

CO513: Advanced Computer Communication Networks - Lab 05

E/19/446: Wijerathna I.M.K.D.I.

02/06/2025

Lab Tasks

Part 1: Build the Video Streaming System

Task 1: Server Setup

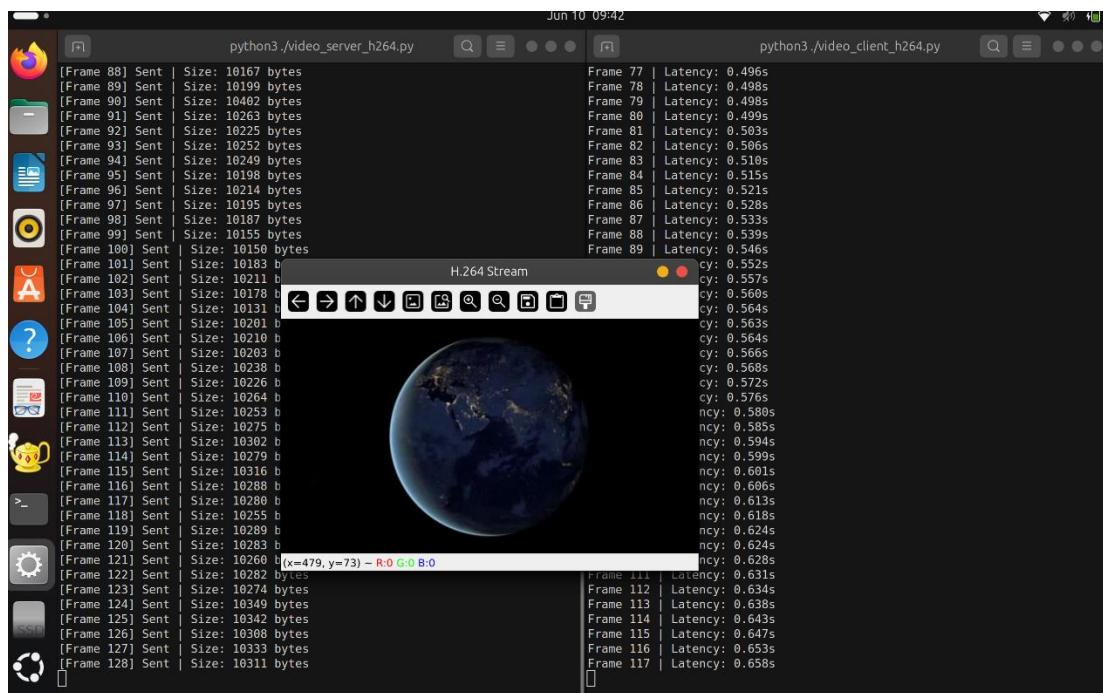
This was implemented using the python script “./video_server_h264.py” which captures video frames from a given mp4 file using OpenCV, then encodes them to H.264 format with FFmpeg, and streams the encoded frames over a TCP socket to a connected client. It sends each frame's ID, timestamp, and size along with the encoded byte data, handling connections until the video ends or the connection is lost. (all the used packages are in the ./README.md)

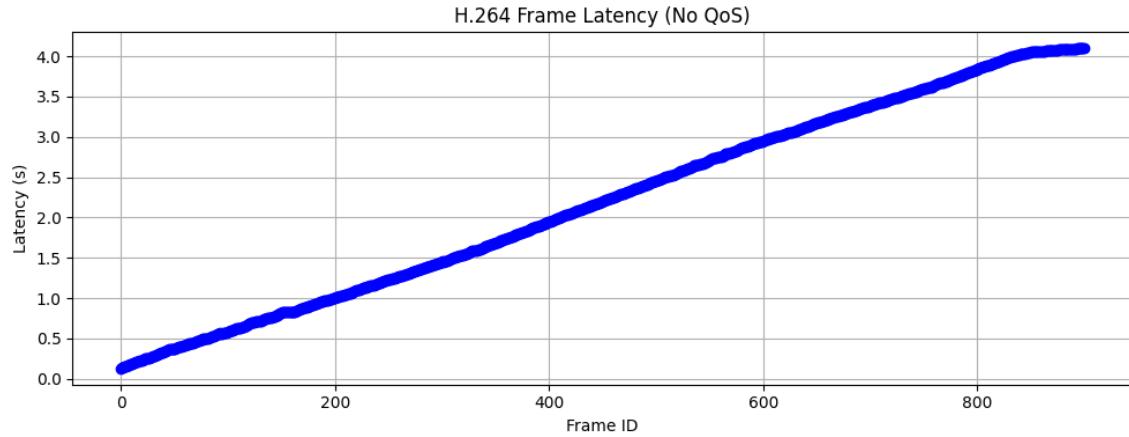
Task 2: Client Setup

The client script (“./video_client_h264.py”) connects to the video server over the TCP socket, receives and decodes H.264 video frames, and then displays them (live playback) using OpenCV. It tracks frame IDs to detect losses, measures latency for each frame, and saves latency and summary statistics to CSV files, along with generating plots for latency and frame loss.

Task 3: Initial Testing

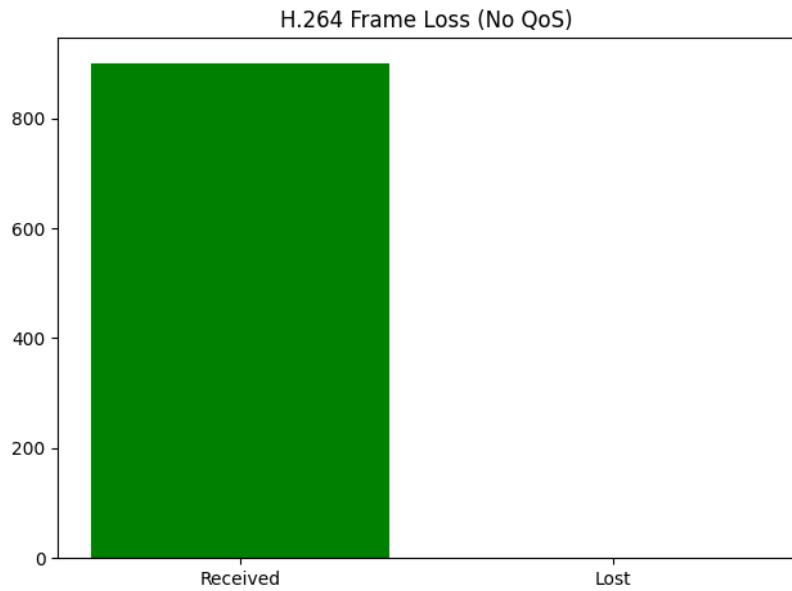
Following is a screenshot taken at the initial testing.





As it can be observed from the above graph, frame latency for this ‘without QoS implementation’ has increased steadily with frame ID, indicating a consistent accumulation of delay as frames were processed. This linear trend suggests potential bottlenecks in the encoding or transmission pipeline, leading to significant end-to-end delays, reaching around 4 seconds for later frames.

On the other hand, as the below graph shows, the lost frame count is 0 for this ‘without QoS implementation’ which possibly could not be achieved if we have used a QoS mechanism.



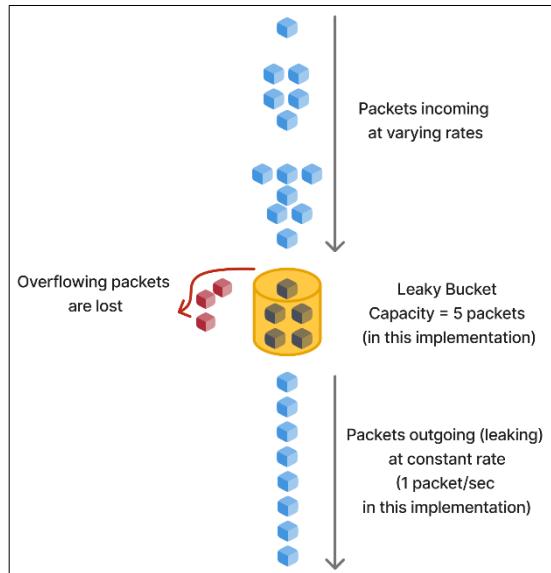
Considering all the above observations we can conclude that, a QoS mechanism is essential to avoid potential bottlenecks in the video stream, but it may also lead to frame/packet drops.

Part 2: Implement QoS Algorithms

Task 4: Leaky Bucket Algorithm

- #### 1. Leaky Bucket Server Logic (“./leaky_bucket_server.py”):

Implemented a simple server that manages packet processing using a bucket with a fixed size. It accepts incoming packets, adds them to the bucket if there's space, and processes them at a defined leak rate, while tracking the number of processed and dropped packets due to overflow. (Implemented algorithm is as follows)



- ## 2. Leaky Bucket Client Logic (“./leaky_bucket_client.py”):

