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

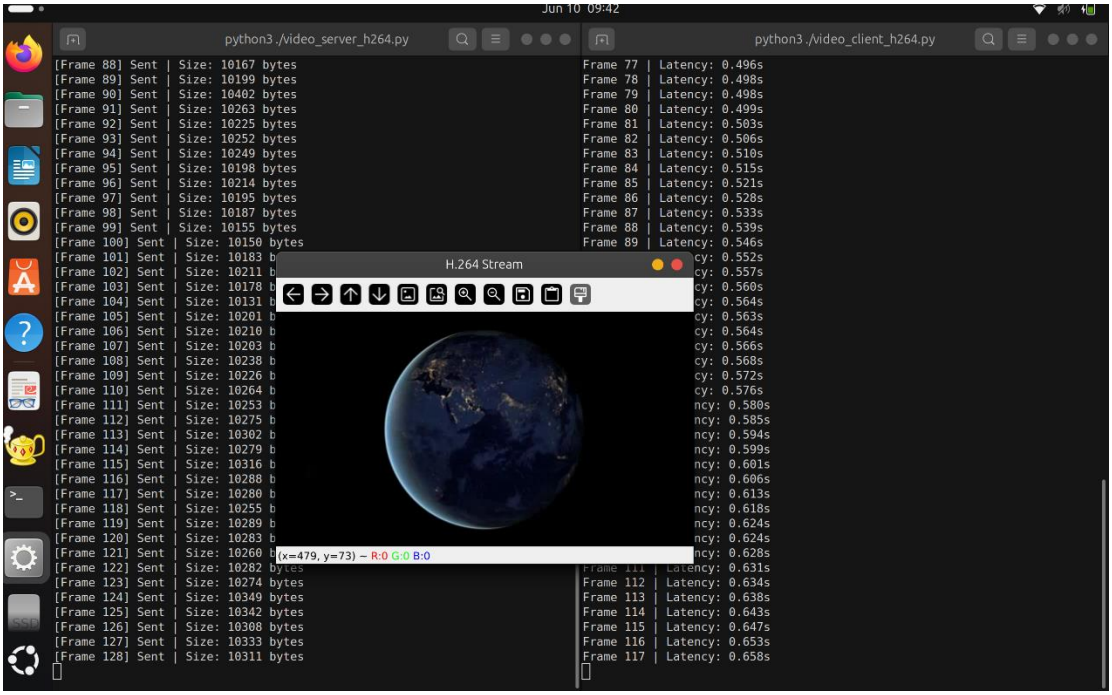
This was implemented using the python script "./video_server_h264.py" which captures video frames from a given mp4 file using OpenCV, then encodes them to H.264 format with FFmpeg, and streams the encoded frames over a TCP socket to a connected client. It sends each frame's ID, timestamp, and size along 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)

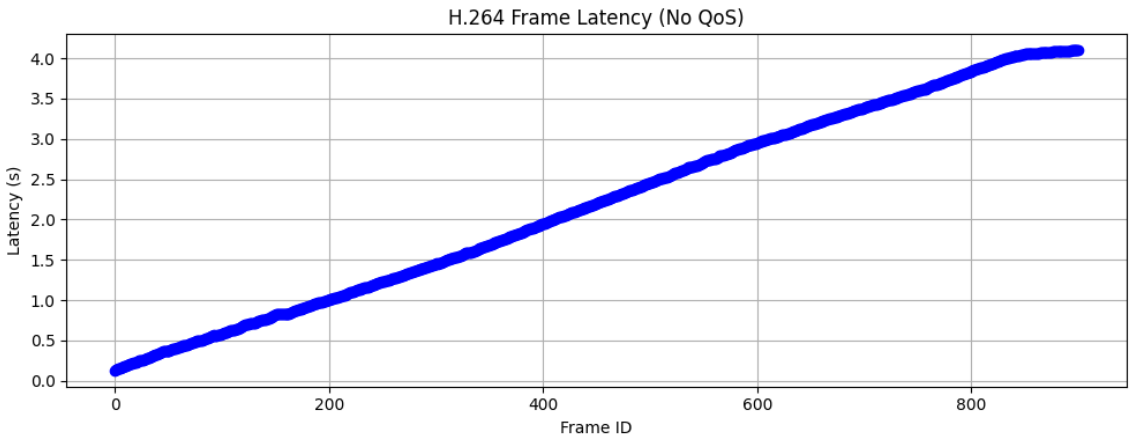
Task 2: Client Setup

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

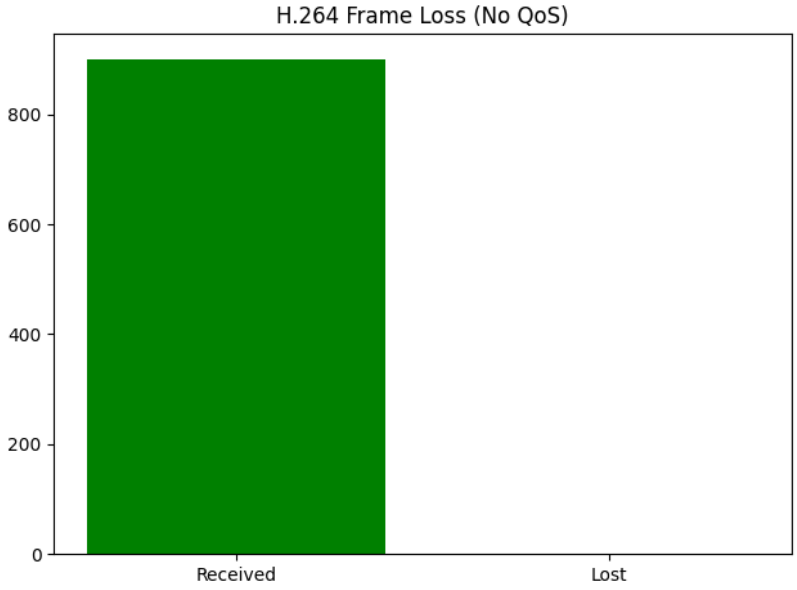
Following is a screenshot taken at the initial testing.





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.

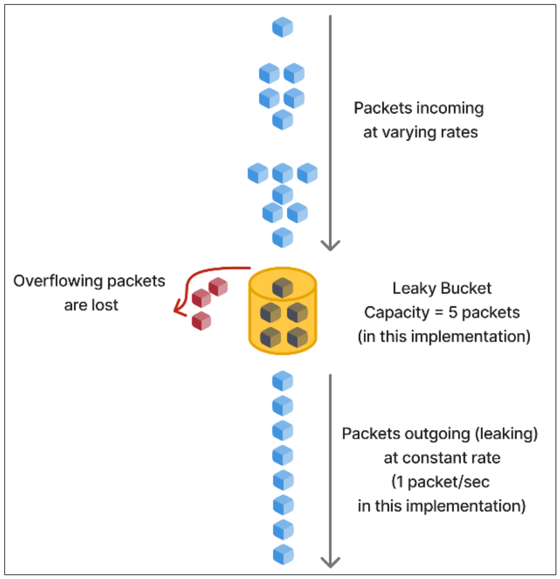


Considering all the above observations we can conclude that, a QoS mechanism is essential to avoid potential bottlenecks in the video stream, but it may also lead to frame/packet drops.

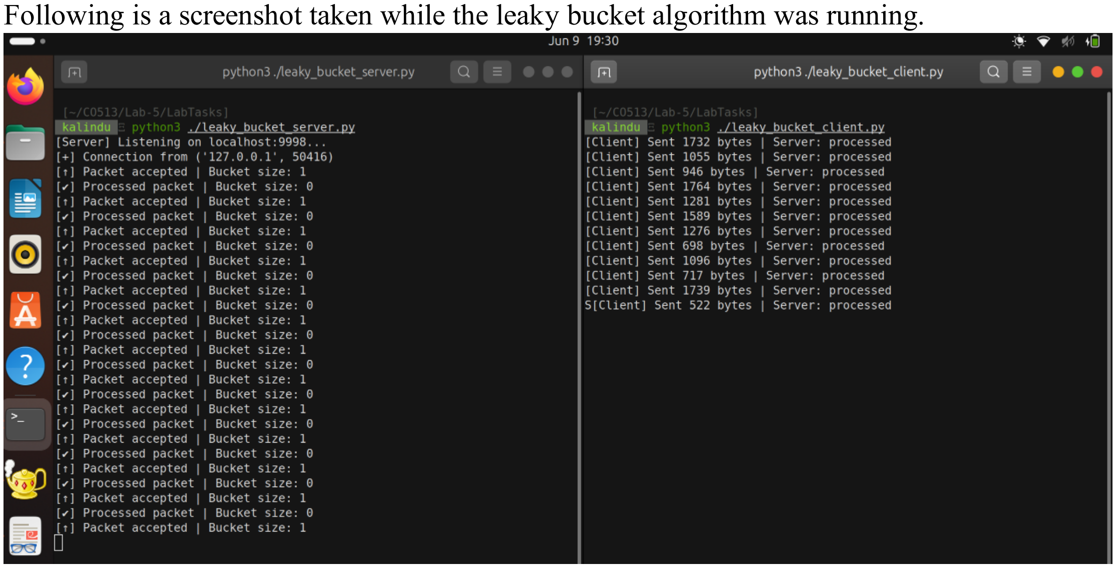
Part 2: Implement QoS Algorithms

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)

