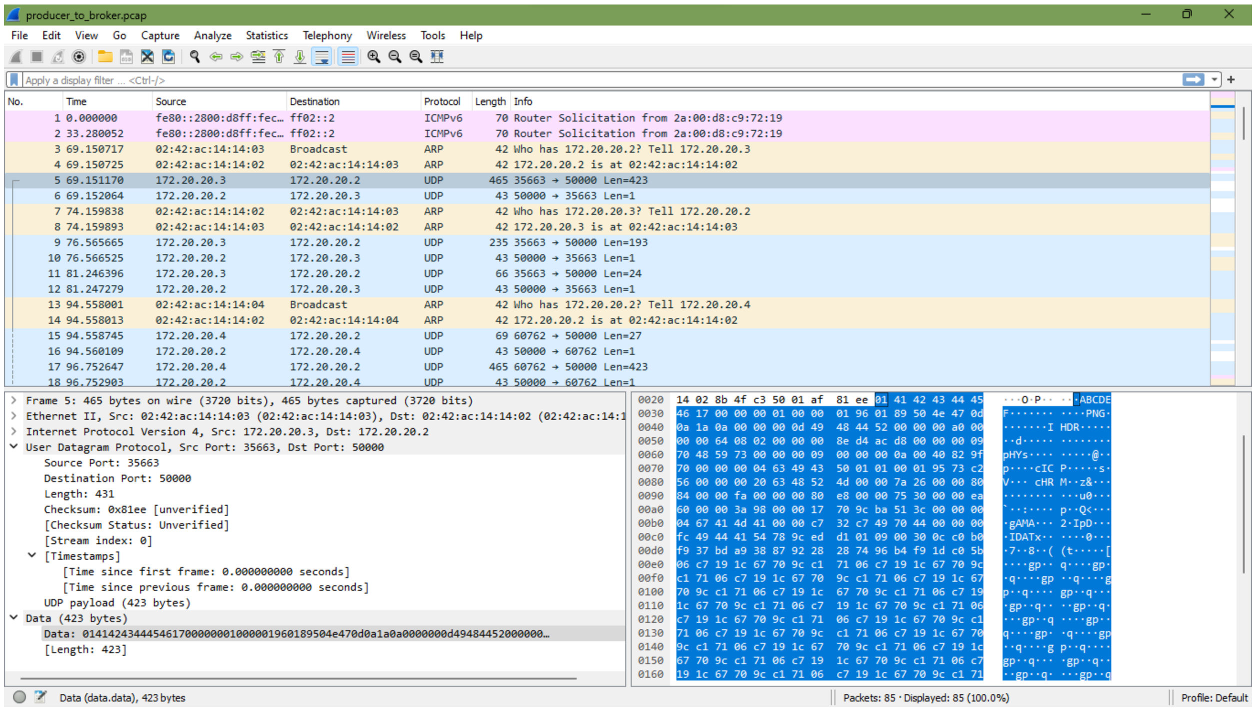
1. **UDP Limitations:** Using UDP means there's no guarantee of data packet delivery, and the protocol might not be suitable for applications where packet loss can't be afforded. While the protocol attempts to mitigate this by waiting for acknowledgments, it doesn't inherently guarantee delivery or correct sequencing on the consumer side.

2. **Header Overhead:** Each packet sent contains a header with producer_id, stream_no, and other metadata. For large numbers of small packets, this could lead to inefficiencies due to the repetitive transmission of this metadata.

3. **Lack of Security Measures:** The protocol doesn't implement any encryption or security measures for the transmitted content. Given the rise of security concerns in modern-day networking, this is a significant drawback.