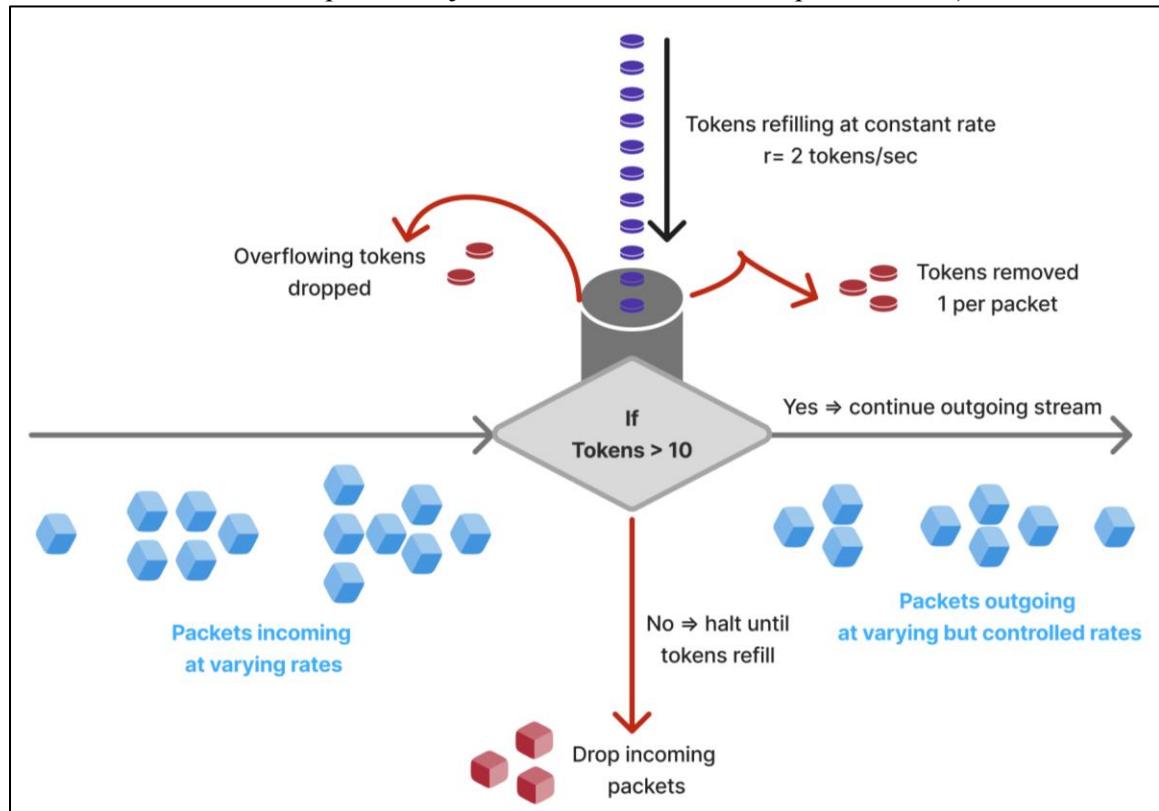
This connects to the server and sends a series of randomly sized packets (between 500 and 2000 bytes) in a loop. After sending each packet, it waits for the server's response and pauses for a random interval between 10ms and 500ms before sending the next packet, allowing for realistic and controlled transmission rates.

Following is a screenshot taken while the leaky bucket algorithm was running.

Task 5: Token Bucket Algorithm

1. Token Bucket Server Logic (“./token_bucket_server.py”):

This server script implements a token bucket rate-limiting mechanism to control packet processing. It refills tokens at a specified rate and processes incoming packets only if sufficient tokens are available, tracking the number of processed and rejected packets due to insufficient tokens. (Implemented algorithm is as follows. Parameters like 2 tokens/sec, 10 tokens, removal at 1 token/packet are just chosen values for this implementation.)



2. Token Bucket Client Logic (“./token_bucket_client.py”):

This client script connects to a server and simulates sending a series of packets with randomly generated payload sizes (between 500 and 2000 bytes). After each packet is sent, it waits for the server's response and introduces a random delay (between 10ms and 500ms) to mimic bursty traffic patterns.

The following is a screenshot taken while running the token bucket algorithm. (PTO)

```

[~/C0513/Lab-5/LabTasks]
kalindu$ python3 ./token_bucket_server.py
[Server] Listening on localhost:9997...
[+] Connection from ('127.0.0.1', 53918)
[✓] Packet processed | Tokens left: 9
[✓] Packet processed | Tokens left: 8
[+] Token added | Tokens now: 8
[✓] Packet processed | Tokens left: 7
[+] Token added | Tokens now: 8
[✓] Packet processed | Tokens left: 7
[+] Token added | Tokens now: 8
[✓] Packet processed | Tokens left: 7
[+] Token added | Tokens now: 8
[✓] Packet processed | Tokens left: 6
[+] Token added | Tokens now: 7
[✓] Packet processed | Tokens left: 6
[+] Token added | Tokens now: 7
[✓] Packet processed | Tokens left: 6
[+] Token added | Tokens now: 6
[✓] Packet processed | Tokens left: 5
[+] Token added | Tokens now: 6
[✓] Packet processed | Tokens left: 5
[+] Token added | Tokens now: 5
[✓] Packet processed | Tokens left: 4
[+] Token added | Tokens now: 5
[✓] Packet processed | Tokens left: 4
[✓] Packet processed | Tokens left: 3
[✓] Packet processed | Tokens left: 2
[+] Token added | Tokens now: 3
[✓] Packet processed | Tokens left: 2
[✓] Packet processed | Tokens left: 1
[+] Token processed | Tokens left: 0
[+] Token added | Tokens now: 1
[✓] Packet processed | Tokens left: 0
[+] Token added | Tokens now: 1
[Summary] Processed: 20, Rejected: 0

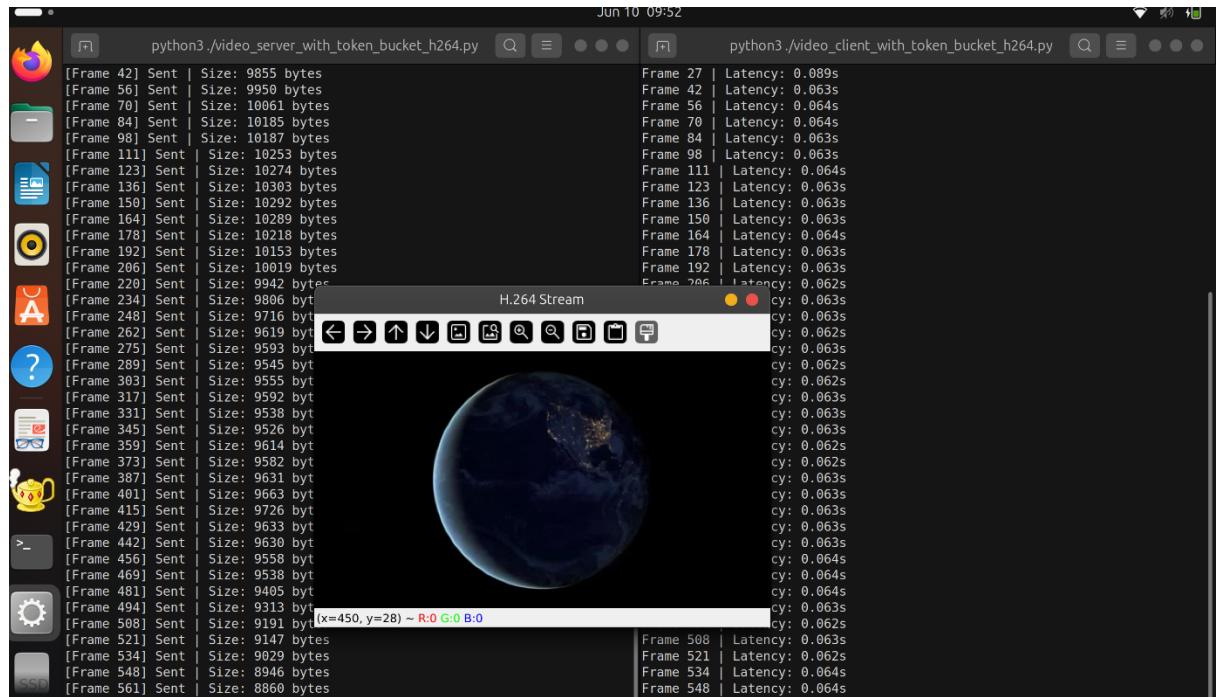
[~/C0513/Lab-5/LabTasks]
kalindu$ python3 ./token_bucket_client.py
[Client] Sent 1162 bytes | Server: processed
[Client] Sent 1768 bytes | Server: processed
[Client] Sent 1717 bytes | Server: processed
[Client] Sent 1238 bytes | Server: processed
[Client] Sent 1876 bytes | Server: processed
[Client] Sent 1230 bytes | Server: processed
[Client] Sent 1885 bytes | Server: processed
[Client] Sent 754 bytes | Server: processed
[Client] Sent 738 bytes | Server: processed
[Client] Sent 1037 bytes | Server: processed
[Client] Sent 1484 bytes | Server: processed
[Client] Sent 613 bytes | Server: processed
[Client] Sent 1743 bytes | Server: processed
[Client] Sent 1107 bytes | Server: processed
[Client] Sent 859 bytes | Server: processed
[Client] Sent 1101 bytes | Server: processed
[Client] Sent 1458 bytes | Server: processed
[Client] Sent 1012 bytes | Server: processed
[Client] Sent 643 bytes | Server: processed
[Client] Sent 1343 bytes | Server: processed

```

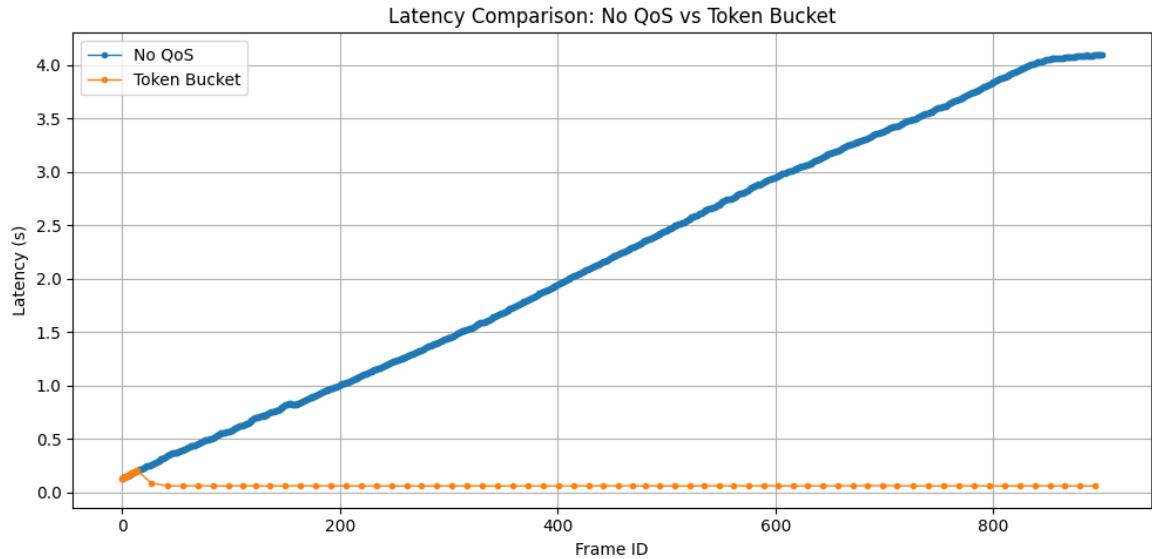
Part 3: QoS Integration for Streaming

Task 6: Integrate QoS into Streaming System

Following is a screenshot taken while running the integrated video server with token bucket algorithm. (Token Bucket Algorithm was chosen for the integration)

