Video Streaming and Synchronization through Multicast

Ryan Kwon '18
Williams College

Abstract

Video streaming has become a major feature of daily life, and a significant part of internet traffic. Limitations exist however. Video availability may be restricted based on geographic location, and content may be hidden by a paywall. Although both of those "limitations" are ameliorated by the prevalence of pirate websites, it does severely complicate watching a online video stream as a group where individuals may be in different locations. Not only is it somewhat difficult to synchronize viewtimes, but the streaming service provides no effort to ensure that all viewing members of the group stay synchronized. This paper attempts to resolve all of the above.

Specifically we present a design and early implementation for both a host-side video streaming tool that captures the host's desktop screen for streaming to multiple clients, as well as a client-side video player. The video player allows clients to communicate with the host, keeping all client playback positions consistent regardless of latency or geographic location.

1. INTRODUCTION

The particular use case we'd like to highlight are groups who would like to watch the same online video together. These groups may not be in the same geographic location, and so may be subject to regional restrictions on video content as well as network instability. In addition to regional restrictions, videos may also be set behind paywalls, preventing non-subscribed users from accessing video content directly.

In all of the above cases, we should intuitively be able to overcome these issues to allow synchronized sharing within the group. Network latency and instability can be managed so long as no user is actually isolated from the rest of the group, and if any user has access to a particular video, they should be able to route that video data to every other member in the group.

With the above in mind, we begin a discussion on relevant background in both video streaming as well as multicast networks in section 2. Section 3 will discuss our architecture and design at a high level. Section 4 will focus on our implementation, and section 5 will discuss evaluation.

2. BACKGROUND

2.1. Overview of Video Streaming Technology

The video streaming process can be reduced to six steps: capture, encode, transmit, receive, decode, and display [**1**]. We will briefly discuss relevant topics in encoding and decoding as well as transmission and reception.

Codecs and Compression

A video stream is a collection of frames, which are themselves individual images. Video compression works to take advantage of temporal redundancy, spatial redundancy, and by estimating motion between adjacent frames [**1**]. Conceptually we expect temporally close frames to hold similar objects (possibly in motion), and within each frame we expect pixels to be correlated. Ideally we send perceptually important data while reducing resources spent on perceptually irrelevant information.

In practice, there are three types of coded frames: intra-coded frames (I-frames), predictively coded frames (P-frames), and bi-directionally predicted frames (B-frames) [**1**]. I-frames are coded independently and serve as references, while P-frames are coded based on previously coded frames (for example, they may instead encode any differences from the previous frame). B-frames are coded using

both past and future frames.

A codec is compression system that comprises both an encoder and a decoder [**1**]. The encoder compresses a video source, while a decoder interprets the compressed data to reproduce the video. Critically however, standardization in codecs are fairly lacking. Standards (such as MPEG) only specify the format for representing the compressed data and the decoding process [**1**]. Implementation of the decoder is not standardized, nor is the encoding process standardized.

Transmission and Reception

Video streaming takes a compressed video, breaks it up into packets, and sends those packets to some end user (here referred to as the client). Some difficulties with effective video streaming include varying bandwidth and the normally high-bandwidth nature of video data; delay jitter, which are fluctuations in the end-to-end delay of packets; and losses, which are especially concerning given that P-frames propagate errors if an I-frame is lost or damaged.

Additionally, video streaming is time-sensitive in the sense that if a packet is not received by the time the client wishes to see it, we normally do not wish to wait to resend the missing packet. Consequently, video streams are normally implemented with UDP, typically with rate controls built into the application layer.

The bit rate needed for reasonable playback is dependent on the chosen video encoding (which determines how the video was compressed), consequently, three commong methods of modulating bandwidth usage are: transcoding, i.e. decoding and then re-encoding the selected video; multiple file switch, by which we offer multiple copies of the same video encoded at different bit rates; and scalable compression, which produces a set of ordered bitstreams where a subset produces the original video at a lower quality [**1**].

Delay jitter is handled by inserting a playback buffer, which relaxes constraints on when a particular set of frames must appear [**1**]. Lossy networks also have their own set of proposed solutions, but are less immediately relevant to this report and so will not be covered here.

2.2. Multicast and Related Work

Multicast is a third alternative to broadcast and unicast, offering a method of communication that is some-to-some, or one-to-some. IP multicast has multiple receivers connected to a single multicast address, creating a multicast group [**2**]. Routers which are part of this group create a distribution tree, which allows packets sent to the multicast group to be delivered to each recipient, creating copies of the data where the tree splits [**2**]. [1] Multicast ideally allows us to efficiently deliver high-bandwidth content to multiple destinations [**10**].