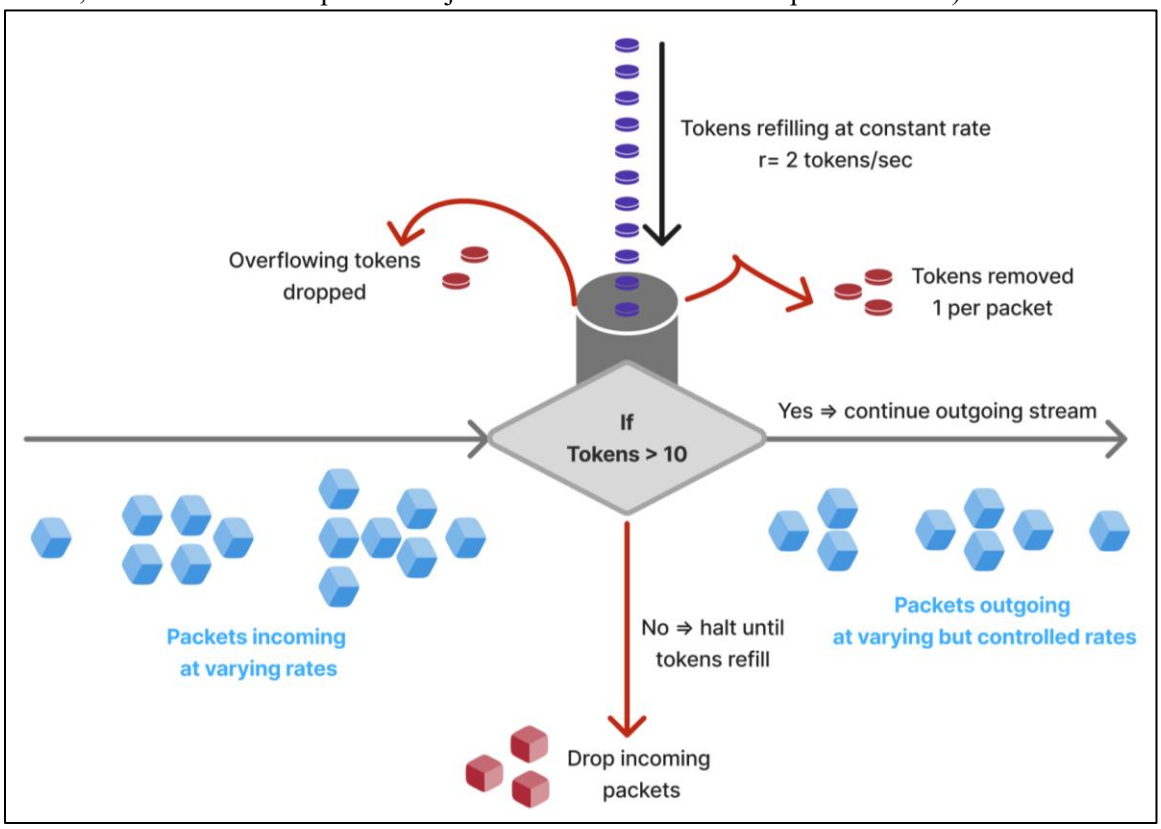


2. Leaky Bucket Client Logic ("./leaky_bucket_client.py"):
   This connects to the server and sends a series of randomly sized packets (between 500 and 2000 bytes) in a loop. After sending each packet, it waits for the server's response and pauses for a random interval between 10ms and 500ms before sending the next packet, allowing for realistic and controlled transmission rates.
   Following is a screenshot taken while the leaky bucket algorithm was running.



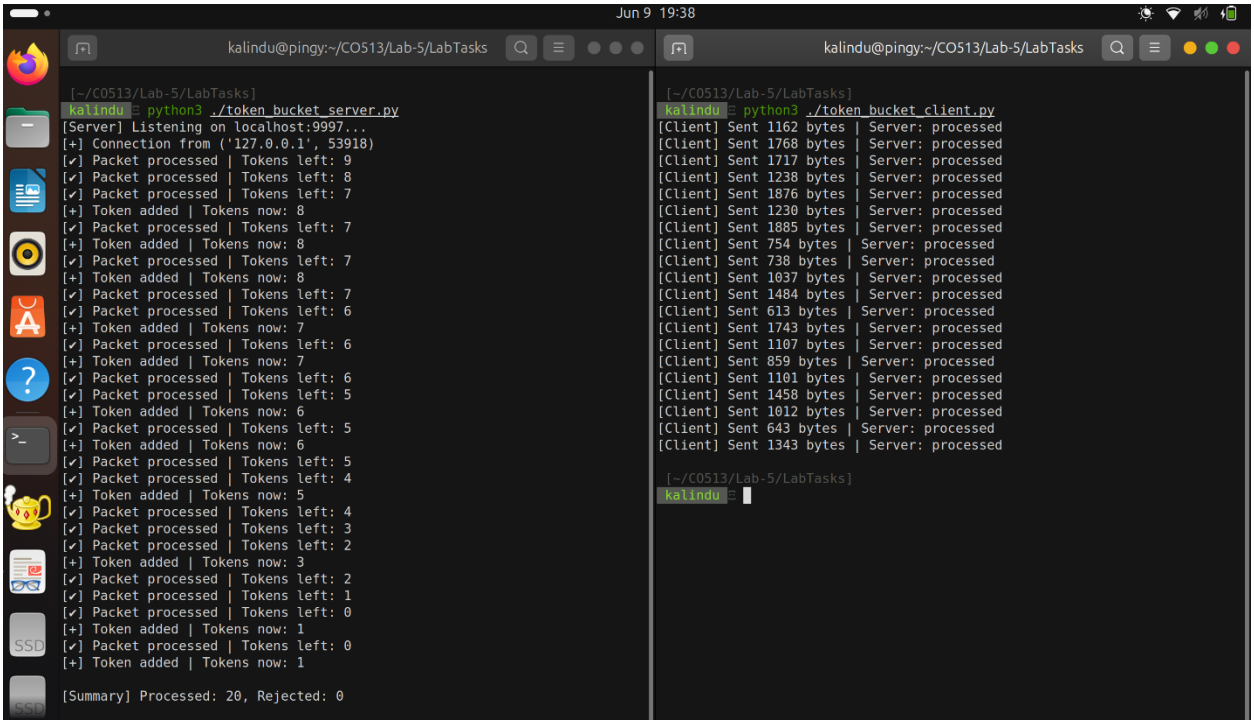
Task 5: Token Bucket Algorithm

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.

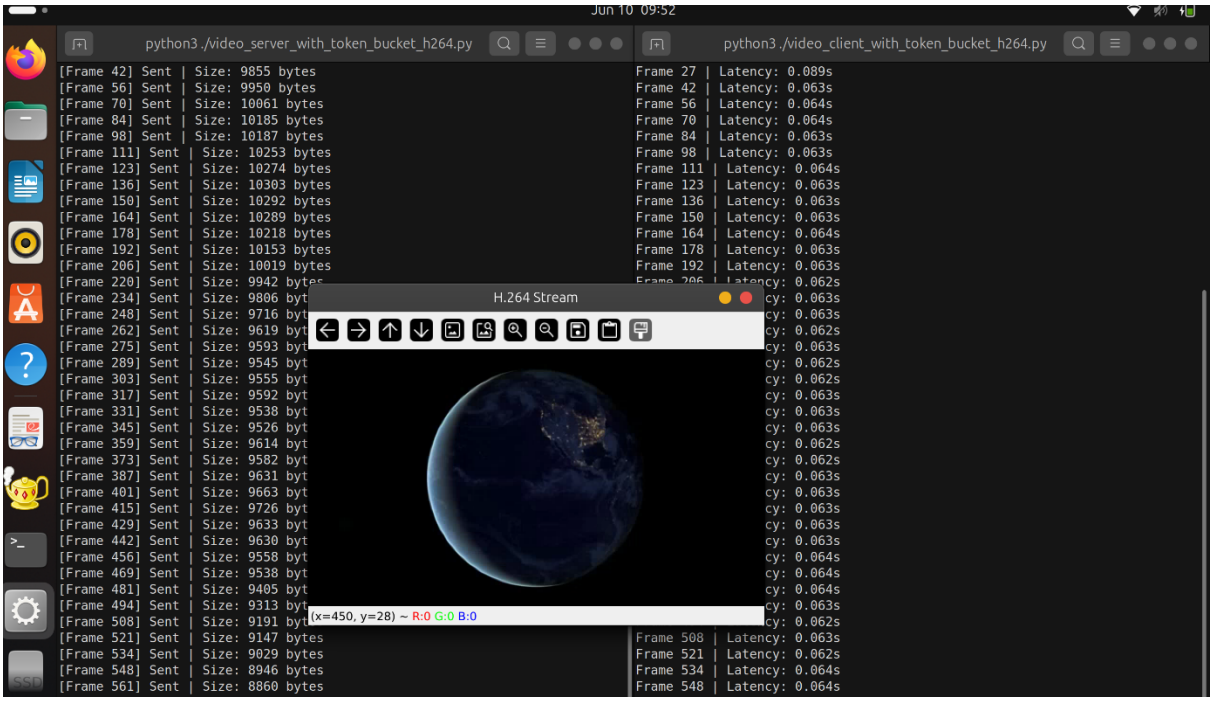
   The following is a screenshot taken while running the token bucket algorithm. (PTO)



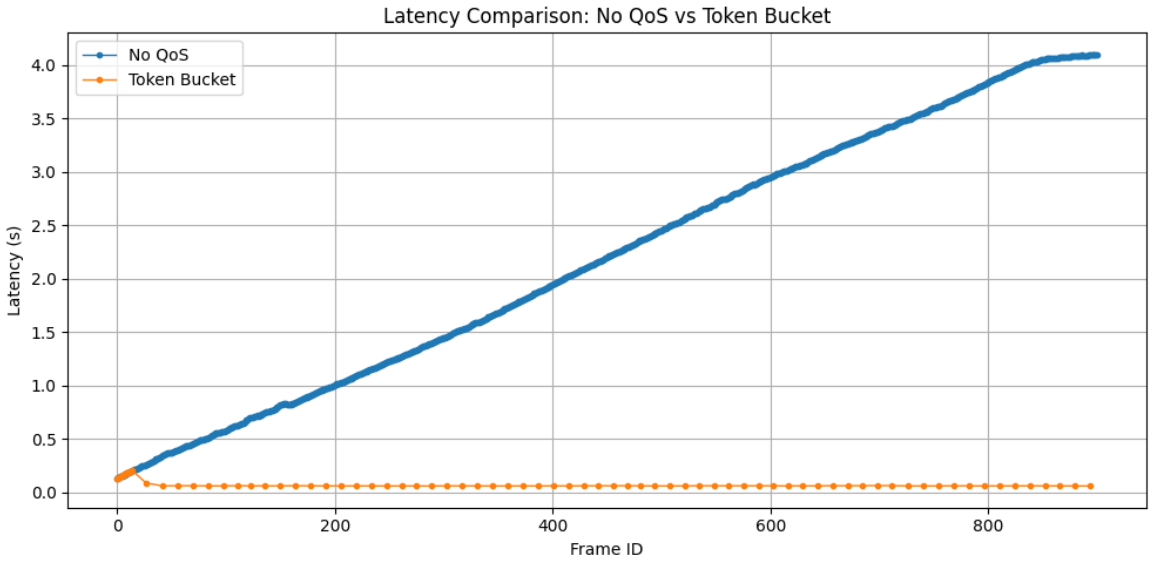
Part 3: QoS Integration for Streaming

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 intergration)