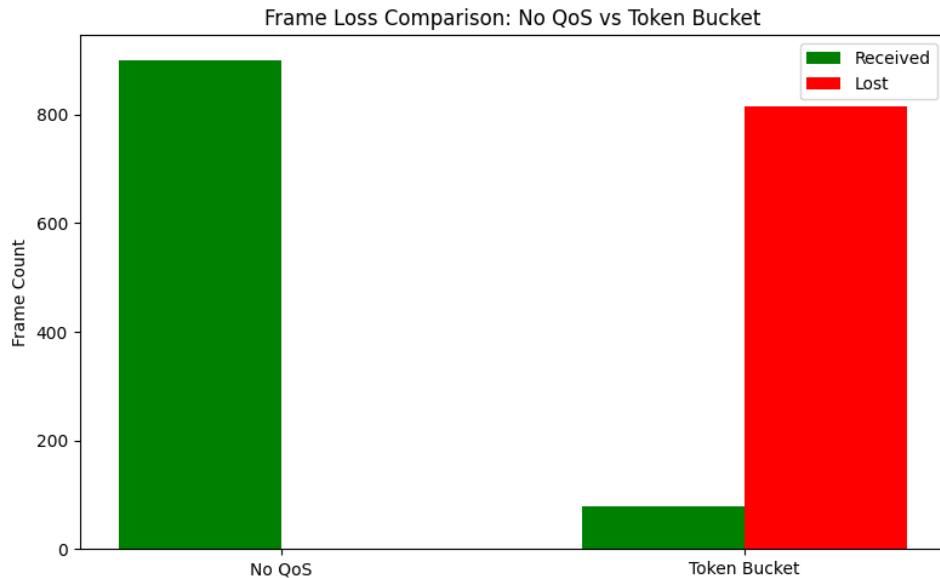


Testing and comparing with the no-qos implementation



It can be observed from the above comparison graph, that the token bucket qos implementation has maintained a much lower and more stable latency throughout the frame flow avoiding possible bottlenecks. This suggests that the Token Bucket algorithm effectively manages traffic and reduces latency.

However, as expected, a considerable number of packets have been lost due to the QoS mechanism as shown below.

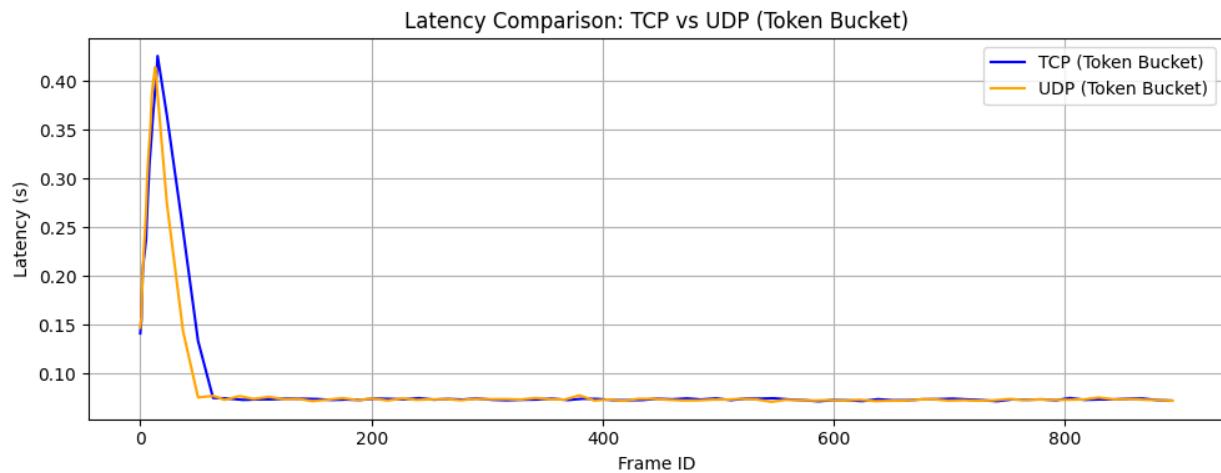


The above lost frame ratio can be lowered, if the refill rate, token bucket capacity, and packets dropped rate per token are adjusted accordingly.

Experimentation & Analysis

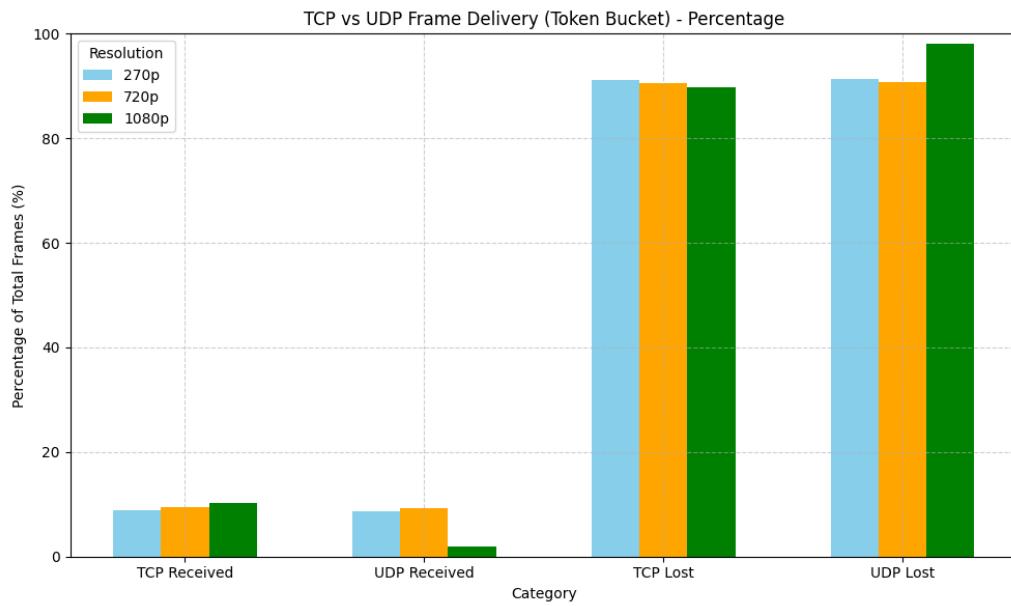
1. Protocol Behavior (TCP vs. UDP)

Similar to the previous TCP setup, I created video server-client scripts to use UDP with H.264 encoding. Below are the comparison results.



As it can be observed from the above graph, UDP shows slightly less latency for the video traffic. The gap between the latencies of the two protocols would grow larger if the frame rate, video quality are increased. This indicates UDP's suitability for real time traffic over TCP. (like Live streaming, VOIP)

Even though, UDP reduces latency it increases the number of lost frames/packets making it unreliable over TCP. The following chart depicts that when the quality of the video stream is increased the number of lost packets for UDP have drastically increased while for TCP it remains almost the same. Therefore, when a reliable packet transfer is required, TCP is better than UDP. (suitable for File transfer, email transfer, web browsing)



Following is a snapshot taken while running the UDP server-client setup.