However, IP multicast is relatively murky to the author. It is unclear exactly what has been implemented, though multicast is reasonably popular among use-cases such as video conferencing. Our focus from now on will be on application-level multicast, which mimics the structure of multicast without necessarily using any unusual functionality in the IP level. At least some application-level multicast implementations have not been found to produce significant performance penalties [**14**], so we shall assume that most application-level implementations produce no significant overhead.

An interesting example implementation of application-level multicast using content-addressable networks was proposed in [**7**]. The authors create a graph, where individual nodes are responsible for disseminating data to neighbor node, replicating the functions of multicast. This approach is mostly mentioned because it's interesting, as their topology would greatly complicate our synchronization concerns.

Given the increasing popularity of online media, a number of methods have been developed to deliver this data. Real-time transfer protocol (RTP) provides a mechanism to deliver real-time data, but does not guarantee delivery or provide congestion controls, and is typically built on top of UDP [**4**]. Other protocols, such as RTCP (Real-time transfer control protocol) provide rate control [**4**]. Another popular solution to distributing video is dynamic adaptive streaming over HTTP (DASH), which breaks up the video into small HTTP downloads which individual clients can request statelessly from some known server [**9**].

Majumdar proposes an algorithm to better handle losses in wireless LANs, and evaluates it in the context of unicast and multicast video streaming data. Their algorithm combines forward error control (FEC) with a protocol called Automatic Repeat ReQuest (ARQ)[2] to create what they call Hy-

[1] There are different tree designs, the two most prominent being source trees (the source is the root, creating a tree through the network where the leaves are the other group members) and shared trees (there's a shared root called a rendevous point, where all incoming data is sent first).

[2] In ARQ, every packet is given some error check sequence.

brid ARQ, which they claim produces performance advantages in real-time video multicast and unicast [6]. They also briefly discuss MDFEC, which transcodes a prioritized multiresolution bitstream into one that lets decoded quality be a function of number of received packets, rather than which packets [6]. MDFEC has interesting implications for this project, as we do desire to minimize resource usage on the host while still providing best-possible video playback for each client.

As one of our concerns is bypassing paywalls and geographic restrictions, onion routing provides a mechanism for obfuscating traffic analysis. Specifically it works by wrapping data in layers of encryption, such that each hop between the source and the receiver is able to decrypt one layer, which lets them determine the next hop in the network, and thus send it forward [11].

3. ARCHITECTURE AND DESIGN

We have two primary goals for this system. We want the host to be able to send video data from an arbitrary online source, and we want every client connected to a particular host to be at the same position on the video at a given time (we want them to have synchronized playback).

3.1. Streaming an Arbitrary Online Video through the Host

Video data sent to the host, regardless of encryption or protocol, must be visible to the host. However, capturing this data directly is difficult, and any method may not generalize to all cases or stay relevant in the future.

Consequently, we aim to capture this video data indirectly. We use screen capture tools to take the video and audio of the host's machine, and stream that to our clients. Optimally, our video and audio capture is restricted to whatever application is receiving the online video, leaving the host free to do other activities.

3.2. Clients have Synchronized Playback

We have some number of clients who may not be geographically concentrated, and any single client may not have a stable network for the duration of the stream. At any given moment during the stream, we wish for the position of all clients in the stream to be within at most δ of any other client. The host may or may not be within δ of any client.

We'll examine client playback in two stages: the start of video streaming and the during the video streaming.

Synchronizing at the beginning of Video Streaming

The key concern here is that clients may receive the video stream at differing times due to differing latencies between individual clients and the host. Thus, an individual client may not know when to begin playback from when they first begin receiving the stream in order to stay synchronized with the other clients.

To resolve this, we do the following before we begin streaming:

1. For each client C_i, the host estimates that client's average latency. Call that client's latency, $latency_i$.
2. The host then computes $max\_latency = max(latency_1, ..., latency_n)$.
3. The host then informs every client C_i to have an initial buffer before beginning playback of

   $initial\_buffer_i = max\_latency - latency_i + extra\_buffer\_size$

   where $extra\_buffer\_size$ is some additional amount of buffering we do to minimize interruptions during streaming. In other words, when each client first receives the stream, they buffer for $initial\_buffer_i$ before beginning playback.

Figure 1 is a simple illustration of this.