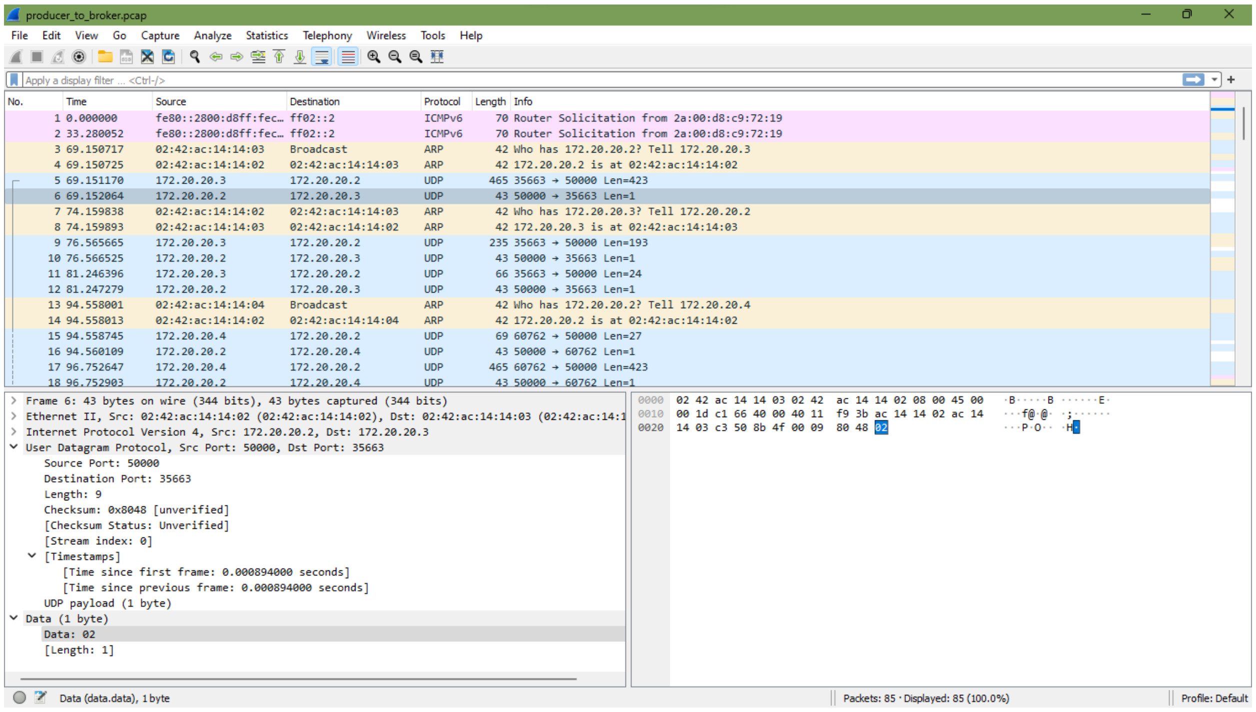
- **Comparison with SRT:**

The Secure Reliable Transport (SRT) protocol, as referenced in the assignment, provides a robust means of streaming video content. While my protocol offers a simplified mechanism based on the publish-subscribe model, SRT has inherent advantages in terms of reliability and error correction, stemming from its roots in being designed explicitly for streaming. However, for a simple, lightweight implementation in controlled environments, my protocol offers a practical solution without the overheads of more comprehensive protocols like SRT.