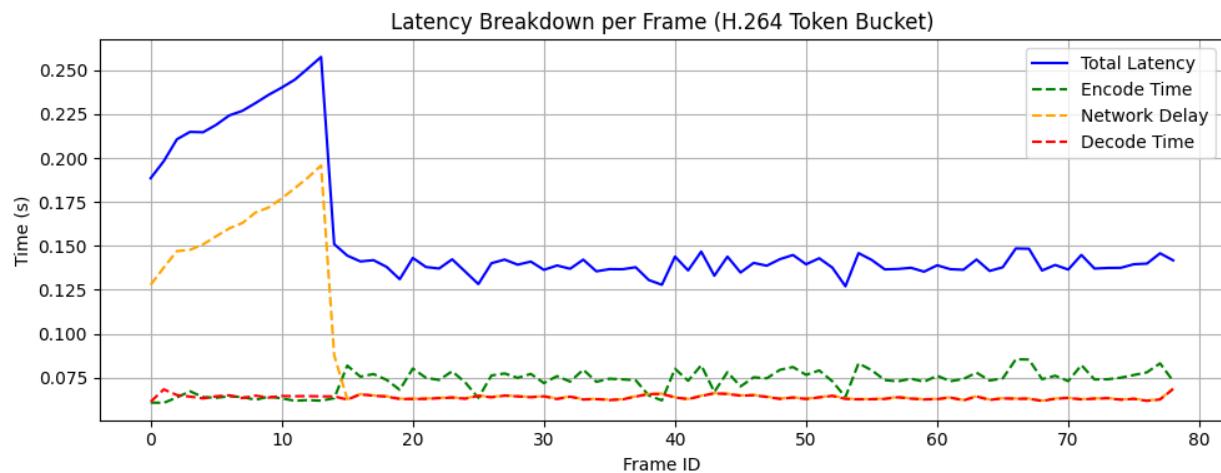
```
[~/C0513/Lab-5/Tasks] kalindu@kalindu: ~$ python3 ./video_server_with_token_bucket_h264_udp.py
[+] UDP Streaming started...
[Frame 0] Sent | Size: 9864 bytes
[Frame 1] Sent | Size: 9786 bytes
[Frame 2] Sent | Size: 9766 bytes
[Frame 3] Sent | Size: 9713 bytes
[Frame 4] Sent | Size: 9736 bytes
[Frame 5] Sent | Size: 9755 bytes
[Frame 6] Sent | Size: 9744 bytes
[Frame 7] Sent | Size: 9739 bytes
[Frame 8] Sent | Size: 9735 byte
[Frame 9] Sent | Size: 9806 byte
[Frame 10] Sent | Size: 9781 byte
[Frame 11] Sent | Size: 9725 bytes
[Frame 12] Sent | Size: 9757 byte
[Frame 13] Sent | Size: 9735 byte
[Frame 25] Sent | Size: 9777 byte
[Frame 40] Sent | Size: 9861 byte
[Frame 54] Sent | Size: 9925 byte
[Frame 67] Sent | Size: 10084 byte
[Frame 80] Sent | Size: 10191 byte
[Frame 94] Sent | Size: 10249 byte
[Frame 108] Sent | Size: 10238 byte
[Frame 122] Sent | Size: 10282 byte
[Frame 136] Sent | Size: 10303 byte
[Frame 150] Sent | Size: 10292 byte
[Frame 164] Sent | Size: 10289 byte

Jun 10 10:33 [~/C0513/Lab-5/Tasks] kalindu@kalindu: ~$ python3 ./video_client_with_token_bucket_h264_udp.py
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
Frame 0 | Latency: 0.126s
Frame 1 | Latency: 0.133s
Frame 2 | Latency: 0.144s
Frame 3 | Latency: 0.146s
Frame 4 | Latency: 0.151s
Frame 5 | Latency: 0.154s
Frame 6 | Latency: 0.158s
Frame 7 | Latency: 0.165s
Frame 8 | Latency: 0.171s
Frame 9 | Latency: 0.177s
Frame 10 | Latency: 0.179s
Frame 11 | Latency: 0.185s
Frame 12 | Latency: 0.194s
Frame 13 | Latency: 0.203s
Frame 14 | Latency: 0.095s
Frame 15 | Latency: 0.064s
Frame 16 | Latency: 0.064s
Frame 17 | Latency: 0.064s
Frame 18 | Latency: 0.064s
Frame 19 | Latency: 0.063s
Frame 20 | Latency: 0.064s
Frame 21 | Latency: 0.064s
Frame 22 | Latency: 0.062s
Frame 23 | Latency: 0.062s

H.264 UDP Stream
(x=316, y=11) - R:0 G:0 B:0
```

2. Latency Analysis

For this analysis, I have modified the server-client setup to log various latency metrics during video streaming, capturing total latency, encoding latency, network delay, and decoding latency for each frame. The data is saved to CSV files and visualized in plots to analyze performance and frame loss.



As it can be observed from the above chart, the initial spike in the total latency is due to the network delay while initiating network connection (TCP handshake etc.). After the initial spike, the decode time and

network delay remain consistently low, while encode time also stays manageable, indicating efficient processing. Overall, the results suggest that the Token Bucket approach effectively minimizes latency after an initial adjustment period.

Following output shows that, in detail latency logs are being shared between the server and client.

```

Jun 10 12:48
[~/C0513/Lab-5/Tasks]
kalindu@pingy:~/C0513/Lab-5/Tasks]$ python3 ./video_server_with_token_bucket_h264_indepth_latency.py
[+] Waiting for client...
[+] Client connected.
[Frame 0] Sent | Size: 9864 bytes | Encode Time: 0.0607s
[Frame 1] Sent | Size: 9786 bytes | Encode Time: 0.0606s
[Frame 2] Sent | Size: 9706 bytes | Encode Time: 0.0637s
[Frame 3] Sent | Size: 9713 bytes | Encode Time: 0.0672s
[Frame 4] Sent | Size: 9736 bytes | Encode Time: 0.0638s
[Frame 5] Sent | Size: 9755 bytes | Encode Time: 0.0635s
[Frame 6] Sent | Size: 9744 bytes | Encode Time: 0.0643s
[Frame 7] Sent | Size: 9739 bytes | Encode Time: 0.0639s
[Frame 8] Sent | Size: 9735 bytes | Encode Time: 0.0623s
[Frame 9] Sent | Size: 9806 bytes | Encode Time: 0.0641s
[Frame 10] Sent | Size: 9781 bytes | Encode Time: 0.0631s
[Frame 11] Sent | Size: 9725 bytes | Er
[Frame 12] Sent | Size: 9757 bytes | Er
[Frame 13] Sent | Size: 9735 bytes | Er
[Frame 14] Sent | Size: 9777 bytes | Er
[Frame 15] Sent | Size: 9861 bytes | Er
[Frame 16] Sent | Size: 9919 bytes | Er
[Frame 17] Sent | Size: 10084 bytes | E
[Frame 18] Sent | Size: 10191 bytes | E
[Frame 19] Sent | Size: 10249 bytes | E
[Frame 20] Sent | Size: 10238 bytes | E
[Frame 21] Sent | Size: 10268 bytes | E
[Frame 22] Sent | Size: 10292 bytes | E
[Frame 23] Sent | Size: 10314 bytes | E
[Frame 24] Sent | Size: 10254 bytes | E
H.264 Stream

```

```

python3 ./video_client_with_token_bucket_h264_indepth_latency.py
[~/C0513/Lab-5/Tasks]
kalindu@pingy:~/C0513/Lab-5/Tasks]$ python3 ./video_client_with_token_bucket_h264_indepth_latency.py
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
Frame 0 | Total: 0.188s | Encode: 0.061s | Net: 0.128s | Decode: 0.061s
Frame 1 | Total: 0.198s | Encode: 0.061s | Net: 0.138s | Decode: 0.068s
Frame 2 | Total: 0.211s | Encode: 0.064s | Net: 0.147s | Decode: 0.065s
Frame 3 | Total: 0.215s | Encode: 0.067s | Net: 0.148s | Decode: 0.064s
Frame 4 | Total: 0.215s | Encode: 0.064s | Net: 0.151s | Decode: 0.063s
Frame 5 | Total: 0.219s | Encode: 0.063s | Net: 0.155s | Decode: 0.064s
Frame 6 | Total: 0.224s | Encode: 0.064s | Net: 0.160s | Decode: 0.065s
Frame 7 | Total: 0.227s | Encode: 0.064s | Net: 0.163s | Decode: 0.063s
Frame 8 | Total: 0.231s | Encode: 0.062s | Net: 0.169s | Decode: 0.065s
Frame 9 | Total: 0.235s | Encode: 0.064s | Net: 0.172s | Decode: 0.064s
Frame 10 | Total: 0.238s | Encode: 0.063s | Net: 0.177s | Decode: 0.064s
Frame 11 | Total: 0.241s | Encode: 0.062s | Net: 0.183s | Decode: 0.064s
Frame 12 | Total: 0.244s | Encode: 0.062s | Net: 0.189s | Decode: 0.064s
Frame 13 | Total: 0.247s | Encode: 0.063s | Net: 0.196s | Decode: 0.064s
Frame 14 | Total: 0.250s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 15 | Total: 0.253s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 16 | Total: 0.256s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 17 | Total: 0.259s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 18 | Total: 0.262s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 19 | Total: 0.265s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 20 | Total: 0.268s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 21 | Total: 0.271s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 22 | Total: 0.274s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 23 | Total: 0.277s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 24 | Total: 0.280s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 25 | Total: 0.283s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 26 | Total: 0.286s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s
Frame 27 | Total: 0.289s | Encode: 0.063s | Net: 0.198s | Decode: 0.064s

```

3. Encoding Efficiency

For this part I have used real time video codecs H.264, VP9, H.265 with varying resolutions of 270p, 720p and 1080p while logging latency, CPU usage, and bandwidth.