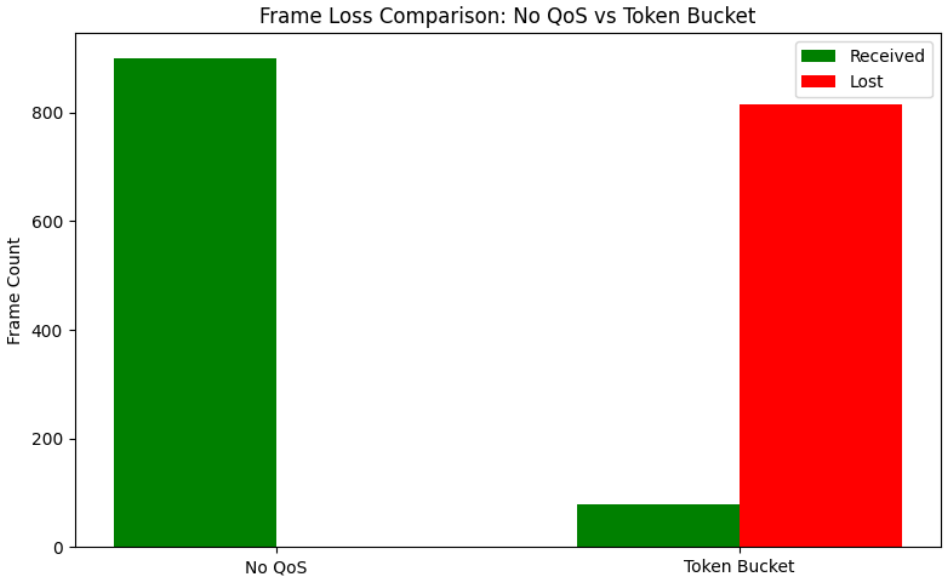


Testing and comparing with the no-qos implementation



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.

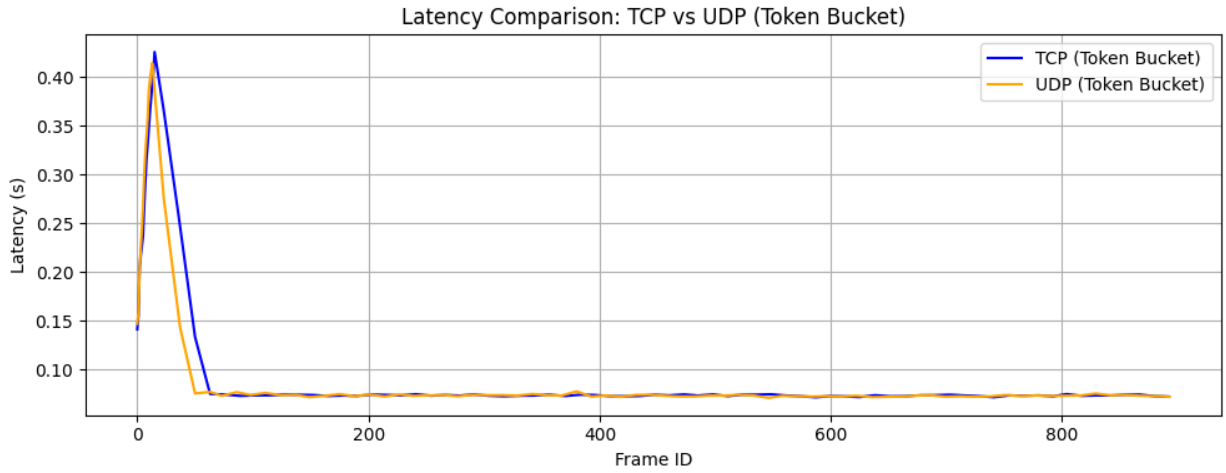


The above lost frame ratio can be lowered, if the refill rate, token bucket capacity, and packets dropped rate per token are adjusted accordingly.

Experimentation & Analysis

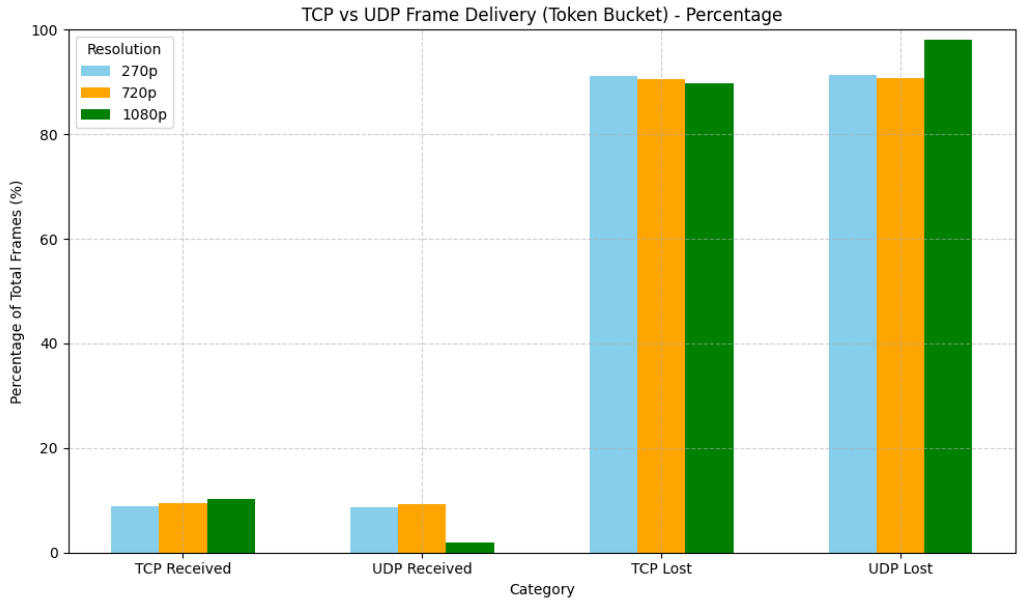
1. Protocol Behavior (TCP vs. UDP)

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

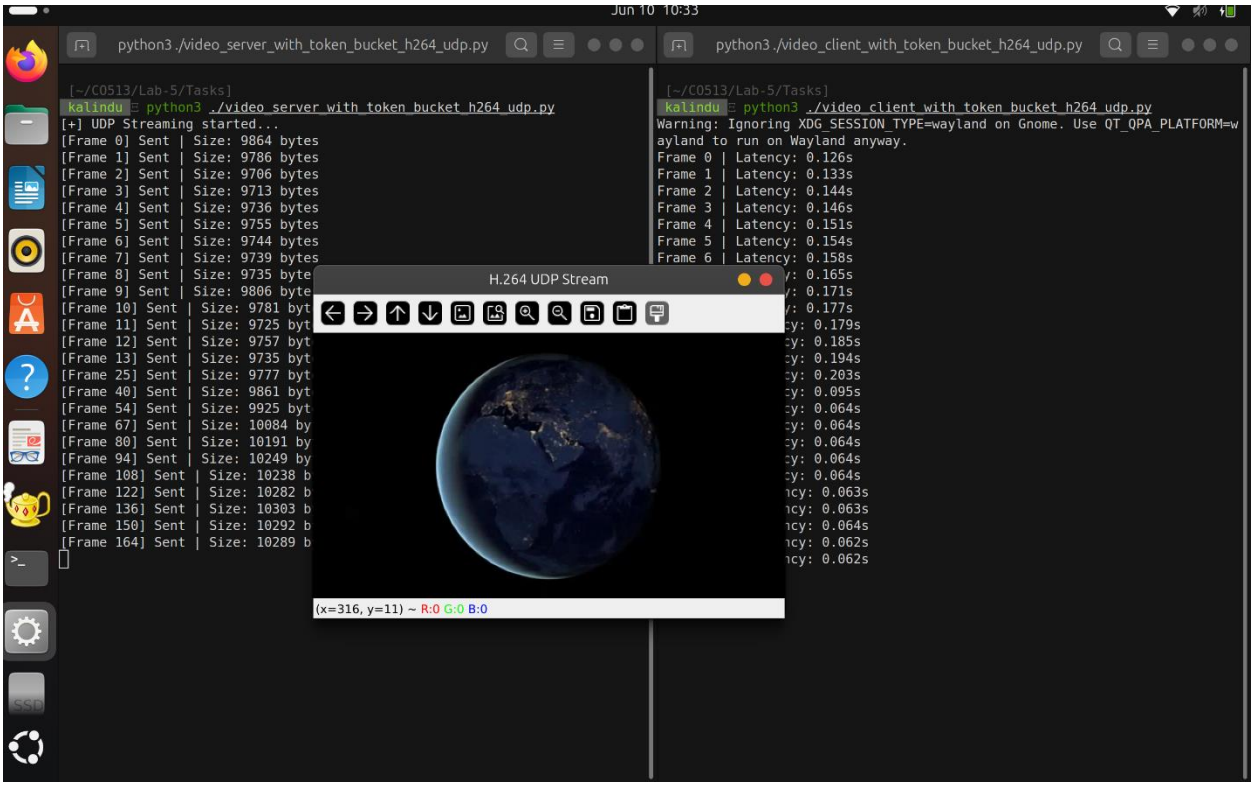


As it can be observed from the above graph, UDP shows slightly less latency for the video traffic. The gap between the latencies of the two protocols would grow larger if the frame rate, video quality are increased. This indicates UDP's suitability for real time traffic over TCP. (like Live streaming,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