Synchronizing during the Video Stream

During video streaming, a client's network may experience congestion (increasing latency) or, in the worst case, be unreachable from the host's network. Either will result in lost packets. There are two scenarios which may occur: the client does not notice, instead continuing playback from video stored in their buffer; or the outage lasts long enough that the client depletes their buffer, and has their video freeze.

In the former case, the client's buffer will still be missing frames, leading to artifacts or hiccups in playback. In the latter case, the client should rebuild their buffer before resuming playback (every

For every correct packet received, the receiver sends an acknowledgement to the sender. Furthermore, the sender automatically sends a packet again if it does not receive an acknowledgement within some time frame.

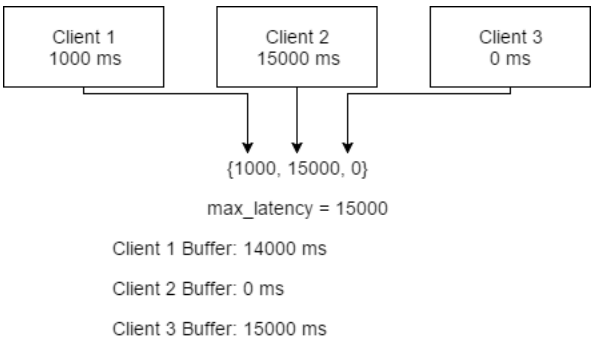

Figure 1. Initial buffer playback for 3 clients, with *extra_buffer_size* set to 0. Note that, if our latency estimates are still valid, every client will begin playing at the same time (15000 ms after streaming starts) though they begin receiving the stream at different times.

client's playback is behind what the host is streaming[3]) but the client may not rebuild their buffer correctly to account for changed network conditions, leading to future interruptions.

Guo and Ammar provide a nice framework for streaming video to clients from a single host to provide video playback with minimal interruptions [**13**]. We have the host provide sped-up time-shifted streams for when a client momentarily loses access to its video stream. When the client reconnects, they connect to the time-shifted stream set just before their disconnection, and they build up their buffer to catch up to the original stream, at which point they reconnect to the original stream. Every time-shifted stream streams as twice the rate of the original stream, allowing clients to catch up.

Their framework allows both clients with depleted buffers and partially incomplete buffers to regenerate and resume playback in sync with the other clients.

4. IMPLEMENTATION

In this application, two independent modules were needed. We needed some way to grab an arbitrary video source for streaming, and we needed a streaming tool with a nuanced enough API to allow us to maintain playback synchronization among clients. Additionally, we focused exclusively on tools with a Java API in order to take advantage of Java's portability.

4.1. Early Endeavors and Tools

Reinventing the Wheel

In order to grab an arbitrary video source, an initial thought was to provide a small browser, where the user would be able to specify a video player, and from which we could directly download snippets of video. Most sites however are extremely competent at preventing this sort of activity, so this route was quickly abandoned.

A similar route was attempted for video streaming, and while a large number of resources exist for custom streaming solutions[4], many of these were limited in some respect (unable to send audio, for example), and ultimately took a disproportional amount of time to develop relative to its value to the project. Additionally, there were concerns that a custom platform would be narrower and have more issues than a specialized published product.

Although these experiences were informative, we moved on to examining possible boxed solutions.

Xuggler

Xuggler appeared to be a promising start. It was a project which was mature at the beginning of this project, was made for Java, and had associations with Google. Unfortunately, by the time of this project Xuggler was no longer being maintained, and many of its relevant resources (videos and downloads included) have disappeared. Its .jar file is still archived in some sites, but Xuggler did not appear as tantalizing as it promised.

FFmpeg and ffplay

FFmpeg is a cross-platform application with a very powerful array of tools to stream, transcode, and record video and audio. Although no official Java API exists that is known to the author, a number of good wrappers exist that execute FFMPEG from Java code[5].

However, FFmpeg's system was fairly complicated, which slowed progress. Furthermore, ffplay, a separate media player that's nonetheless part of the larger FFmpeg "package" did not have an associated Java API. Early experiments to stream a

[3]What's meant here is that the host streams the video continuously. Therefore the host is at some position T, and it sends that data to all clients. Because every client has some latency, every client's playback position is offset by some amount. So every client is "behind" the host in video position. The only exception is when the client has a latency of 0 ms, in which case they play exactly what the host streams.

[4]A lab on video streaming by Dr. Mundur at UMBC is a good example of such a resource.