1. H.264 Codec (same encoding which used throughout from the begining)

```

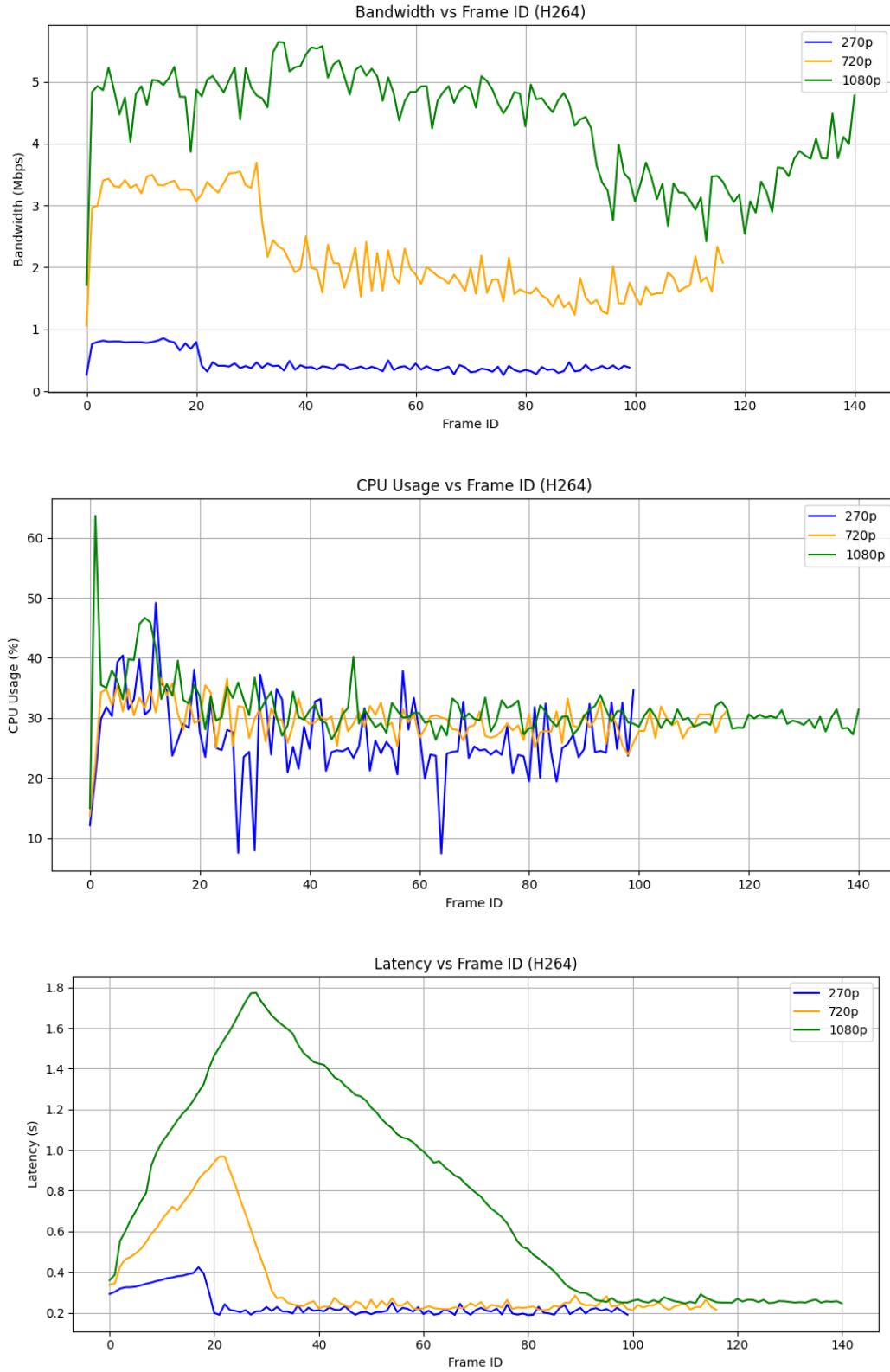
[~/C0513/Lab-5/Tasks]
kalindu@pingy:~/C0513/Lab-5/Tasks]$ python3 ./video_server_with_token_bucket_h264_log_all.py
[+] Waiting for client to connect...
[+] Client connected!
[Frame 0] Sent | Size: 9864 bytes | Encode Time: 0.0942s
[Frame 1] Sent | Size: 9786 bytes | Encode Time: 0.0911s
[Frame 2] Sent | Size: 9768 bytes | Encode Time: 0.0925s
[Frame 3] Sent | Size: 9713 bytes | Encode Time: 0.1012s
[Frame 4] Sent | Size: 9736 bytes | Encode Time: 0.0979s
[Frame 5] Sent | Size: 9755 bytes | Encode Time: 0.0947s
[Frame 6] Sent | Size: 9744 bytes | Encode Time: 0.0947s
[Frame 7] Sent | Size: 9739 bytes | Encode Time: 0.0940s
[Frame 8] Sent | Size: 9735 bytes | Encode Time: 0.0940s
[Frame 9] Sent | Size: 9806 bytes | Encode Time: 0.0940s
[Frame 10] Sent | Size: 9781 bytes | Encode Time: 0.0940s
[Frame 11] Sent | Size: 9725 bytes | Er
[Frame 12] Sent | Size: 9757 bytes | Er
[Frame 13] Sent | Size: 9735 bytes | Er
[Frame 14] Sent | Size: 9747 bytes | Er
[Frame 15] Sent | Size: 9759 bytes | Er
[Frame 16] Sent | Size: 9685 bytes | Er
[Frame 17] Sent | Size: 9682 bytes | Er
[Frame 18] Sent | Size: 9746 bytes | Er
[Frame 19] Sent | Size: 9782 bytes | Er
[Frame 20] Sent | Size: 9889 bytes | Er
[Frame 21] Sent | Size: 9950 bytes | Er
[Frame 22] Sent | Size: 10119 bytes | Er
[Frame 23] Sent | Size: 10110 bytes | Er
[Frame 24] Sent | Size: 10152 bytes | Er
[Frame 25] Sent | Size: 10155 bytes | Er
[Frame 26] Sent | Size: 10264 bytes | Er
[Frame 27] Sent | Size: 10282 bytes | Encode Time: 0.0940s
H.264 Stream

```

```

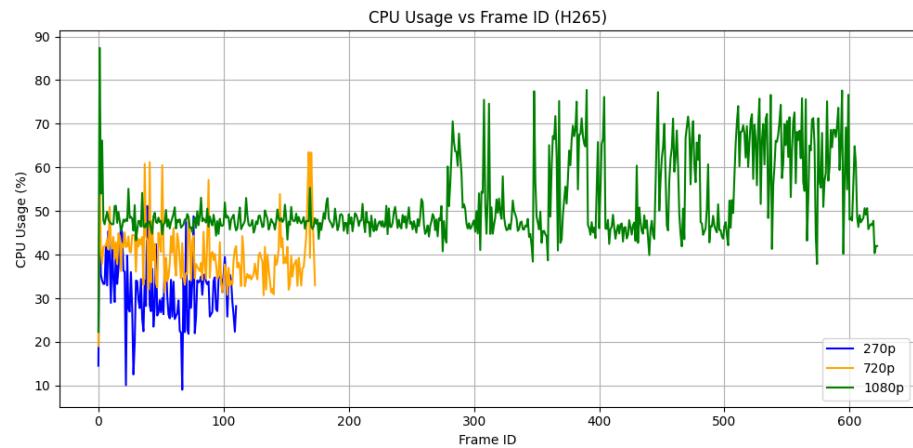
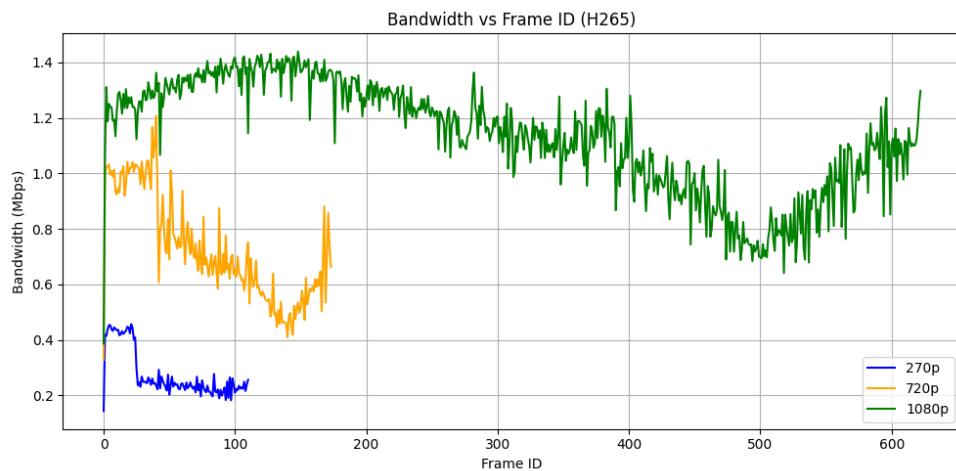
[~/C0513/Lab-5/Tasks]
kalindu@pingy:~/C0513/Lab-5/Tasks]$ python3 ./video_client_with_token_bucket_h264_log_all.py
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
Frame 0 | Latency: 0.292s | CPU: 12.15% | BW: 0.27 Mbps
Frame 1 | Latency: 0.302s | CPU: 20.35% | BW: 0.76 Mbps
Frame 2 | Latency: 0.318s | CPU: 29.80% | BW: 0.80 Mbps
Frame 3 | Latency: 0.325s | CPU: 31.80% | BW: 0.82 Mbps
Frame 4 | Latency: 0.325s | CPU: 30.30% | BW: 0.80 Mbps
Frame 5 | Latency: 0.328s | CPU: 39.30% | BW: 0.88 Mbps
Frame 6 | Latency: 0.335s | CPU: 40.40% | BW: 0.88 Mbps
Frame 7 | Latency: 0.342s | CPU: 31.40% | BW: 0.79 Mbps
Frame 8 | Latency: 0.335s | CPU: 49.15% | BW: 0.80 Mbps
Frame 9 | Latency: 0.348s | CPU: 33.10% | BW: 0.79 Mbps
Frame 10 | Latency: 0.356s | CPU: 39.75% | BW: 0.79 Mbps
Frame 11 | Latency: 0.361s | CPU: 30.55% | BW: 0.79 Mbps
Frame 12 | Latency: 0.376s | CPU: 31.40% | BW: 0.78 Mbps
Frame 13 | Latency: 0.373s | CPU: 49.15% | BW: 0.80 Mbps
Frame 14 | Latency: 0.379s | CPU: 35.80% | BW: 0.82 Mbps
Frame 15 | Latency: 0.382s | CPU: 34.60% | BW: 0.86 Mbps
Frame 16 | Latency: 0.389s | CPU: 23.70% | BW: 0.81 Mbps
Frame 17 | Latency: 0.395s | CPU: 26.25% | BW: 0.79 Mbps
Frame 18 | Latency: 0.423s | CPU: 29.00% | BW: 0.66 Mbps
Frame 19 | Latency: 0.393s | CPU: 28.35% | BW: 0.77 Mbps
Frame 20 | Latency: 0.302s | CPU: 38.05% | BW: 0.68 Mbps
Frame 21 | Latency: 0.199s | CPU: 27.70% | BW: 0.79 Mbps
Frame 22 | Latency: 0.189s | CPU: 23.50% | BW: 0.41 Mbps
Frame 23 | Latency: 0.242s | CPU: 32.98% | BW: 0.32 Mbps
Frame 24 | Latency: 0.213s | CPU: 25.00% | BW: 0.47 Mbps
Frame 25 | Latency: 0.209s | CPU: 24.65% | BW: 0.41 Mbps
Frame 26 | Latency: 0.202s | CPU: 28.00% | BW: 0.41 Mbps
Frame 27 | Latency: 0.213s | CPU: 27.65% | BW: 0.40 Mbps

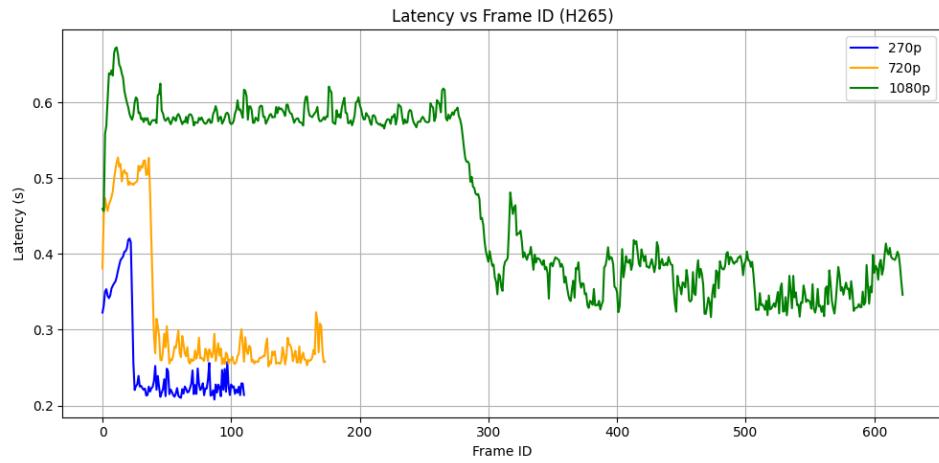
```



It can be observed from the above graphs that when increasing the video quality, H.264 codec's cpu consumption rate stays stable while latency and bandwidth increases.