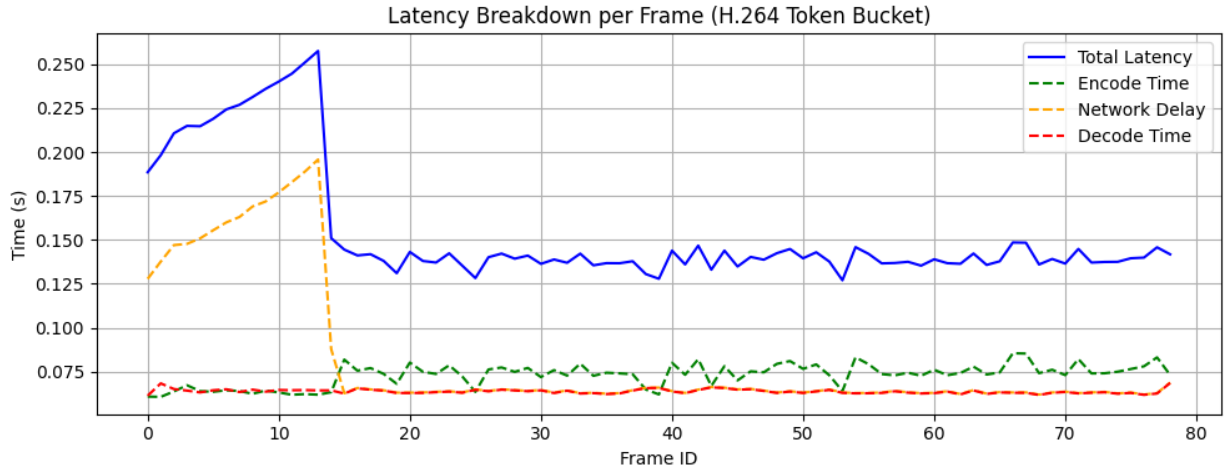


Following is a snapshot taken while running the UDP server-client setup.



2. Latency Analysis

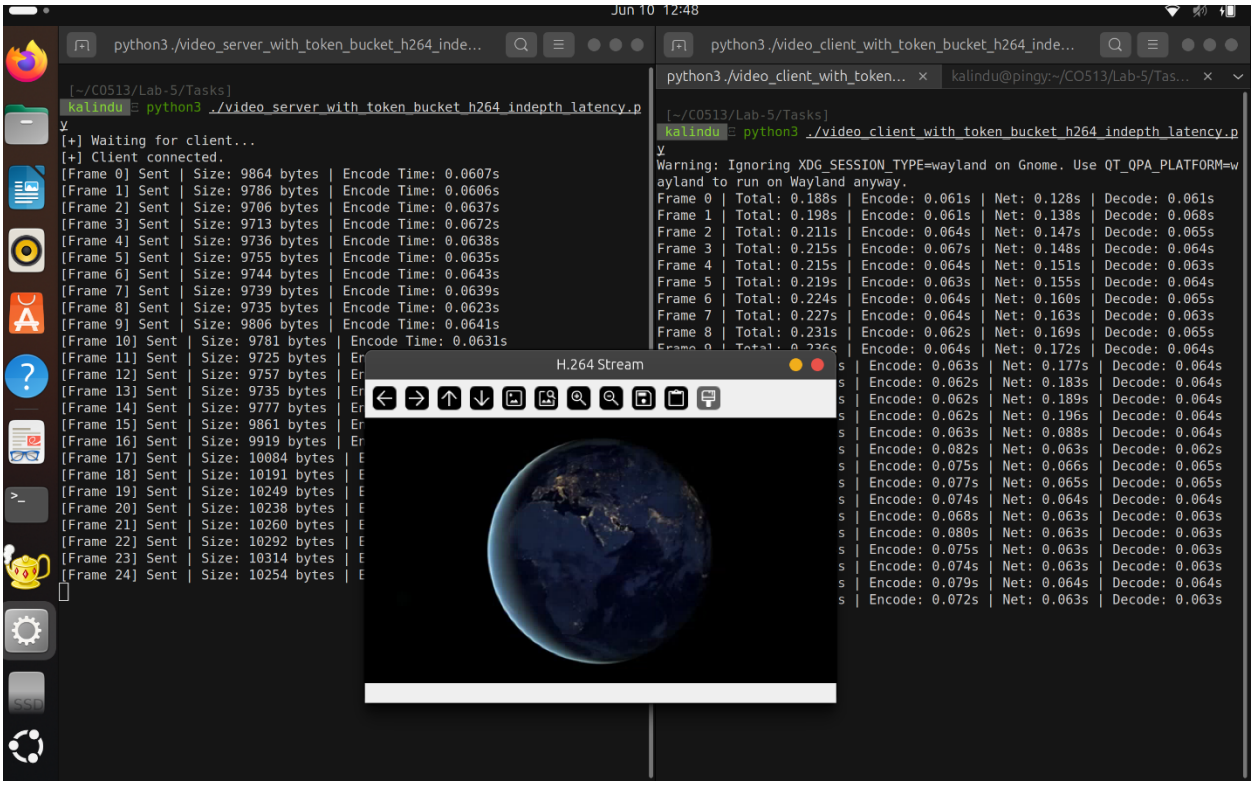
For this analysis, I have modified the server-client setup to log various latency metrics during video streaming, capturing total latency, encoding latency, network delay, and decoding latency for each frame. The data is saved to CSV files and visualized in plots to analyze performance and frame loss.



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.

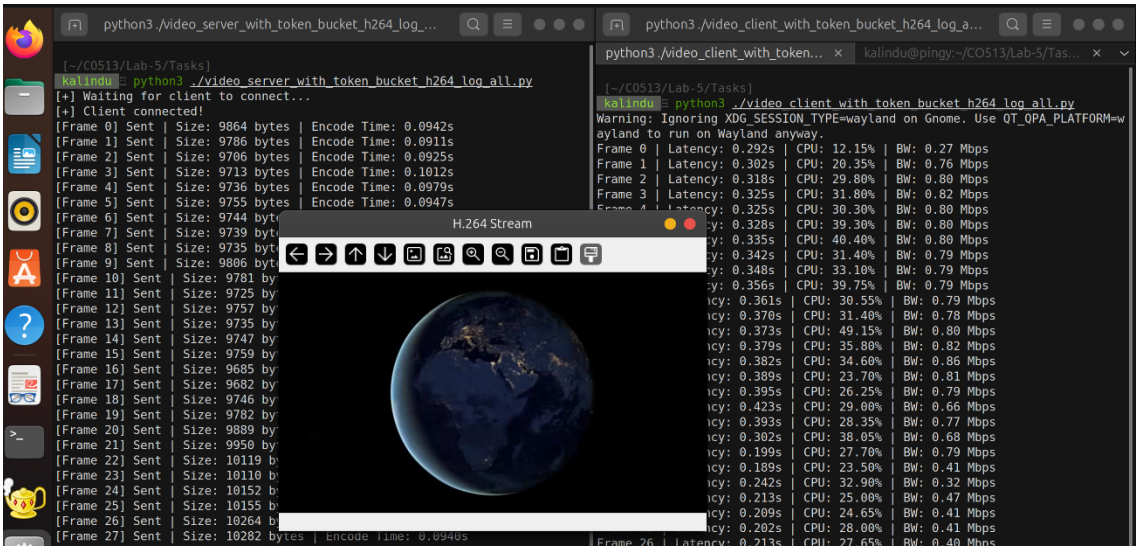
Following output shows that, in detail latency logs are being shared between the server and client.

