CS425 Mini Project
2017-18 I Semester
Feedback-driven Concurrent Video Broadcast Protocol

Ayush Tulsyan (14167)
Pawan Kr. Patel (14453)
Shivansh Rai (14658)
Group 19

November 12, 2017

1 Introduction

Video streaming is one of the most exciting applications on the Internet. It already created a new business known as Internet broadcast, or Intercast/Webcast. In this project we get a glimpse of broadcasting where people can enjoy not only scheduled live but also on-demand and personalized programs. Stirring more excitement is the idea that anyone will be able to set up an Internet TV station and broadcast their video programs on the Internet just like we publish our "homepage" on the Web today.

2 Objective

To build a **RTP** (Real-time Transport Protocol) and **RTCP** (Real-time Transport Control Protocol) based Video Broadcasting application for multiple concurrent clients.

The implemented model constitutes of two connections -

1. **UDP Connection: Transmission**
   This connection is responsible for transferring data to the client. Since we are implementing a broadcast server, states which have been lost during transmission are irrelevant in the future, ruling out the possibility of data retransmissions.

2. **UDP Connection: Feedback**
   This connection is responsible for collecting the transmission feedback from the client at regular intervals. Each feedback contains the number of payloads client received, which can be used by the server to adjust parameters to ensure a better QoS (Quality of Service).

   If the number of concurrent clients increases, the server can be overwhelmed by the large amount of feedbacks from all the clients and the corresponding processing overhead. This issue is addressed by adjusting the duration between two adjacent feedbacks depending on the number of active clients. This means that a large number of clients will result in a larger duration between adjacent feedbacks.

3 Assumptions

1. All clients are interested in live broadcast
   - No client is interested in frames broadcast before the client connected. If those frames also have to be transferred, the video will have to stored locally at the server.
   - Packets dropped while transmission don't have to be transmitted again.
2. All clients have the best possible data connection. The number of losses encountered while transmitting the payloads will increase as the bandwidth supported by the clients decreases.
   - Server tries to deliver as much data as possible to all clients (number of frames or video quality)
   - Parameters can be adjusted after receiving feedback from client

4 Architecture

The design of the implementation is chosen such that the overall processing overhead is minimized.

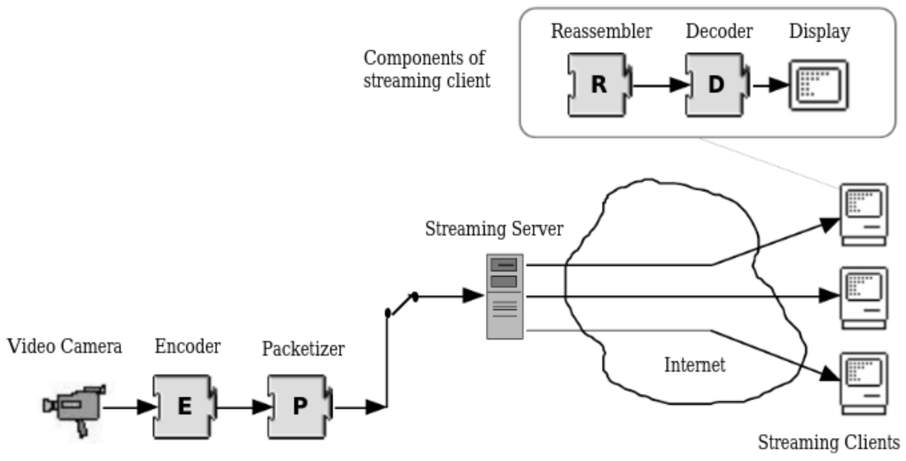

Figure 1: Overview of the implemented design

4.1 Server

The server is designed in accordance with a single process multi-threaded model. There are two types of threads involved -

- **Writer thread:** This thread is responsible for collecting frames from the webcam and writes them to a constant-sized list. There is only one instance of this thread active during runtime.

  An issue to be addressed here is the requirement data serialization before transmission so that the transmitted payloads can be successfully reconstructed on the receiving end. Our first attempt at this was to use python's serialization library `pickle` to serialize every generated frame and then transmit over the socket. However, this turned out to be extremely resource intensive, and had to be replaced with an alternative approach described below.

  The opencv's `VideoCapture` object reads frames from camera device. These frames are `numpy ndarrays` of dimensions `frame width * frame height * 3` of data type `uint8`. These can't be transferred to the client in this form. Various options are available for converting this to string and byte arrays. We use the `tobytes()` method to convert this to byte array. This can be converted back to numpy `ndarray` using the `fromstring()` method in numpy library. The hashed frame dimensions in the packet are used to reshape the byte array at the client side.

  For a rough estimate, the above approach reduced the CPU utilization on the host machine by approximately **20%**.