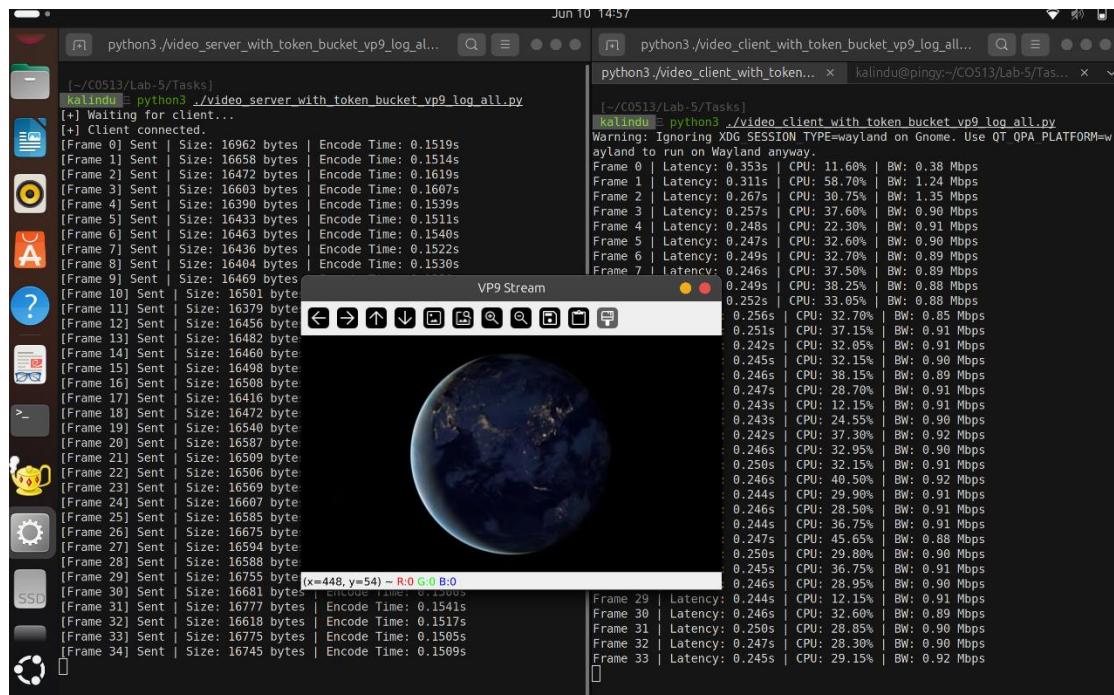
2. H.265 Codec



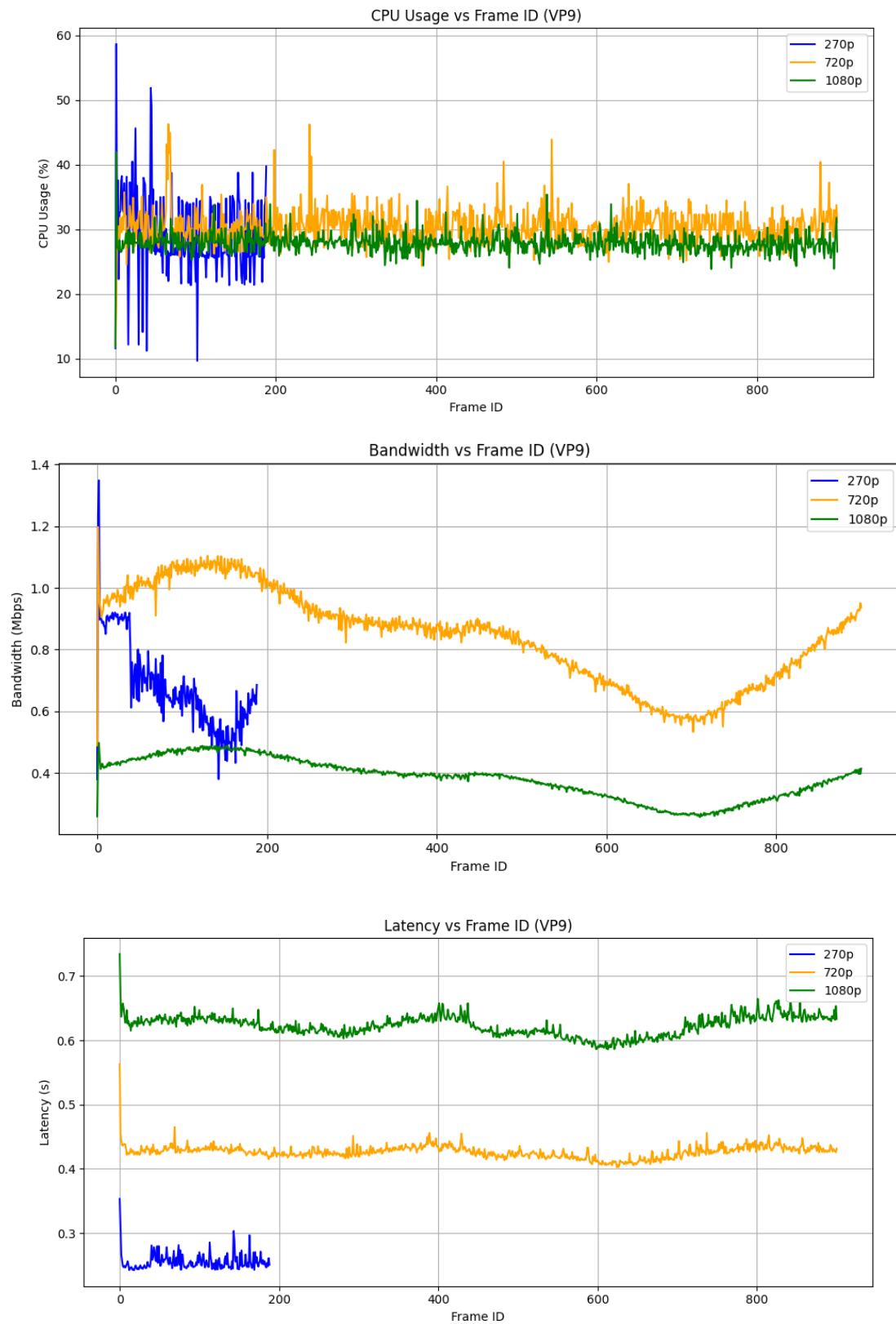


It can be observed from the above graphs that, H.265 codec behaves similar to the H.264 codec when video resolution is increased.

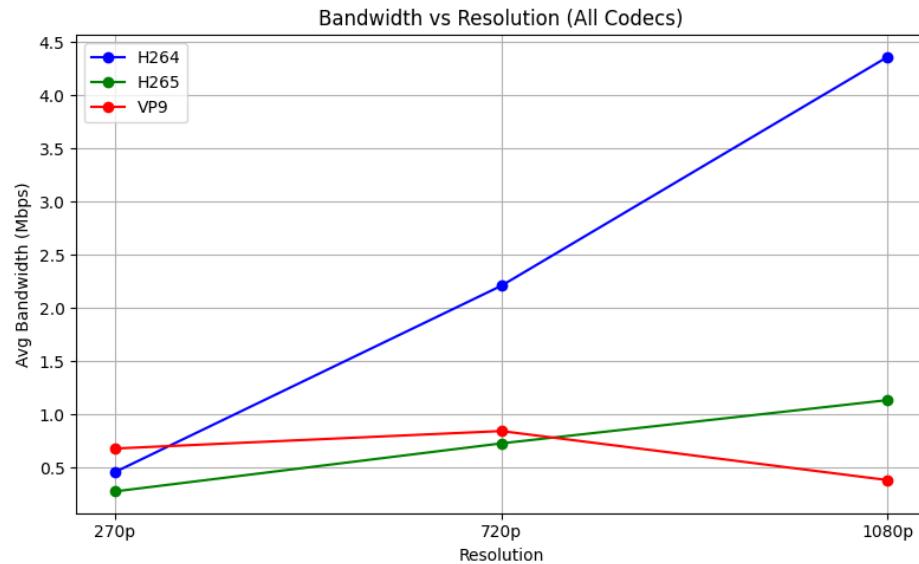
3. VP9 Codec



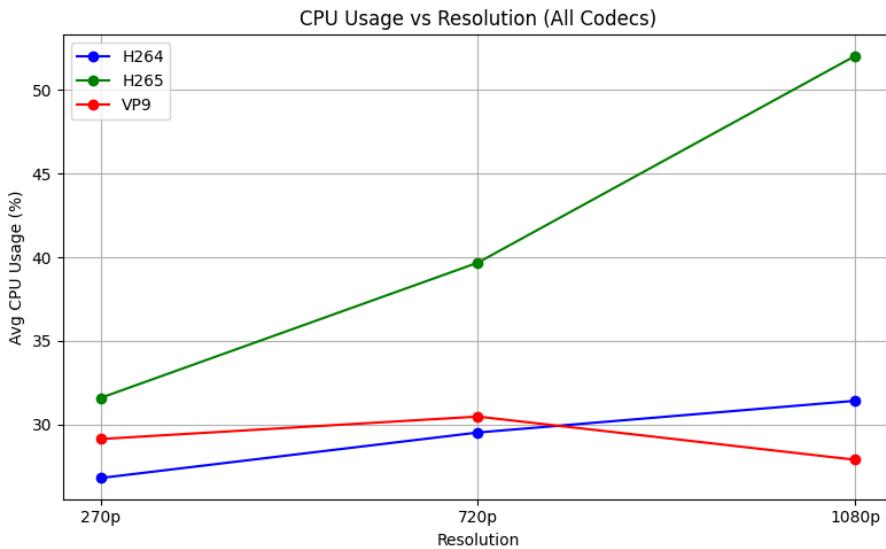
It can be observed from the following graphs that, VP9 codec behaves a bit differently in terms of bandwidth compared to H.264 and H.265 by performing well for mid quality streams.