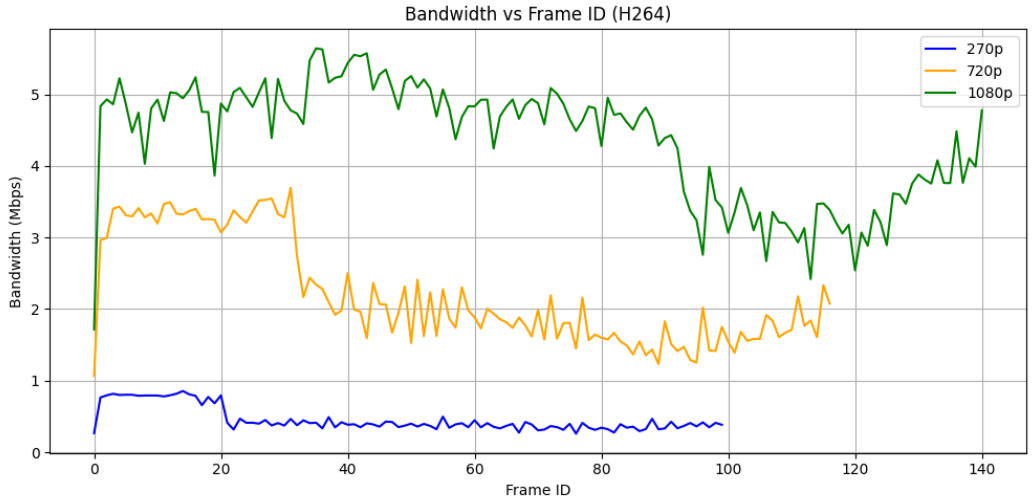


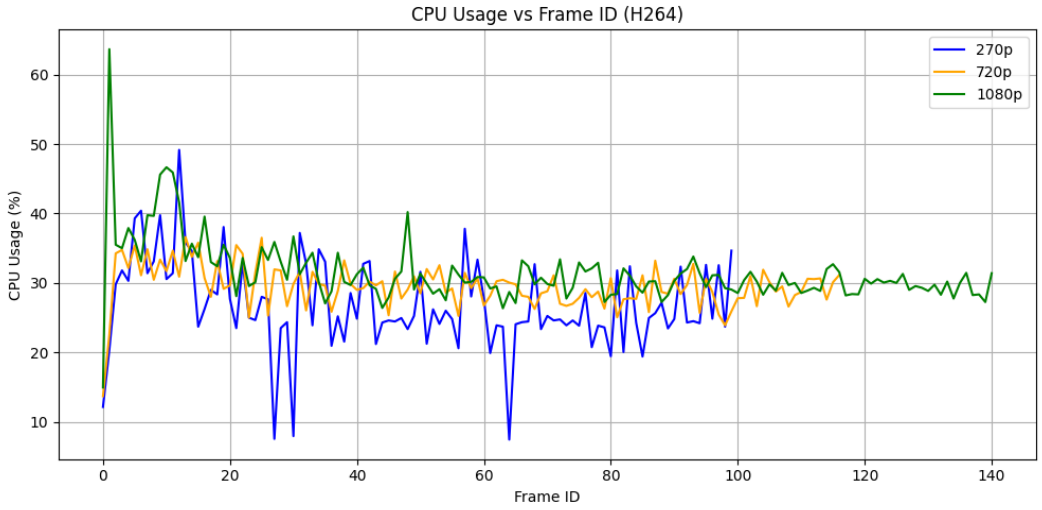
3. Encoding Efficiency

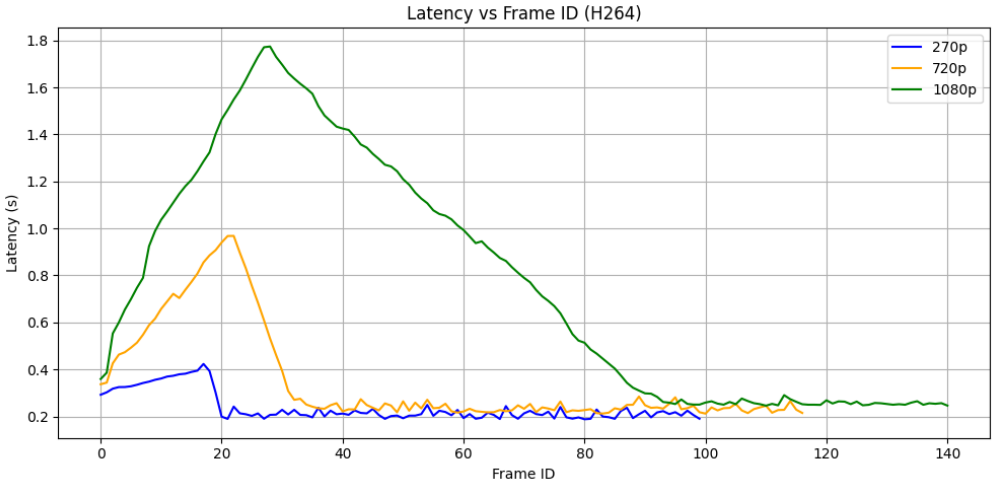
For this part I have used real time video codecs H.264, VP9, H.265 with varying resolutions of 270p, 720p and 1080p while logging latency, CPU usage, and bandwidth.

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



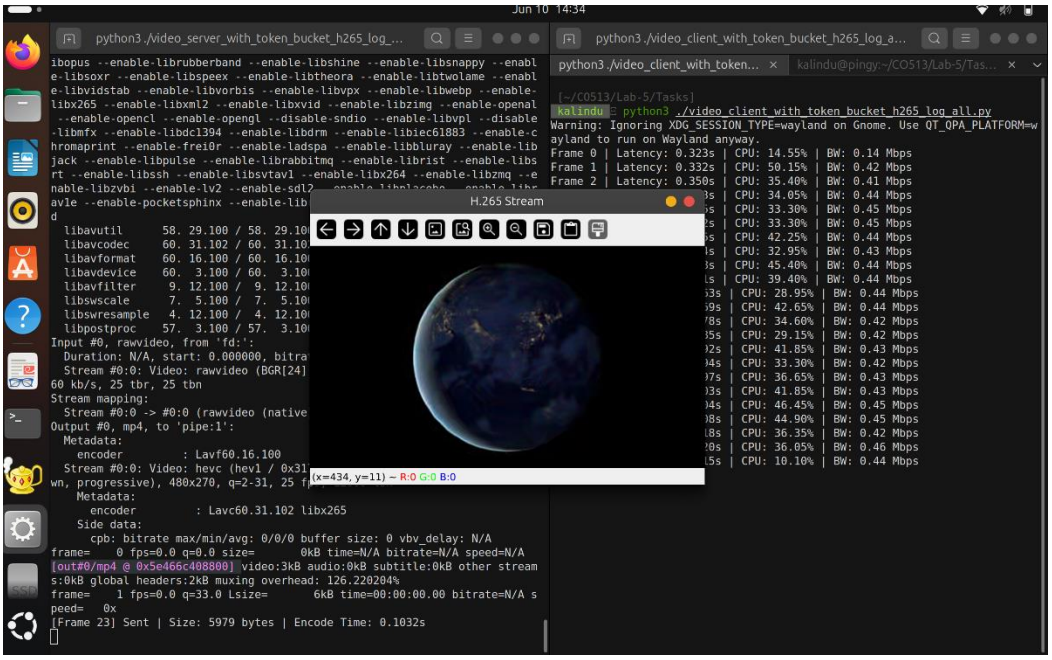


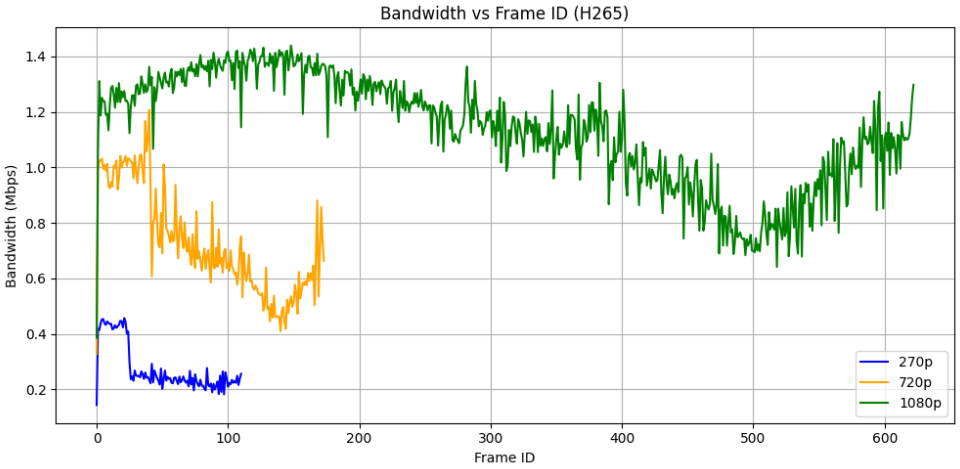


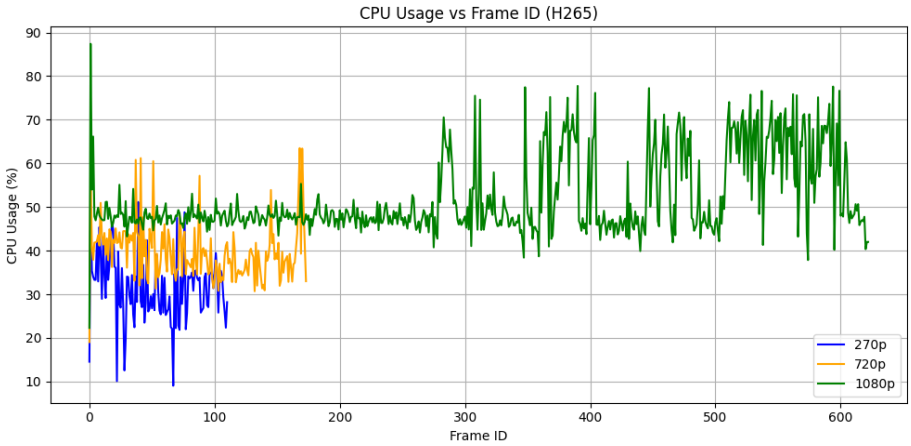


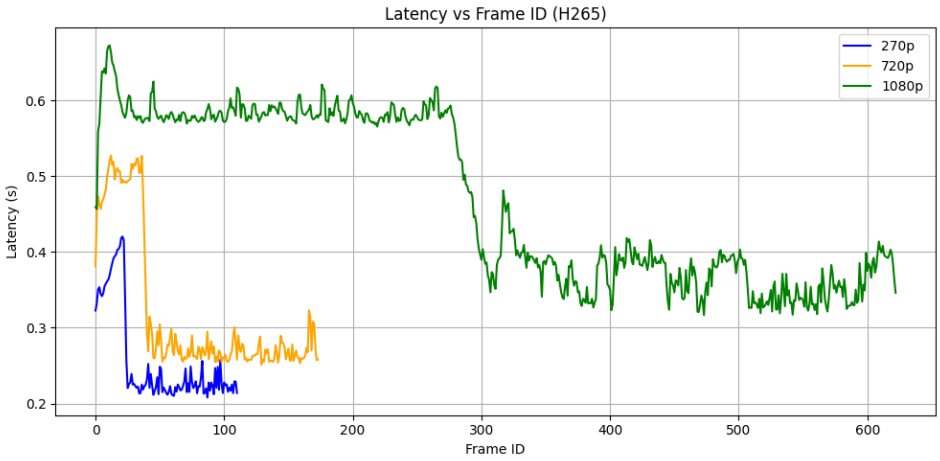
It can be observed from the above graphs that when increasing the video quality, H.264 codec's cpu consumption rate stays stable while latency and bandwidth increases.