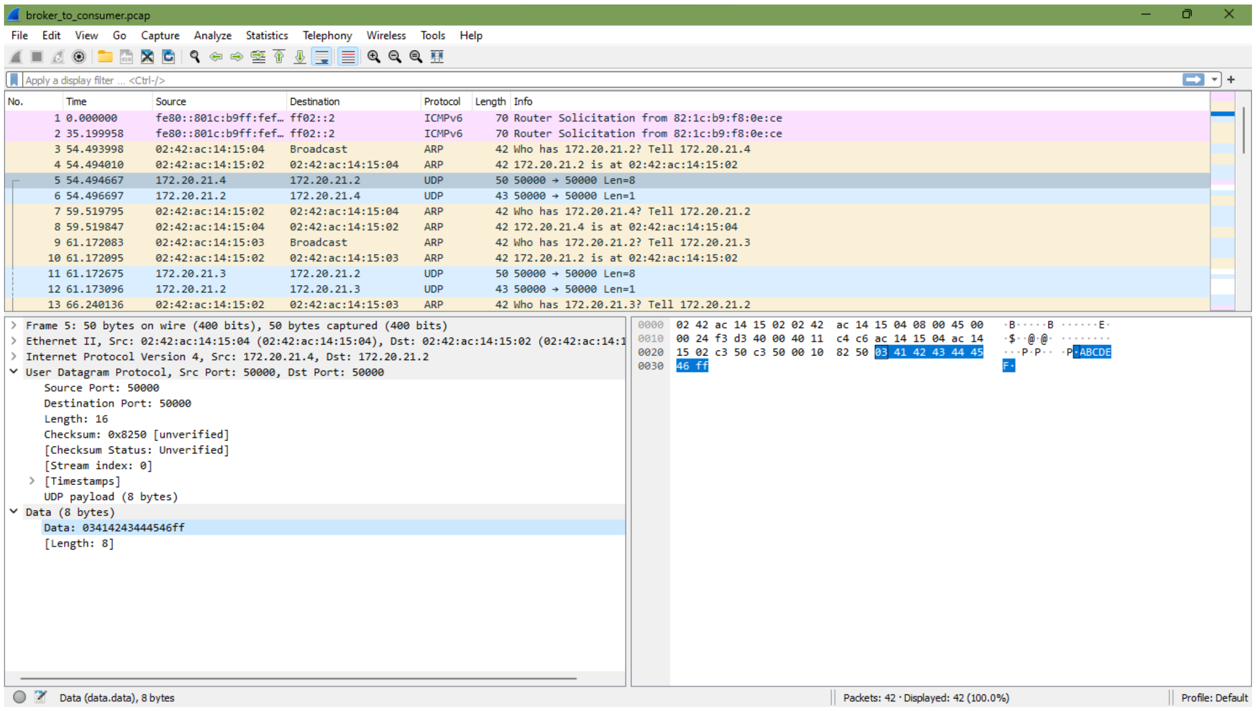
```
producer_to_broker.pcap

No.  Time       Source               Destination          Protocol Length Info
   1 0.000000   fe80::2800:d8ff:fec… ff02::2              ICMPv6     70 Router Solicitation from 2a:00:d8:c9:72:19
   2 33.280052  fe80::2800:d8ff:fec… ff02::2              ICMPv6     70 Router Solicitation from 2a:00:d8:c9:72:19
   3 69.150717  02:42:ac:14:14:03    Broadcast            ARP        42 Who has 172.20.20.2? Tell 172.20.20.3
   4 69.150725  02:42:ac:14:14:02    02:42:ac:14:14:03    ARP        42 172.20.20.2 is at 02:42:ac:14:14:02
   5 69.151170  172.20.20.3          172.20.20.2          UDP       465 35663 → 50000 Len=423
   6 69.152064  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
   7 74.159838  02:42:ac:14:14:02    02:42:ac:14:14:03    ARP        42 Who has 172.20.20.3? Tell 172.20.20.2
   8 74.159893  02:42:ac:14:14:03    02:42:ac:14:14:02    ARP        42 172.20.20.3 is at 02:42:ac:14:14:03
   9 76.565665  172.20.20.3          172.20.20.2          UDP       235 35663 → 50000 Len=193
  10 76.566525  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
  11 81.246396  172.20.20.3          172.20.20.2          UDP        66 35663 → 50000 Len=24
  12 81.247279  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
  13 94.558001  02:42:ac:14:14:04    Broadcast            ARP        42 Who has 172.20.20.2? Tell 172.20.20.4
  14 94.558013  02:42:ac:14:14:02    02:42:ac:14:14:04    ARP        42 172.20.20.2 is at 02:42:ac:14:14:02
  15 94.558745  172.20.20.4          172.20.20.2          UDP        69 60762 → 50000 Len=27
  16 94.560109  172.20.20.2          172.20.20.4          UDP        43 50000 → 60762 Len=1
  17 96.752647  172.20.20.4          172.20.20.2          UDP       465 60762 → 50000 Len=423
  18 96.752903  172.20.20.2          172.20.20.4          UDP        43 50000 → 60762 Len=1

> Frame 5: 465 bytes on wire (3720 bits), 465 bytes captured (3720 bits)
> Ethernet II, Src: 02:42:ac:14:14:03 (02:42:ac:14:14:03), Dst: 02:42:ac:14:14:02 (02:42:ac:14:1
> Internet Protocol Version 4, Src: 172.20.20.3, Dst: 172.20.20.2