[5]The most popular example being Ffmpeg Java by Andrew Brampton, https://github.com/bramp/ffmpeg-cli-wrapper

downloaded sample video were also thwarted by FFmpeg's complexity[6], and the deprecation of ffserver, which provided some promising video and audio streaming support, ultimately induced the search for greener platforms.

VLC and VLCJ

VLC is a open source cross-platform solution for streaming, recording, transcoding, and playing media. Additionally, it has an official Java API called VLCJ, which exposes the most useful parts of the application in a simple manner, while also exposing lower level functions. Happily, their resources are also exceptionally informative and easy to follow, making the inclusion of VLCJ a fairly straightforward process.

Not long after adopting VLCJ, it was also determined that screen capture would be an easier way of capturing an arbitrary video source. VLCJ does provide tools for screen video and audio capture, but unfortunately an unresolvable issue with VLC and Windows prevents simultaneous video and audio capture. Workarounds do exist, but it is currently an ongoing experiment to integrate these workarounds with VLCJ. Modifying initial buffer size also seems to be possible, but has not yet been integrated into the project.

4.2. Overall Structure

We create two separate applications, one for the host and one for a client. The overall application process proceeds as described in figure 2.

In more detail:

1. We initialize the host, which specifies some multicast address to stream to. The host also creates a socket to accept new clients.

2. Individual clients then connect to the host's socket, and are informed what IP address and port number the video stream will be sent to.

3. The host pings all clients multiple times to estimate average latency in ms. We then compute *max_latency* as mentioned earlier in the architecture section, and compute the appropriate initial buffer time for each client. The host sends this initial buffer time as a response to the last ping.

4. The host begins screen capture, and also begins the video stream. Simultaneously, the host opens a new socket to receive relevant data from clients (frames per second experienced, position, metrics about the player, e.t.c.).

5. Each client receives the stream at different times, and they all begin playback synchronously. Each client independently sends updates to the host in order to let us evaluate the system.

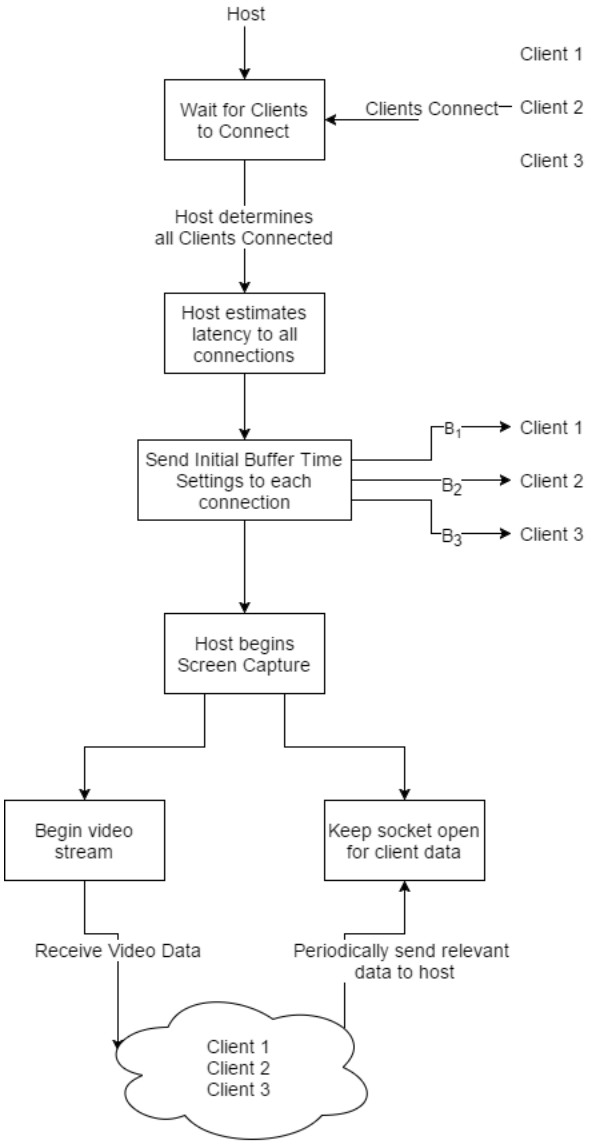

Figure 2. Flowchart for application

[6]Especially when it came to providing a stream descriptor, which normally would describe the format of the streamed data. Adding that descriptor through the Java API was frustrating, and ultimately abandoned in favor of other tools.