2. H.265 Codec


H.265 Stream
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
Frame 0 | Latency: 0.323s | CPU: 14.55% | BW: 0.14 Mbps
Frame 1 | Latency: 0.332s | CPU: 50.15% | BW: 0.42 Mbps
Frame 2 | Latency: 0.350s | CPU: 35.40% | BW: 0.41 Mbps

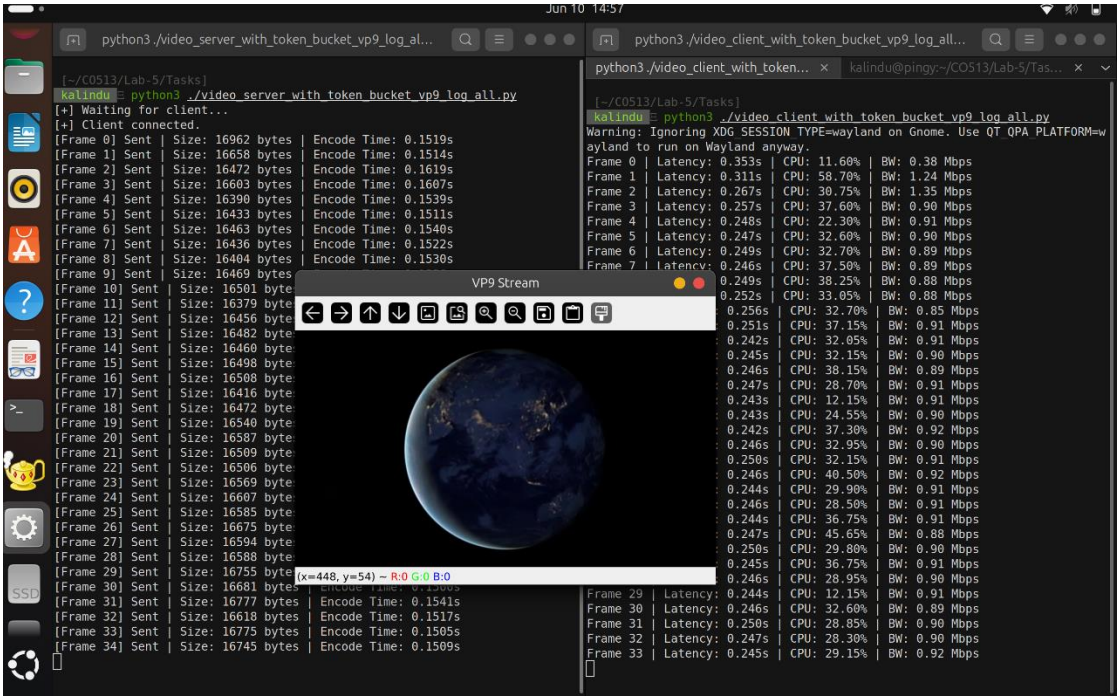

Bandwidth vs Frame ID (H265)
Bandwidth (Mbps)
Frame ID
270p
720p
1080p


CPU Usage vs Frame ID (H265)
CPU Usage (%)
Frame ID
270p
720p
1080p

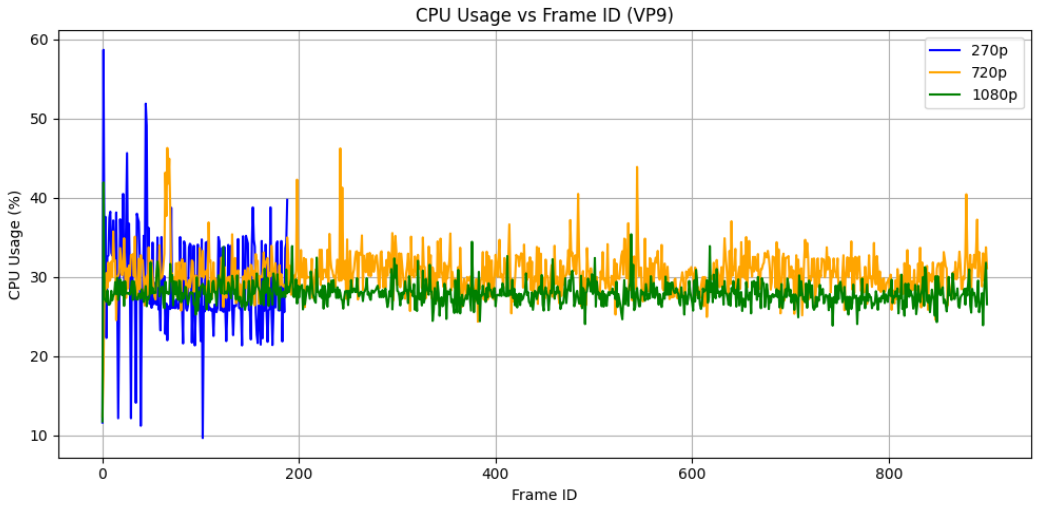


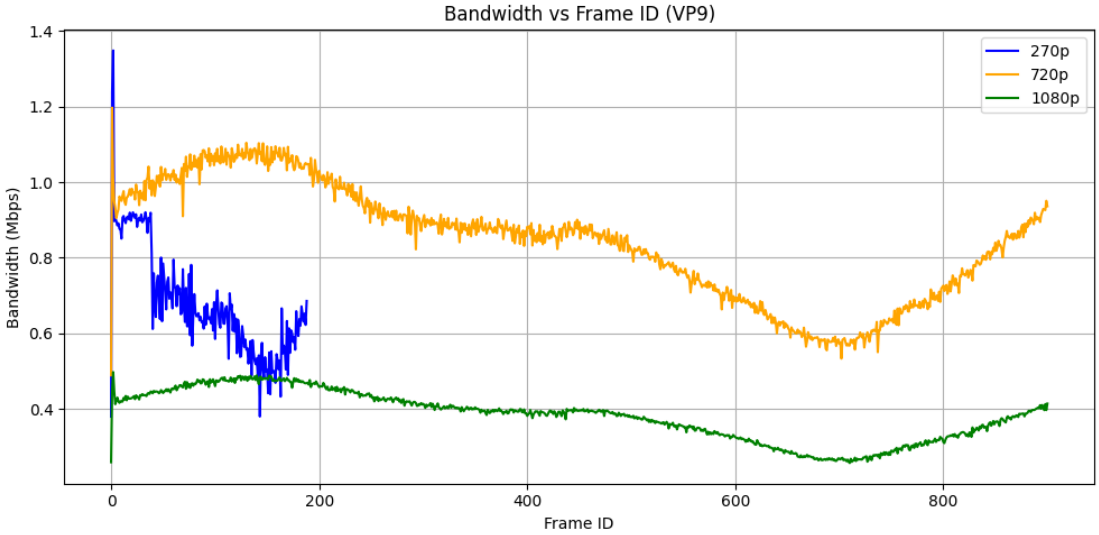
It can be observed from the above graphs that, H.265 codec behaves similar to the H.264 codec when video resolution is increased.

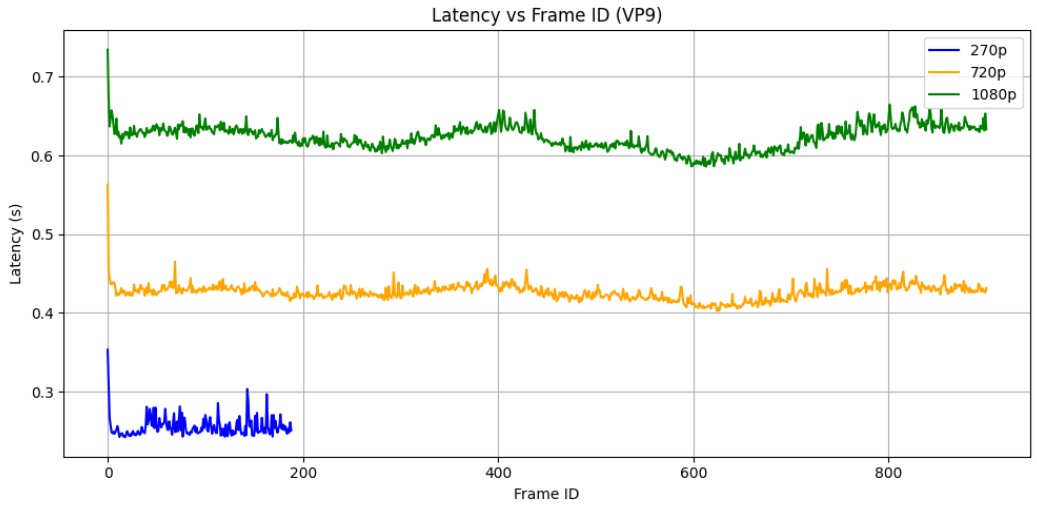
3. VP9 Codec



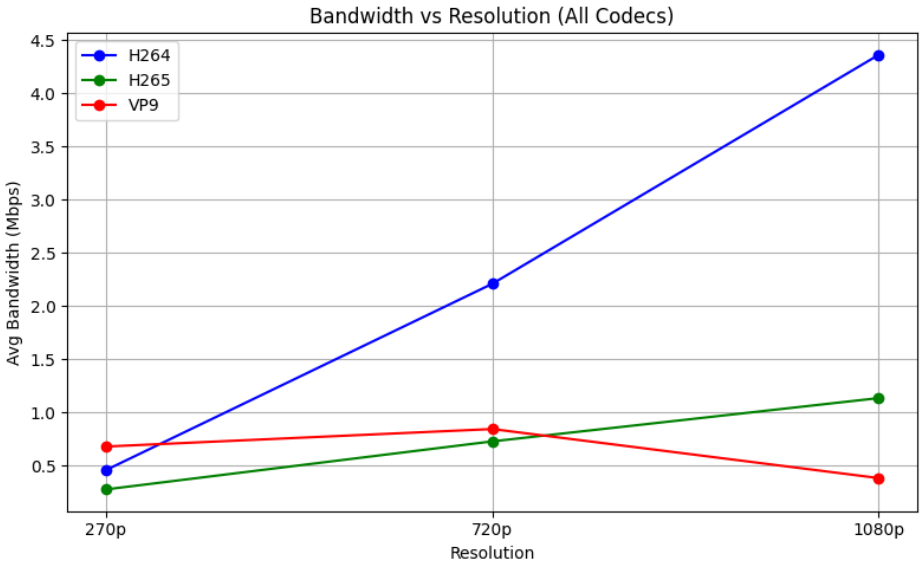
It can be observed from the following graphs that, VP9 codec behaves a bit differently in terms of bandwidth compared to H.264 and H.265 by performing well for mid quality streams.