v User Datagram Protocol, Src Port: 35663, Dst Port: 50000
    Source Port: 35663
    Destination Port: 50000
    Length: 431
    Checksum: 0x81ee [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  v [Timestamps]
      [Time since first frame: 0.000000000 seconds]
      [Time since previous frame: 0.000000000 seconds]
    UDP payload (423 bytes)
v Data (423 bytes)
    Data: 0141424344454617000000010000019601895O4e470d0a1a0a0000000d49484452000000…
    [Length: 423]

0020  14 02 8b 4f c3 50 01 af  81 ee 01 41 42 43 44 45   ···O·P·· ···ABCDE
0030  46 17 00 00 00 01 00 00  01 96 01 89 50 4e 47 0d   F······· ····PNG·
0040  0a 1a 0a 00 00 00 0d 49  48 44 52 00 00 00 a0 00   ·······I HDR·····
0050  00 00 64 08 02 00 00 00  8e d4 ac d8 00 00 00 09   ··d····· ········
0060  70 48 59 73 00 00 00 09  00 00 00 0a 00 40 82 9f   pHYs···· ·····@··
0070  70 00 00 00 04 63 49 43  50 01 01 00 01 95 73 c2   p····cIC P·····s·
0080  56 00 00 00 20 63 48 52  4d 00 00 7a 26 00 00 80   V··· cHR M··z&···
0090  84 00 00 fa 00 00 00 80  e8 00 00 75 30 00 00 ea   ········ ···u0···
00a0  60 00 00 3a 98 00 00 17  70 9c ba 51 3c 00 00 00   `··:···· p··Q<···
00b0  04 67 41 4d 41 00 00 c7  32 c7 49 70 44 00 00 00   ·gAMA··· 2·IpD···
00c0  fc 49 44 41 54 78 9c ed  d1 01 09 00 30 0c c0 b0   ·IDATx·· ····0···
00d0  f9 37 bd a9 38 87 92 28  28 74 96 b4 f9 1d c0 5b   ·7··8··( (t·····[
00e0  06 c7 19 1c 67 70 9c c1  71 06 c7 19 1c 67 70 9c   ····gp·· q····gp·
00f0  c1 71 06 c7 19 1c 67 70  9c c1 71 06 c7 19 1c 67   ·q····gp ··q····g
0100  70 9c c1 71 06 c7 19 1c  67 70 9c c1 71 06 c7 19   p··q···· gp··q···
0110  1c 67 70 9c c1 71 06 c7  19 1c 67 70 9c c1 71 06   ·gp··q·· ··gp··q·
0120  c7 19 1c 67 70 9c c1 71  06 c7 19 1c 67 70 9c c1   ···gp··q ····gp··
0130  71 06 c7 19 1c 67 70 9c  c1 71 06 c7 19 1c 67 70   q····gp· ·q····gp
0140  9c c1 71 06 c7 19 1c 67  70 9c c1 71 06 c7 19 1c   ··q····g p··q····
0150  67 70 9c c1 71 06 c7 19  1c 67 70 9c c1 71 06 c7   gp··q··· ·gp··q··
0160  19 1c 67 70 9c c1 71 06  c7 19 1c 67 70 9c c1 71   ··gp··q· ···gp··q

Data (data.data), 423 bytes        Packets: 85 · Displayed: 85 (100.0%)        Profile: Default
```