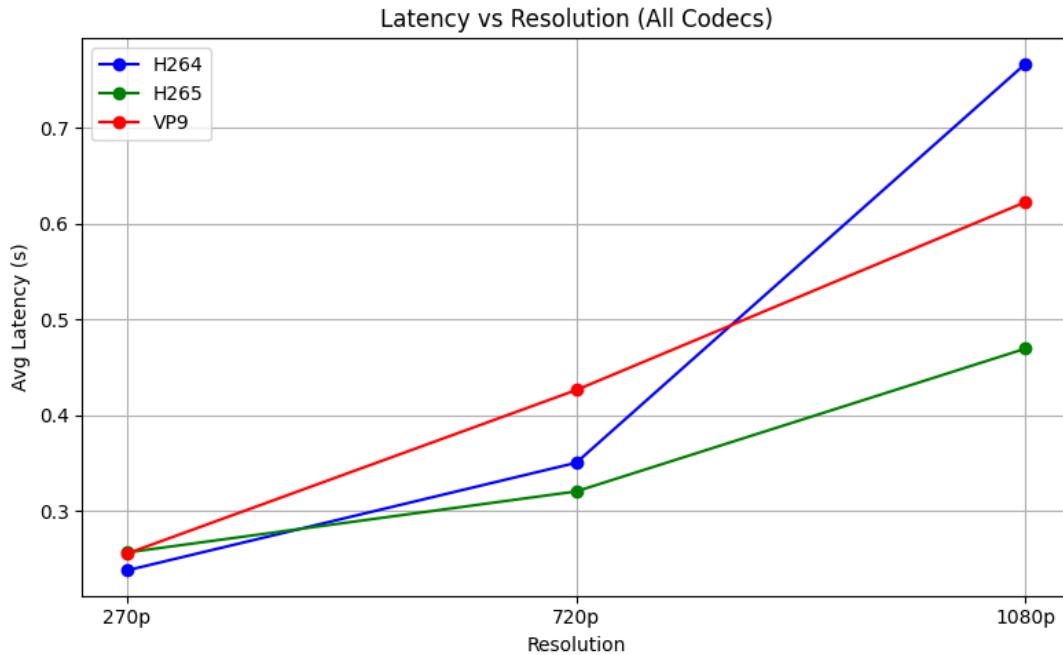
To get an overall idea of how all the three codecs behave with increasing video resolutions, they were plotted using average values of latency, bandwidth, and CPU usage as follows.



The above graph compares average bandwidth usage across different resolutions for H.264, H.265, and VP9 codecs, showing that H.264 requires the most bandwidth, particularly at higher resolutions. H.265 and VP9 demonstrate more efficient compression, resulting in lower bandwidth demands as resolution increases.



This graph illustrates average CPU usage across different resolutions for H.264, H.265, and VP9 codecs, indicating that H.265 requires significantly more CPU resources as resolution increases, while H.264 and VP9 maintain relatively stable usage. VP9 shows the lowest CPU consumption overall, highlighting its efficient processing capabilities compared to the other codecs.



The above graph depicts average latency across different resolutions for H.264, H.265, and VP9 codecs, revealing that latency increases with resolution for all codecs. H.264 exhibits the highest latency, while VP9 maintains the lowest, indicating better performance in minimizing delay as resolution rises.

Conclusion on the codecs:

Based on the above observations, H.264 is best suited for scenarios where bandwidth is not a constraint and lower latency is less critical, such as streaming applications with stable network conditions. H.265 is ideal for high-resolution streaming where bandwidth efficiency is essential but can tolerate higher CPU usage, making it suitable for newer devices. VP9 excels in environments requiring low CPU consumption and minimal latency, making it a strong choice for resource-constrained real-time applications.

4. QoS Algorithm Evaluation

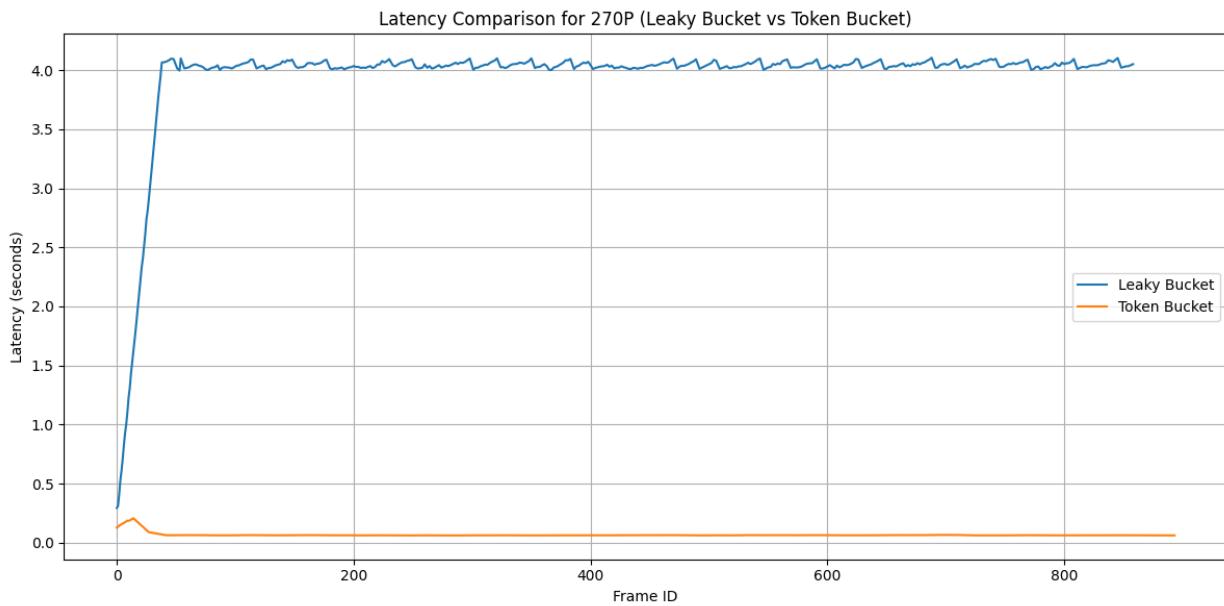
Similar to the integration setup of token bucket, leaky bucket setup was created and it's observations are as follows.