CPU Usage vs Frame ID (VP9)
CPU Usage (%)
Frame ID
270p
720p
1080p

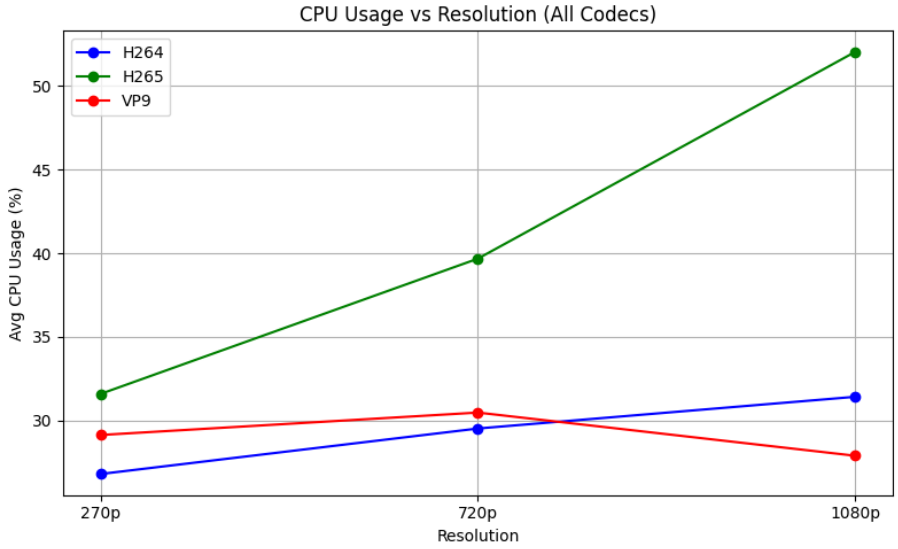

Bandwidth vs Frame ID (VP9)
Bandwidth (Mbps)
Frame ID
270p
720p
1080p


Latency vs Frame ID (VP9)
Latency (s)
Frame ID
270p
720p
1080p

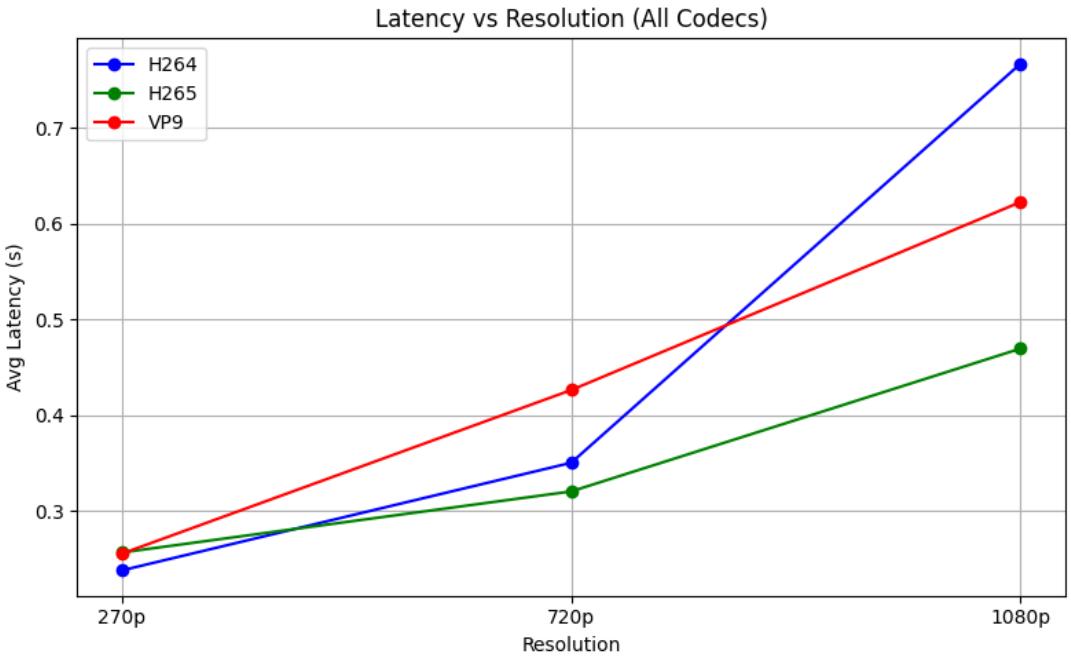
To get an overall idea of how all the three codecs behave with increasing video resolutions, they were plotted using average values of latency, bandwidth, and CPU usage as follows.



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.



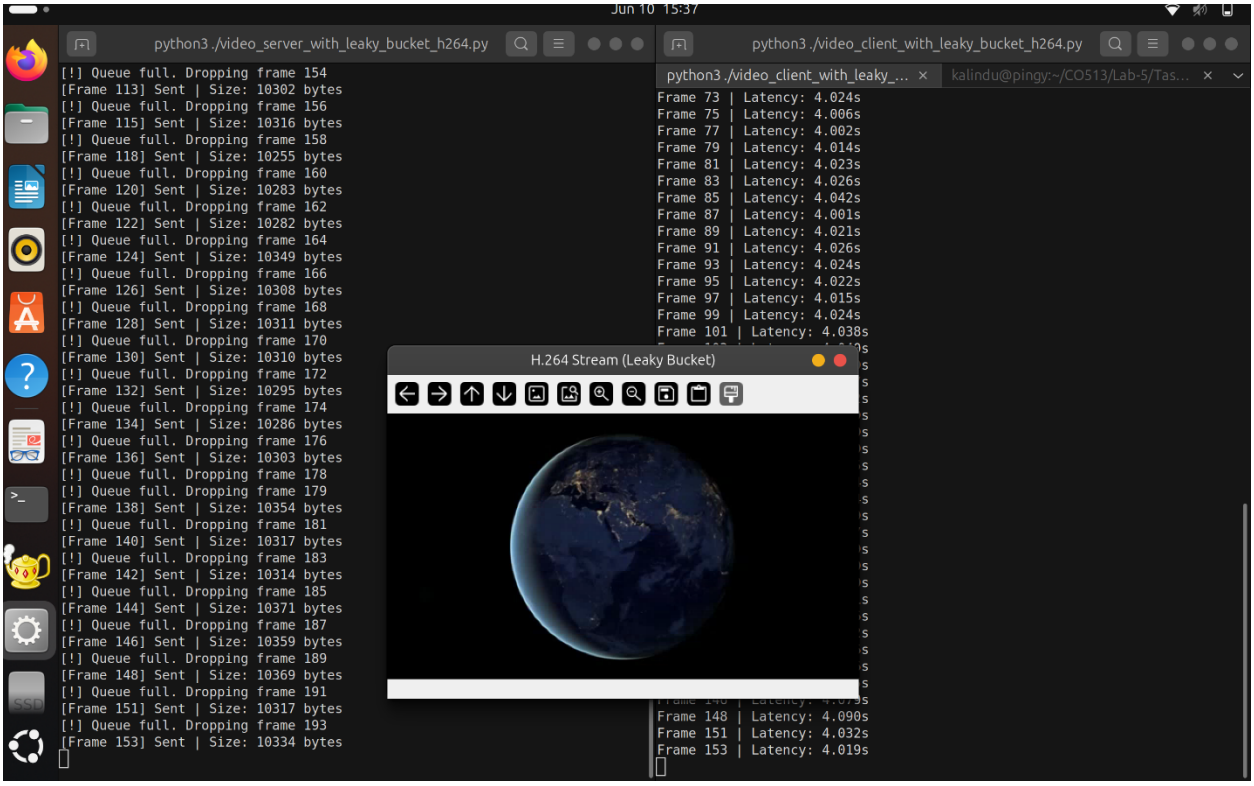
The above graph depicts average latency across different resolutions for H.264, H.265, and VP9 codecs, revealing that latency increases with resolution for all codecs. H.264 exhibits the highest latency, while VP9 maintains the lowest, indicating better performance in minimizing delay as resolution rises.

Conclusion on the codecs:

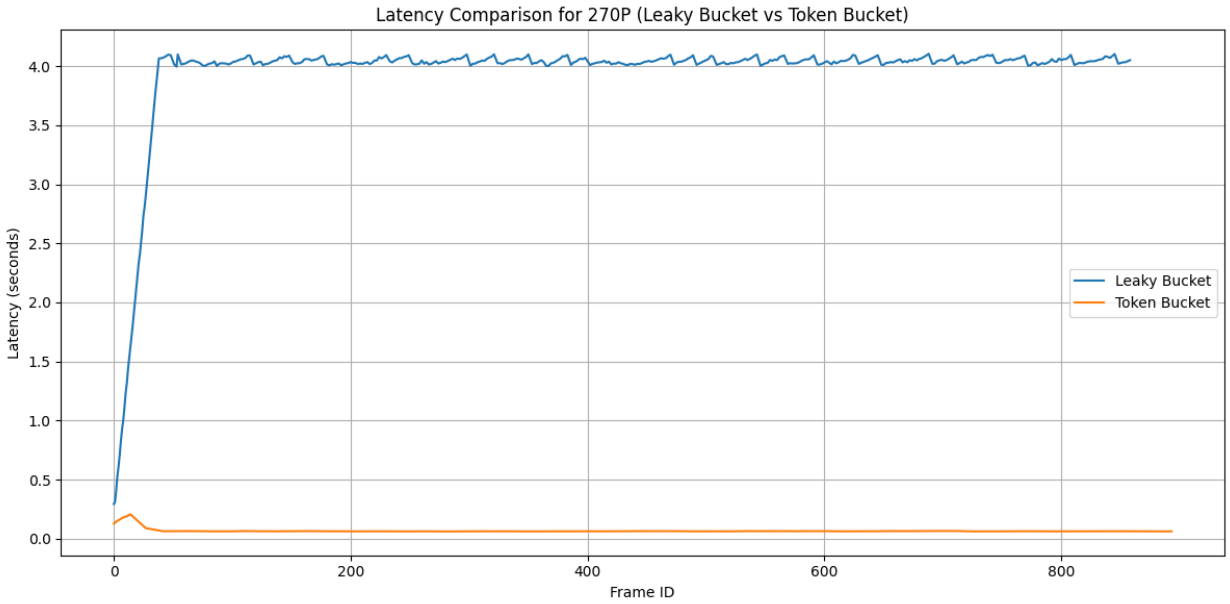
Based on the above observations, H.264 is best suited for scenarios where bandwidth is not a constraint and lower latency is less critical, such as streaming applications with stable network conditions. H.265 is ideal for high-resolution streaming where bandwidth efficiency is essential but can tolerate 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.