Figure 10: Producer publishing video frame to broker in Wireshark. Wireshark view of packet headers and their respective payloads, giving a clearer understanding of the protocol's workings.



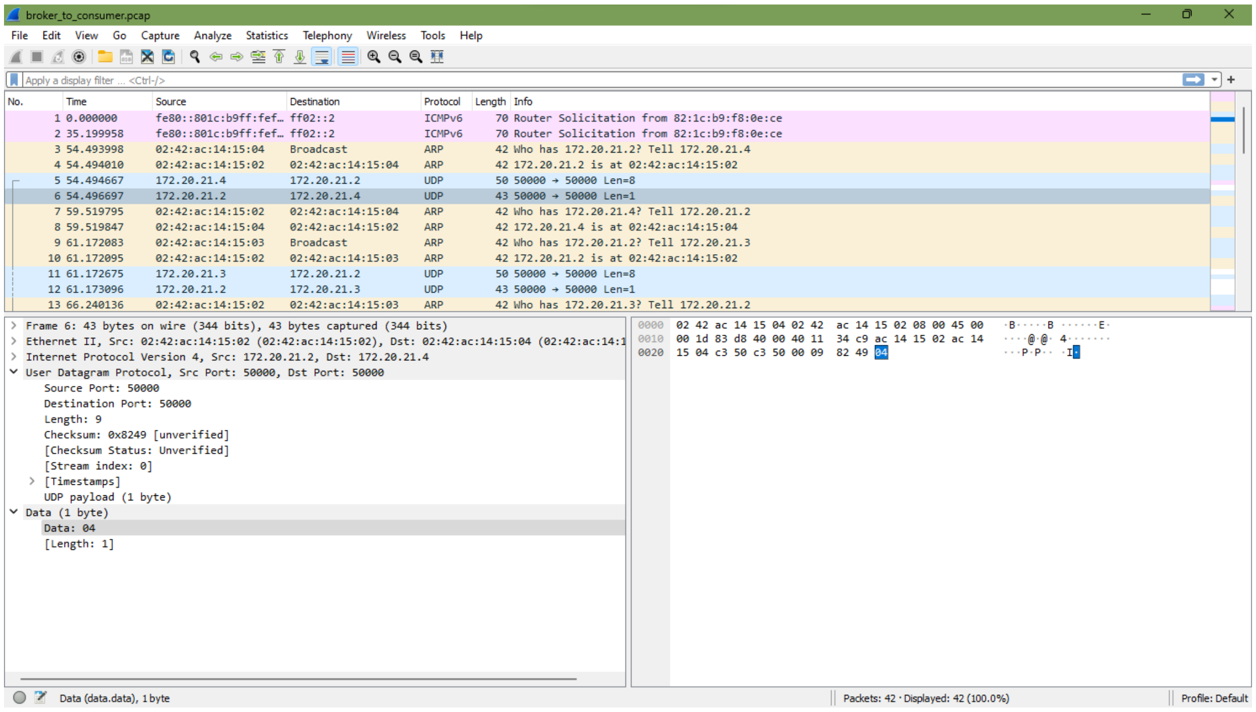
```
producer_to_broker.pcap

No.  Time       Source               Destination          Protocol Length Info
   1 0.000000   fe80::2800:d8ff:fec… ff02::2              ICMPv6     70 Router Solicitation from 2a:00:d8:c9:72:19
   2 33.280052  fe80::2800:d8ff:fec… ff02::2              ICMPv6     70 Router Solicitation from 2a:00:d8:c9:72:19
   3 69.150717  02:42:ac:14:14:03    Broadcast            ARP        42 Who has 172.20.20.2? Tell 172.20.20.3
   4 69.150725  02:42:ac:14:14:02    02:42:ac:14:14:03    ARP        42 172.20.20.2 is at 02:42:ac:14:14:02
   5 69.151170  172.20.20.3          172.20.20.2          UDP       465 35663 → 50000 Len=423
   6 69.152064  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
   7 74.159838  02:42:ac:14:14:02    02:42:ac:14:14:03    ARP        42 Who has 172.20.20.3? Tell 172.20.20.2
   8 74.159893  02:42:ac:14:14:03    02:42:ac:14:14:02    ARP        42 172.20.20.3 is at 02:42:ac:14:14:03
   9 76.565665  172.20.20.3          172.20.20.2          UDP       235 35663 → 50000 Len=193
  10 76.566525  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
  11 81.246396  172.20.20.3          172.20.20.2          UDP        66 35663 → 50000 Len=24
  12 81.247279  172.20.20.2          172.20.20.3          UDP        43 50000 → 35663 Len=1
  13 94.558001  02:42:ac:14:14:04    Broadcast            ARP        42 Who has 172.20.20.2? Tell 172.20.20.4
  14 94.558013  02:42:ac:14:14:02    02:42:ac:14:14:04    ARP        42 172.20.20.2 is at 02:42:ac:14:14:02
  15 94.558745  172.20.20.4          172.20.20.2          UDP        69 60762 → 50000 Len=27
  16 94.560109  172.20.20.2          172.20.20.4          UDP        43 50000 → 60762 Len=1
  17 96.752647  172.20.20.4          172.20.20.2          UDP       465 60762 → 50000 Len=423
  18 96.752903  172.20.20.2          172.20.20.4          UDP        43 50000 → 60762 Len=1

> Frame 6: 43 bytes on wire (344 bits), 43 bytes captured (344 bits)
> Ethernet II, Src: 02:42:ac:14:14:02 (02:42:ac:14:14:02), Dst: 02:42:ac:14:14:03 (02:42:ac:14:1
> Internet Protocol Version 4, Src: 172.20.20.2, Dst: 172.20.20.3
v User Datagram Protocol, Src Port: 50000, Dst Port: 35663
    Source Port: 50000
    Destination Port: 35663
    Length: 9
    Checksum: 0x8048 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  v [Timestamps]
      [Time since first frame: 0.000894000 seconds]
      [Time since previous frame: 0.000894000 seconds]
    UDP payload (1 byte)
v Data (1 byte)
    Data: 02
    [Length: 1]

0000  02 42 ac 14 14 03 02 42  ac 14 14 02 08 00 45 00   ·B·····B ······E·
0010  00 1d c1 66 40 00 40 11  f9 3b ac 14 14 02 ac 14   ···f@·@· ·;······
0020  14 03 c3 50 8b 4f 00 09  80 48 02                  ···P·O·· ·H·

Data (data.data), 1 byte        Packets: 85 · Displayed: 85 (100.0%)        Profile: Default
```