5. EVALUATION

Unfortunately, the client was unable to capture relevant video data. Although video playback was achieved by connecting to the multicast address, the provided API was not able to return non-null data to the client.[7]

However, information we would have liked to have were frames per second (FPS) to proxy video quality, video position in seconds, the IP address of the client to proxy their location, date and time the client sends this information, and date and time the host receives this information (to implicitly approximate latency). The above, in addition to brief conversations with participant clients would give us a meaningful early picture of performance and synchronization, as well as a better idea of what else to capture. As the API is exposed, in practice we also returned a variety of information about the client's player, including aspect ratio of the video, audio delay, window height and width, and video scale (which measures zoom).

6. CONCLUSIONS

A large number of components remain unfinished. Currently, the host is unable to stream both video and audio, the application captures the entire desktop screen (meaning the host cannot perform other tasks while streaming), and the client's media player is currently returning null data after receiving the video stream. Furthermore, time-shifted streams have not yet been implemented, and may require a custom screen capture solution.

Overall experiences with extant technology in this space has generally been very frustrating. Somewhat surprisingly given the ubiquitiousness of video and audio, many free tools are both very complicated and difficult to programmatically manipulate. It is without exaggeration that I can state that the vast majority of my time was spent wrestling with various tools in order to accomplish some milestone.

Although the implications of this project are not groundbreaking, as it relies mostly on combining a number of existing technologies into a single process, it remains interesting. There are curious implications for the larger video streaming market, especially as nothing can firmly be kept private. Furthermore, my experiences in this space have also revealed a fairly significant level of technical acuity necessary to make video applications, and although VLC and VLCJ are extremely competent in lowering this frustration-threshold, both do have limitations and sometimes (in the case of my clients fetching no useful information from their player) fail mysteriously.

[7]The host was successfully able to use the same API to collect relevant data. It is currently unknown why the client failed.

ACKNOWLEDGMENT

I would like to thank Jeannie Albrecht of Williams College, for her steadfast advice and oversight, as well as for sponsoring this particular project. She's the best.

References

[1] J. Apostolopoulos, W. Tan, and S. Wee, "Video Streaming: Concepts, Algorithms, and Systems," Hewlett-Packard Company, 2002.

[2] "IP Multicast Technology Overview," Cisco, San Jose, CA, 2001.

[3] A. Rodriguez and K. Morse, "Evaluating Video Codecs," Kaleida Labs, 1994.

[4] A. Basso, G. Cash, and M. Civanlar, "Real-time MPEG-2 delivery based on RTP: Implementation issues," *Elsevier Science B. V.*, 1999.

[5] R. Mok, E. Chan, and R. Chang, "Measuring the Quality of Experience of HTTP Video Streaming," Hong Kong Polytechnic University.

[6] A. Majumdar, D. Sachs, I. Kozintsev, K. Ramchandran, M. Yeung, "Multicast and Unicast Real-Time Video Streming Over Wireless LANs," *IEEE Trans. Circuits Syst. Video Technol.* vol. 12, pp. 524-534, June 2002.

[7] Ratnasamy, Sylvia, Mark Handley, Richard Karp, and Scott Shenker. "Application-level multicast using content-addressable networks." In International Workshop on Networked Group Communication, pp. 14-29. Springer Berlin Heidelberg, 2001.

[8] Stockhammer, Thomas. "Dynamic adaptive streaming over HTTP–: standards and design principles." In Proceedings of the second annual ACM conference on Multimedia systems, pp. 133-144. ACM, 2011.

[9] Lakshman, T. V., and Upamanyu Madhow. "The performance of TCP/IP for networks with high bandwidth-delay products and random loss." IEEE/ACM Transactions on Networking (ToN) 5, no. 3 (1997): 336-350.

[10] Deering, Stephen E., and David R. Cheriton. "Multicast routing in datagram internetworks and extended LANs." ACM Transactions on Computer Systems (TOCS) 8, no. 2 (1990): 85-110.

[11] Goldschlag, David, Michael Reed, and Paul Syverson. "Hiding routing information." In Information Hiding, pp. 137-150. Springer Berlin/Heidelberg, 1996.

[12] McCanne, Steven Ray, and Martin Vetterli. Scalable compression and transmission of internet multicast video. University of California, Berkeley, 1996.

[13] M. Guo and M. Ammar, "Scalable live video streaming to cooperative clients using time shifting and video patching," IEEE 2004.

[14] Chu, Yang-hua, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. "A case for end system multicast." IEEE Journal on selected areas in communications 20, no. 8 (2002): 1456-1471.