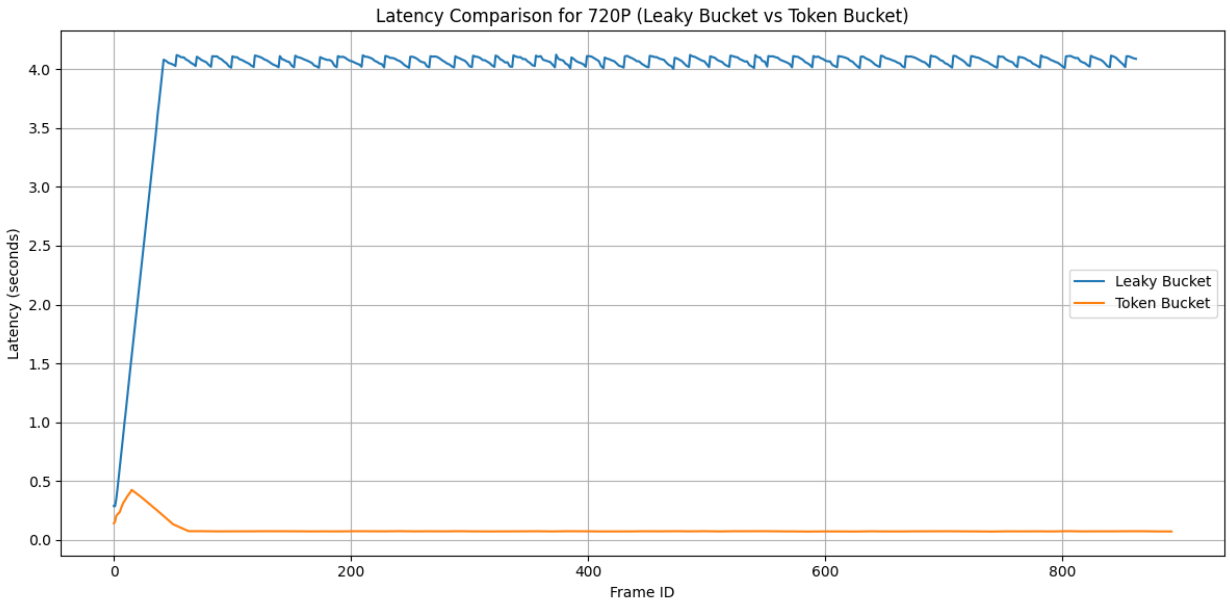
4. QoS Algorithm Evaluation

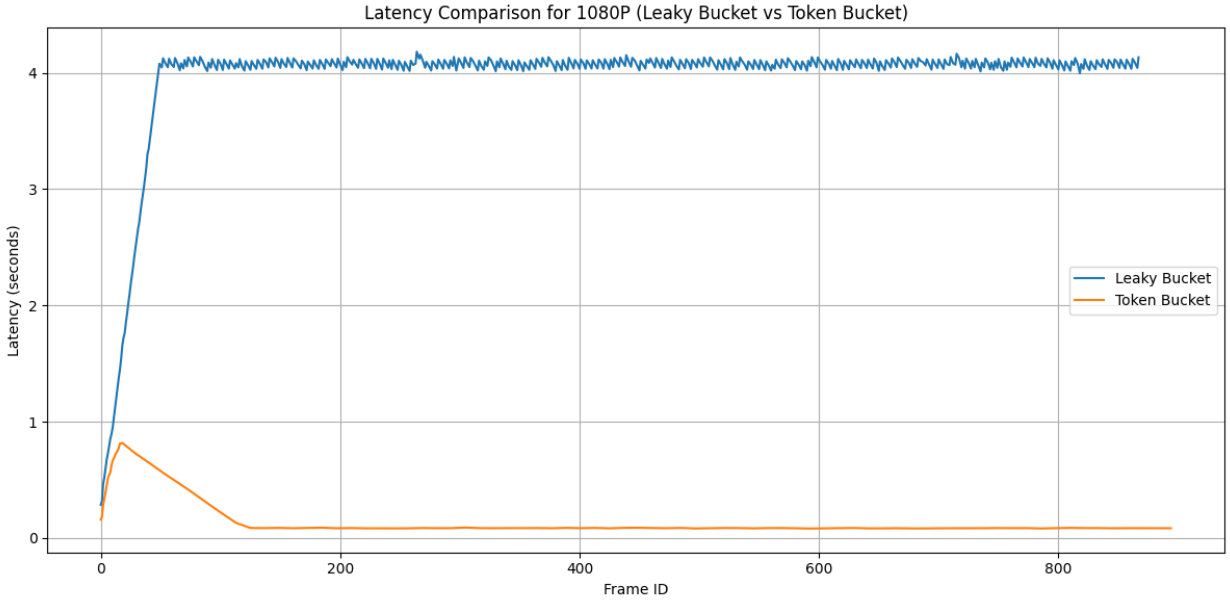
Similar to the integration setup of token bucket, leaky bucket setup was created and it's observations are as follows.



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

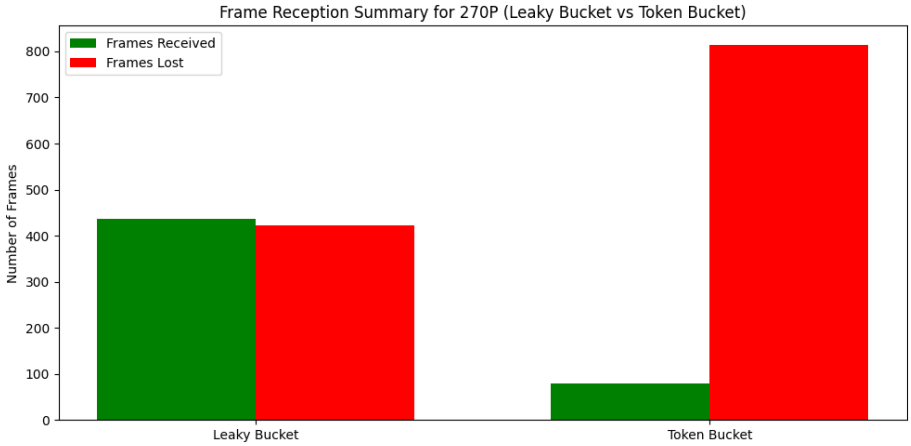


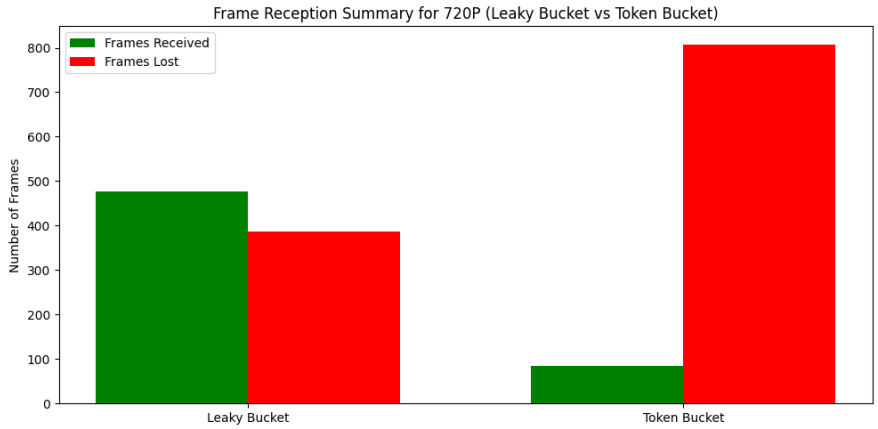


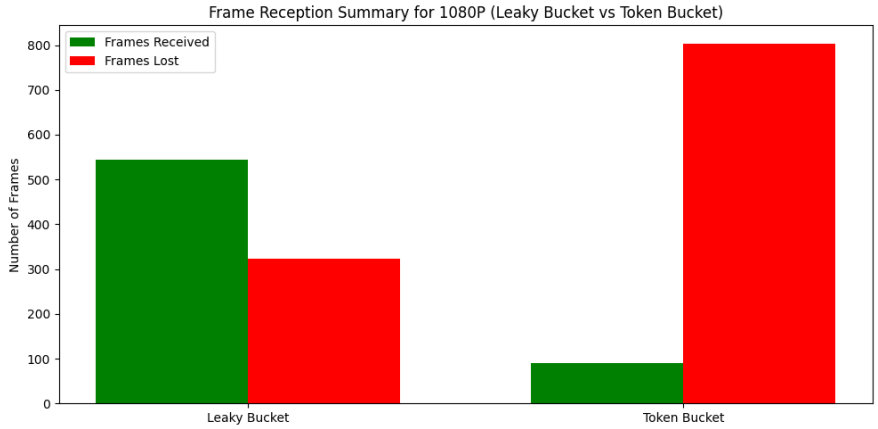


Based on the above observations, the token bucket algorithm shows significantly less latency throughout all the experimented resolutions (270p,720p,1080p) compared to leaky bucket algorithm. This suggests that the token bucket algorithm is more suitable for real-time applications such as live streaming 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.