Figure 11: Publish acknowledgement sent from Broker to Producer in Wireshark. Wireshark view of packet headers and their respective payloads, giving a clearer understanding of the protocol's workings



```
broker_to_consumer.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter ... <Ctrl-/>

No.  Time        Source               Destination        Protocol  Length  Info
  1  0.000000    fe80::801c:b9ff:fef… ff02::2            ICMPv6      70  Router Solicitation from 82:1c:b9:f8:0e:ce
  2  35.199958   fe80::801c:b9ff:fef… ff02::2            ICMPv6      70  Router Solicitation from 82:1c:b9:f8:0e:ce
  3  54.493998   02:42:ac:14:15:04    Broadcast          ARP         42  Who has 172.20.21.2? Tell 172.20.21.4
  4  54.494010   02:42:ac:14:15:02    02:42:ac:14:15:04  ARP         42  172.20.21.2 is at 02:42:ac:14:15:02
  5  54.494667   172.20.21.4          172.20.21.2        UDP         50  50000 → 50000 Len=8
  6  54.496697   172.20.21.2          172.20.21.4        UDP         43  50000 → 50000 Len=1
  7  59.519795   02:42:ac:14:15:02    02:42:ac:14:15:04  ARP         42  Who has 172.20.21.4? Tell 172.20.21.2
  8  59.519847   02:42:ac:14:15:04    02:42:ac:14:15:02  ARP         42  172.20.21.4 is at 02:42:ac:14:15:04
  9  61.172083   02:42:ac:14:15:03    Broadcast          ARP         42  Who has 172.20.21.2? Tell 172.20.21.3
 10  61.172095   02:42:ac:14:15:02    02:42:ac:14:15:03  ARP         42  172.20.21.2 is at 02:42:ac:14:15:02
 11  61.172675   172.20.21.3          172.20.21.2        UDP         50  50000 → 50000 Len=8
 12  61.173096   172.20.21.2          172.20.21.3        UDP         43  50000 → 50000 Len=1
 13  66.240136   02:42:ac:14:15:02    02:42:ac:14:15:03  ARP         42  Who has 172.20.21.3? Tell 172.20.21.2

> Frame 5: 50 bytes on wire (400 bits), 50 bytes captured (400 bits)
> Ethernet II, Src: 02:42:ac:14:15:04 (02:42:ac:14:15:04), Dst: 02:42:ac:14:15:02 (02:42:ac:14:1
> Internet Protocol Version 4, Src: 172.20.21.4, Dst: 172.20.21.2
v User Datagram Protocol, Src Port: 50000, Dst Port: 50000
    Source Port: 50000
    Destination Port: 50000
    Length: 16
    Checksum: 0x8250 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  > [Timestamps]
    UDP payload (8 bytes)
v Data (8 bytes)
    Data: 03414243444546ff
    [Length: 8]

0000  02 42 ac 14 15 02 02 42  ac 14 15 04 08 00 45 00   ·B····B ······E·
0010  00 24 f3 d3 40 00 40 11  c4 c6 ac 14 15 04 ac 14   ·$··@·@· ········
0020  15 02 c3 50 c3 50 00 10  82 50 03 41 42 43 44 45   ···P·P·· ·P·ABCDE
0030  46 ff                                              F·

Data (data.data), 8 bytes        Packets: 42 · Displayed: 42 (100.0%)        Profile: Default
```