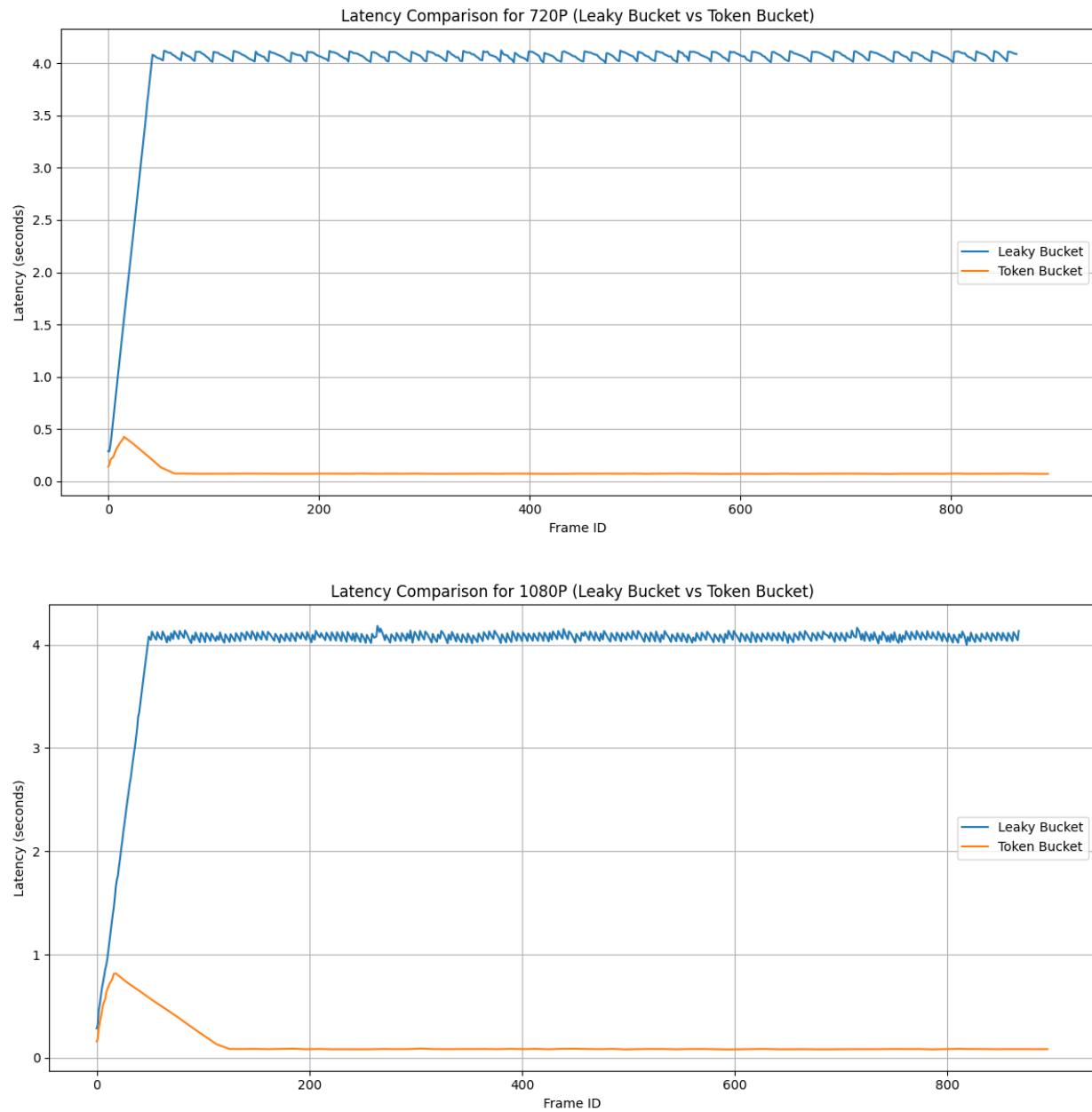
```
python3 ./video_server_with_leaky_bucket_h264.py
[!] Queue full. Dropping frame 154
[Frame 113] Sent | Size: 10302 bytes
[!] Queue full. Dropping frame 156
[Frame 115] Sent | Size: 10316 bytes
[!] Queue full. Dropping frame 158
[Frame 118] Sent | Size: 10255 bytes
[!] Queue full. Dropping frame 160
[Frame 120] Sent | Size: 10283 bytes
[!] Queue full. Dropping frame 162
[Frame 122] Sent | Size: 10282 bytes
[!] Queue full. Dropping frame 164
[Frame 124] Sent | Size: 10349 bytes
[!] Queue full. Dropping frame 166
[Frame 126] Sent | Size: 10308 bytes
[!] Queue full. Dropping frame 168
[Frame 128] Sent | Size: 10311 bytes
[!] Queue full. Dropping frame 170
[Frame 130] Sent | Size: 10310 bytes
[!] Queue full. Dropping frame 172
[Frame 132] Sent | Size: 10295 bytes
[!] Queue full. Dropping frame 174
[Frame 134] Sent | Size: 10286 bytes
[!] Queue full. Dropping frame 176
[Frame 136] Sent | Size: 10303 bytes
[!] Queue full. Dropping frame 178
[!] Queue full. Dropping frame 179
[Frame 138] Sent | Size: 10354 bytes
[!] Queue full. Dropping frame 181
[Frame 140] Sent | Size: 10317 bytes
[!] Queue full. Dropping frame 183
[Frame 142] Sent | Size: 10314 bytes
[!] Queue full. Dropping frame 185
[Frame 144] Sent | Size: 10371 bytes
[!] Queue full. Dropping frame 187
[Frame 146] Sent | Size: 10359 bytes
[!] Queue full. Dropping frame 189
[Frame 148] Sent | Size: 10369 bytes
[!] Queue full. Dropping frame 191
[Frame 151] Sent | Size: 10317 bytes
[!] Queue full. Dropping frame 193
[Frame 153] Sent | Size: 10334 bytes

Jun 10 15:37
python3 ./video_client_with_leaky_bucket_h264.py
Frame 73 | Latency: 4.024s
Frame 75 | Latency: 4.006s
Frame 77 | Latency: 4.002s
Frame 79 | Latency: 4.014s
Frame 81 | Latency: 4.023s
Frame 83 | Latency: 4.026s
Frame 85 | Latency: 4.042s
Frame 87 | Latency: 4.001s
Frame 89 | Latency: 4.021s
Frame 91 | Latency: 4.026s
Frame 93 | Latency: 4.024s
Frame 95 | Latency: 4.022s
Frame 97 | Latency: 4.015s
Frame 99 | Latency: 4.024s
Frame 101 | Latency: 4.038s

H.264 Stream (Leaky Bucket)
Frame 148 | Latency: 4.075s
Frame 148 | Latency: 4.090s
Frame 151 | Latency: 4.082s
Frame 153 | Latency: 4.019s
```

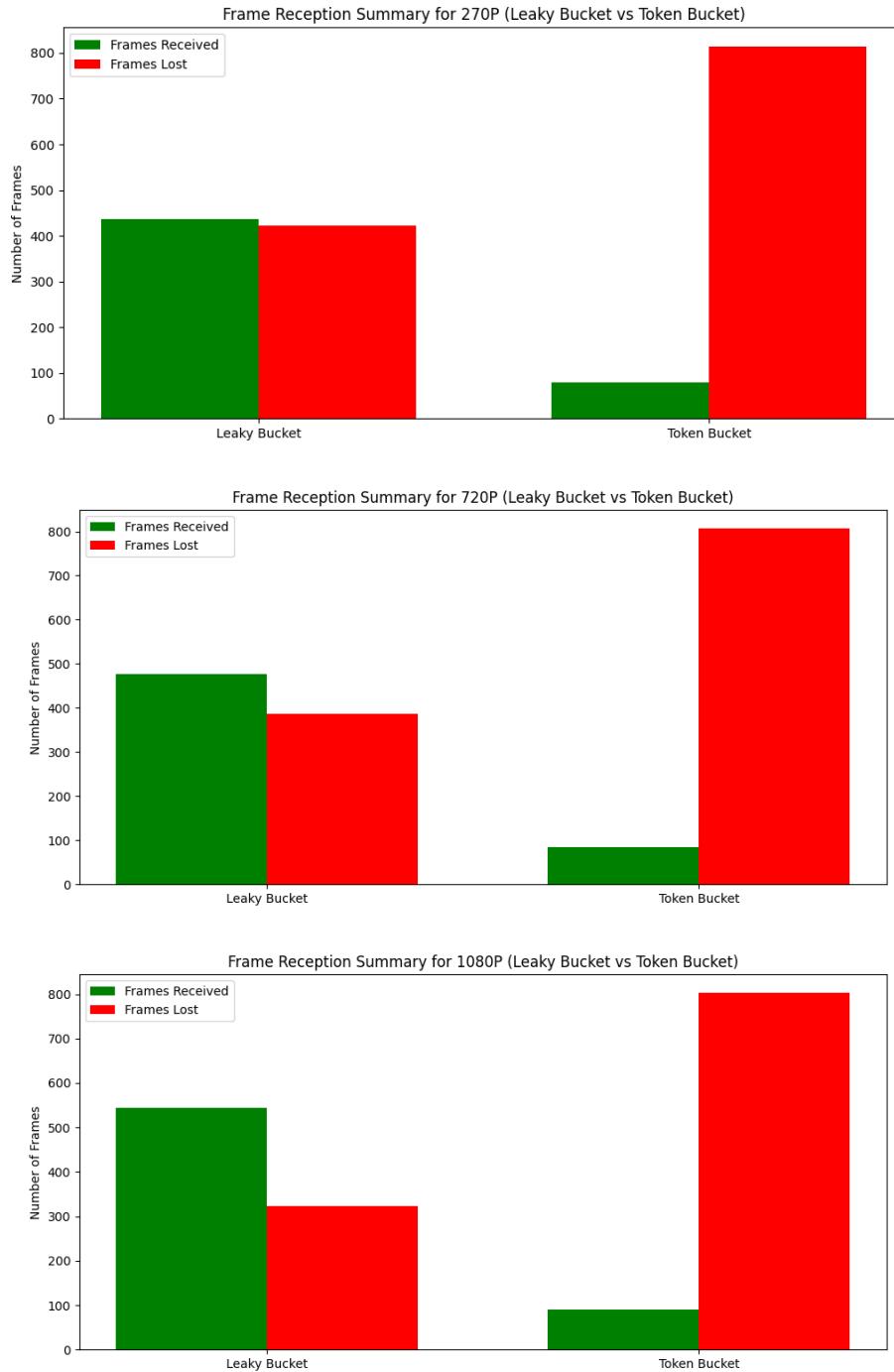
Following are the latency comparison results of leaky bucket vs token bucket.





Based on the above observations, the token bucket algorithm shows significantly less latency throughout all the experimented resolutions (270p, 720p, 1080p) compared to leaky bucket algorithm. This suggests that the token bucket algorithm is more suitable for real-time applications such as live streaming and online gaming, where minimizing latency is crucial for a smooth user experience. Additionally, its ability to handle bursty traffic efficiently makes it ideal for scenarios with varying bandwidth availability.

The following graphs depict numbers of frames received and lost per each algorithms at different resolutions.



Based on the above three graphs, we could observe that compared to token bucket algorithm, more frames survive when using the leaky bucket algorithm. This suggests that the leaky bucket algorithm is more suitable for applications where ensuring frame delivery is critical, such as buffered video streaming where maintaining a consistent quality of service is prioritized over low latency. Its ability to smooth out traffic can help prevent packet loss in congested networks

Therefore,

- For Real-time conferencing (low-latency): Token Bucket Algorithm
- For Buffered video streaming (high-quality): Leaky Bucket Algorithm

Bonus:

Figure out which works best for real-time streaming vs video streaming

1. Algorithm Wise

- Real-Time Streaming: The Token Bucket Algorithm is best suited due to its ability to handle bursty traffic and minimize latency, which is crucial for live interactions.
- Video Streaming(buffered): The Leaky Bucket Algorithm works better as it can ensure consistent frame delivery, helping to maintain quality during playback even in variable network conditions.

2. Codec Wise

- Real-Time Streaming: VP9 is the most effective codec, as it offers low latency and efficient compression, making it ideal for real-time applications.
- Video Streaming: H.265 or H.264 is preferable for buffered video streaming due to its high compression efficiency, allowing for high-quality playback at lower bandwidth usage.