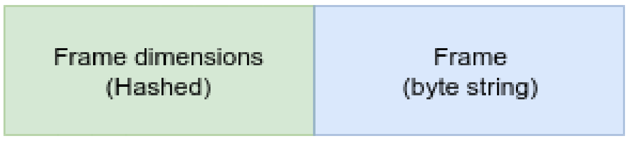

Figure 2: Structure of the payload transferred over socket

- **Reader thread:** This thread is responsible for maintaining a UDP connection with the client. Each client has a corresponding reader thread. It reads payloads (generated by the writer) from the list and transfers them to the client. The maximum number of concurrent readers active at a time are bounded by resource limits such as CPU, memory and bandwidth to enumerate a few.

Besides these, another thread handles the incoming connection requests and spawns reader threads accordingly. The constant-size list to maintain frame buffer is favorable because of memory constraints at the server's end.

4.1.1 The reader-writer model

The goal of this design is to have a fast writer and a (relatively) slow reader.

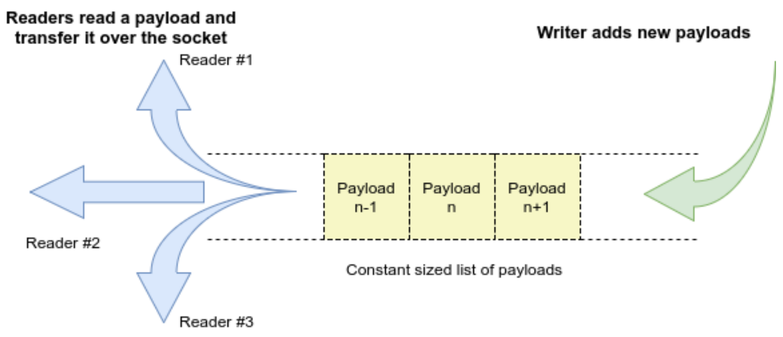
Readers read a payload and transfer it over the socket
Reader #1
Writer adds new payloads
Payload n-1
Payload n
Payload n+1
Reader #2
Constant sized list of payloads
Reader #3

Figure 3: The reader-writer model depicting 3 concurrent readers

4.1.2 Synchronizing a lagging reader

It might be possible that a reader is far behind the writer and transferring stale payloads. This is avoided by keeping track of the distance between the index that the reader is sending and the index to which the writer wrote last. As soon as the difference between these two indices increases beyond a threshold value `lag_threshold`(set to 5), the reader is synchronized with the writer. The downside of this is that a loss of `lag_threshold` number of payloads is encountered.

4.1.3 Space complexity

Since the reader is always behind the writer, the space occupied by the stale payloads which have been already transmitted can be reused. The current implementation uses a constant sized list, to which the writer writes until it reaches the last index, and then starts writing from the first index. As mentioned in the above section, a reader can never lag behind the writer beyond a specific threshold. This ensures that when the writer starts overwriting from the first index, none of the readers will encounter a loss of payloads.

4.2 Client

The client is implemented in accordance with a single process model. The client does a blocking read on the connected socket and retrieves the payload sent by the server. Each payload contains `frames_per_payload` number of frames. The value of `frames_per_payload` is chosen to reduce the overall socket operations required, while also reducing the size of each payload so that a loss of a payload will not cause a notable difference in the rendered stream.

5 Implementation environment

- Complete implementation is in Python.
- Two external libraries are utilized in building up this Project are listed below.
  - OpenCV (3.2.0)
  - Numpy
- Implemented the RTP protocol for video broadcasting and collected the various statistics to monitor the overall performance such as -
  - number of packet delivered
  - number of thread switches
  - number of lags occurred (caused when the reader is synchronized with the writer)

6 Limitations

1. RTP does not offer any guarantee on quality of service.
2. Video buffering on client side is not supported.
3. The maximum number of concurrent clients is bounded by the available resources.
4. Audio broadcasting is not supported.

7 Possible Extensions

- Video buffering on client side: This will users to go back and forth (to some extent) in the video stream.
- Experiments with various media encodings such as CELB, JPEG, nv, H261, MPV etc.

8 Summary

Video broadcast is a resource intensive service. A right balance has to be achieved between various things. Our current implementation uses a single process with multiple threads, which means only one CPU core can be used. This introduces a limitation on computation resource. The frame read from camera are huge in size. One raw frame in HD is approximately 2.76 MB in size. With such large frames, one cannot transfer more than 1 one of these every second. So, the server has to maintain a balance between using network bandwidth, providing a good frame rate for video, good resolution for frames and using computation resources for compressing data. If any compression algorithm is used, one has to ensure that the encoding and decoding can be done in real time.

Our objective included the following:

1. Implement remote video player for client, and corresponding server
   - RTP based server to stream videos
2. Concurrent clients
   - Handle multiple clients simultaneously
   - Different videos to different clients
3. Feedback-driven quality management
   - RTCP based parallel UDP connection to avoid congestion in network and ensure a better QoS

Our current implementation consists of a RTP based server to stream the camera feed to clients. Although, the server can handle multiple clients, but the clients don't have any list of videos to choose from. Along with the server, we have the client module which consists of a receiver and a video player.

The implementation is available at github.com/shivansh/CS425-Project.