Figure 12: Consumer subscribing to all streams of producer with 3-byte ID ABCDEF in Wireshark. Wireshark view of packet headers and their respective payloads, giving a clearer understanding of the protocol's workings



```
broker_to_consumer.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter ... <Ctrl-/>

No.  Time        Source               Destination        Protocol  Length  Info
  1  0.000000    fe80::801c:b9ff:fef… ff02::2            ICMPv6      70  Router Solicitation from 82:1c:b9:f8:0e:ce
  2  35.199958   fe80::801c:b9ff:fef… ff02::2            ICMPv6      70  Router Solicitation from 82:1c:b9:f8:0e:ce
  3  54.493998   02:42:ac:14:15:04    Broadcast          ARP         42  Who has 172.20.21.2? Tell 172.20.21.4
  4  54.494010   02:42:ac:14:15:02    02:42:ac:14:15:04  ARP         42  172.20.21.2 is at 02:42:ac:14:15:02
  5  54.494667   172.20.21.4          172.20.21.2        UDP         50  50000 → 50000 Len=8
  6  54.496697   172.20.21.2          172.20.21.4        UDP         43  50000 → 50000 Len=1
  7  59.519795   02:42:ac:14:15:02    02:42:ac:14:15:04  ARP         42  Who has 172.20.21.4? Tell 172.20.21.2
  8  59.519847   02:42:ac:14:15:04    02:42:ac:14:15:02  ARP         42  172.20.21.4 is at 02:42:ac:14:15:04
  9  61.172083   02:42:ac:14:15:03    Broadcast          ARP         42  Who has 172.20.21.2? Tell 172.20.21.3
 10  61.172095   02:42:ac:14:15:02    02:42:ac:14:15:03  ARP         42  172.20.21.2 is at 02:42:ac:14:15:02
 11  61.172675   172.20.21.3          172.20.21.2        UDP         50  50000 → 50000 Len=8
 12  61.173096   172.20.21.2          172.20.21.3        UDP         43  50000 → 50000 Len=1
 13  66.240136   02:42:ac:14:15:02    02:42:ac:14:15:03  ARP         42  Who has 172.20.21.3? Tell 172.20.21.2

> Frame 6: 43 bytes on wire (344 bits), 43 bytes captured (344 bits)
> Ethernet II, Src: 02:42:ac:14:15:02 (02:42:ac:14:15:02), Dst: 02:42:ac:14:15:04 (02:42:ac:14:1
> Internet Protocol Version 4, Src: 172.20.21.2, Dst: 172.20.21.4
v User Datagram Protocol, Src Port: 50000, Dst Port: 50000
    Source Port: 50000
    Destination Port: 50000
    Length: 9
    Checksum: 0x8249 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  > [Timestamps]
    UDP payload (1 byte)
v Data (1 byte)
    Data: 04
    [Length: 1]

0000  02 42 ac 14 15 04 02 42  ac 14 15 02 08 00 45 00   ·B····B ······E·
0010  00 1d 83 d8 40 00 40 11  34 c9 ac 14 15 02 ac 14   ····@·@· 4·······
0020  15 04 c3 50 c3 50 00 09  82 49 04                  ···P·P·· ·I·

Data (data.data), 1 byte        Packets: 42 · Displayed: 42 (100.0%)        Profile: Default
```

Figure 13: Subscription acknowledgement sent from Broker to Consumer in Wireshark. Wireshark view of packet headers and their respective payloads, giving a clearer understanding of the protocol's workings

While the protocol achieves its primary goals as set out in the assignment, there are areas of potential improvement, especially when benchmarked against professional solutions like SRT. Future enhancements could involve incorporating security measures, refining the header design, or considering the transition to a more reliable transport protocol while retaining the publish-subscribe architecture.

7 Summary

In this assignment, I designed and implemented a publish/subscribe protocol for distributing audio, video, and text content using UDP datagrams. Drawing inspiration from the Secure Reliable Transport (SRT) protocol and taking into consideration the real-life scenario of streamers, the focus was to ensure efficient communication among multiple producers, subscribers, and a broker.

To accomplish this, my protocol is constructed around the following components:

- **Producer:** Responsible for generating and forwarding frames of varied content types, such as video, audio, and text, to the broker. Each producer has a unique identification, and each stream they produce is uniquely numbered.

- **Broker:** Acts as the intermediary that receives content from producers and directs them to appropriate subscribers. The broker manages subscriptions and keeps track of which subscribers are interested in which streams.

- **Consumer:** Subscribes to specific streams or to all streams from a given producer. They rely on the broker to receive the relevant content based on their subscription.

The key component of my solution was the header design, which is vital for effective protocol communication. The header includes information such as the producer's ID, stream number, frame number, length of payload, and content type. These details enable the broker to accurately forward content to the respective subscribers.

For video content, individual frames were read from pre-existing samples. For audio, segments of an audio file were captured and sent, ensuring each segment was of a uniform duration. Text content was dynamically input by the user and then relayed.

Throughout the assignment, emphasis was placed on ensuring that the protocol could handle out-of-order or missing frames. Given the nature of streaming, where occasional missing frames might not greatly disrupt the experience, the real challenge lies in preventing older frames from being displayed after newer ones, which would result in an inconsistent viewing experience.

On the implementation side, Python was chosen for its ease of use and support for networking functionalities. Using UDP sockets, each component of the protocol communicated efficiently. Both `sendto` and `recvfrom` functions facilitated this communication, allowing for sending data to and receiving data from the intended addresses.

To summarise, this project served as an immersive experience in protocol design and communication, challenging me to devise a system that can handle real-time content distribution efficiently. The end solution is a robust and functional protocol that manages subscriptions, publishes content, and distributes it, ensuring the right content reaches the right subscribers.

8 Reflection

During the course of this assignment, I encountered a spectrum of challenges and learning opportunities that honed my understanding of protocol development and the intricate balance between functionality and overhead.

Time Commitment: Initially, I anticipated this project to take a limited amount of hours. However, as I dove into the intricacies of the assignment, especially the UDP datagram and publish-subscribe mechanism, I found myself investing significant time. Cumulatively, I spent approximately 30 hours working on the assignment, divided amongst designing, coding, debugging, and testing.

Strengths:

1. **Producer Implementation:** My producer logic, particularly for video and audio streams, worked flawlessly. Leveraging the pydub and PIL libraries for handling audio and image content respectively streamlined the process.
2. **Designing Headers:** The binary encoding of header information was one of the more intuitive aspects. With a clear understanding from the assignment description, I was able to develop a structured and coherent header design, allowing the identification and management of subscriptions and publications.
3. **Use of UDP:** Working with UDP datagrams was initially daunting, but as I became more accustomed to it, I felt comfortable and appreciated its lightweight nature, especially for streaming purposes.

Challenges:

1. **Handling Out-of-Order Frames:** The nature of UDP means frames can arrive out of order or be lost altogether. Ensuring that older frames aren't displayed after newer ones proved to be tricky.
2. **Broker Logic:** While the producer implementation went smoothly, the broker logic presented challenges. Maintaining and managing the list of subscribers while also efficiently forwarding the content took multiple iterations to perfect.
3. **Binary Encoding:** While I managed to implement binary encoding effectively, I did find myself occasionally tripped up by byte order and conversions, especially during debugging.

Areas of Improvement:

1. **Scalability:** My current approach is functional but could be optimised further for scalability, especially when considering a high number of subscribers and producers.
2. **Error Handling:** While I've implemented basic error checks, a more robust error-handling mechanism would make the protocol more resilient.
3. **Feedback Mechanism:** I realised that some feedback mechanism for lost frames or delays, perhaps something akin to SRT's mechanism, could improve the streaming experience.

Final Thoughts:
This assignment was undoubtedly challenging, yet immensely rewarding. It was a true deep dive into the nuances of protocol development, allowing me to appreciate the complexity behind seemingly simple streaming services. While I'm proud of my protocol's current state, I acknowledge that there's always room for improvement. As for future assignments, I plan to start even earlier, allocate more time for testing, and possibly collaborate more with peers for brainstorming and feedback.