Lecture 17: Video streaming

Anirudh Sivaraman

2020/11/23

1 Overview of the next few lectures

Starting this lecture, we'll talk about some more recent developments in networking, which are less than about 15 years old.

1. We'll spend two lectures on two new classes of applications that have taken off on the Internet: video streaming and peer-to-peer applications.
2. We'll then spend one lecture talking about *datacenter networks*. A datacenter network consists of tens of thousands of server machines interconnected by a low-latency and high-capacity network at a single geographic site that spans a few football fields. These networks are owned and operated by companies such as Google, Amazon, Microsoft, and Facebook, which use these large networks to offer services to millions of users over the Web.
3. We'll spend one lecture talking about programmable networks: the ability to change the functionality of a network's routers or switches in the same way that we can change the functionality of end hosts today.

Unlike the course so far, which was structured around specific themes (e.g., physical layer or network security), the remaining material will be significantly more unstructured. This is because the topics themselves are new, the right way to teach them hasn't yet emerged, and I'll be relying on primary sources (such as peer-reviewed papers) instead of textbooks to develop the lecture material. On the other hand, they hopefully convey to you the excitement of being in a continuously evolving field, which you can contribute to during your own careers.

2 The stored-video streaming problem

In today's lecture, we'll talk about the problem solved by YouTube, Netflix, and Hulu. We are going to mostly be talking about *stored-video streaming*, where there is a video stored as a file on a server somewhere (e.g., the YouTube server), which needs to be sent to a client (e.g., your web browser or a mobile phone app). Towards the end of the lecture, we'll talk about live-video streaming, where a live video (e.g., a concert) is transmitted to viewers in real time.

First, as background for the stored-video streaming problem, we'll describe how videos are stored on a server in a space-efficient manner. Second, we'll describe a few simple strawman designs for stored-video streaming. Third, we'll compare these strawman designs to extract a user's requirements for stored-video streaming. These requirements often conflict. Hence, a real stored-video streaming system needs to compromise between different conflicting requirements. Fourth, we'll compare stored-video streaming to two related video-based systems on the Internet: video conferencing and live-video streaming.

3 Storing videos efficiently

A video is a sequence of images, called frames, at a certain frame rate, e.g., 24 frames per second. When a video is first captured by a device such as a webcam or a video camcorder, the frames are fairly large in size.

Each frame of video captured by the device is called a raw frame. To first approximation, a raw frame is a 2D array of pixels. For instance, the Ultra High Definition Telivision (UHDTV) standard, informally called 4K video, represents raw frames as a 3840*2160 array.[1] Similarly, the Full HD standard represents raw frames as a 1920*1080 array.

Raw frames are *uncompressed*: the light intensity at each pixel is independently represented using a certain number of bits depending on how this light intensity is encoded. A simple method is to represent the red, green, and blue components of the light intensity with a separate 8 bit number, though more sophisticated methods are used in practice today.

Notably, raw frames do not take advantage of two forms of redundancy in a video: intra-frame redundancy and inter-frame redundancy. Intra-frame redundancy occurs when there are large portions of a frame that are similar in light intensity, e.g., a frame showing a white wall. Inter-frame redundancy occurs when consecutive frames in a video are similar to each other, e.g., a relatively slow moving portion of a movie where a person moves from one side of the room to another, but the room itself remains unchanged across frames.

Taking advantange of these redundancies results in *compressed* videos where one part of the video can be predicted from another. In intra-frame compression, one portion of a frame (say B) can be predicted from another portion (say A) of the same frame, implying that only the light intensity values for A need to be stored. Similarly, in inter-frame compression, one portion of a frame can be predicted from another portion of a *different* frame, again resulting in storage savings.

These storage savings can be significant. For instance, in a 4K video, the raw frame can be around 11 megabytes. An *intra-coded frame*, which uses intra-frame compression alone, is around 1 megabyte. An *inter-coded frame*, which can use both intra-frame and inter-frame compression, is even smaller at around 10–30 kilobytes [1].

Based on the discussion above, it might seem like we should only use inter-coded frames for all frames but the first frame.[2] Why do we need intra-coded frames at all?

First, the redundancy between two consecutive frames might be so low that an inter-coded frame does not significantly decrease the frame size relative to just starting afresh and using an intra-coded frame. Second, even if there is significant inter-frame redundancy, it is useful to periodically insert intra-coded frames. This is to allow a user playing the compressed video to seek to an arbitrary time within the video.

Without a periodic intra frame, the user might have to decompress (the inverse process of compression) every frame starting from the first one to seek to (say) the 10000th frame[3] in the video. This can take a long time, especially with larger 4K videos. Instead, having a periodic intra-coded frame allows the user to start decoding from the intra-coded frame closest to the seek point. This is because the intra-coded frame can be decoded without access to any of the previous frames because it doesn't depend on any frame before it. Systems like YouTube insert an intra-coded frame every 4–5 seconds [1]. We'll call a combination of an intra-coded frame and a set of inter-code frames, until the next intra-coded frame, a *chunk*.

With all of this background out of the way, let's describe what happens when you upload a video that you captured to YouTube. You first capture the video using a device like a webcam. Then, your video recording software transforms the raw frames from the webcam into a compressed video employing both intra-coded and inter-coded frames. Your video recording software likely has some default settings for the quality of the video, which is most directly indicated by the number of pixels in the 2D array we mentioned and the frame rate in frames per second. So, by this definition, UHDTV has higher quality than Full HD.

The video you upload to YouTube is likely of a fairly high quality. To store the video, YouTube decompresses the video and compresses it again into several different quality levels that correspond to different number of pixels in the pixel array. This decompress+recompress process is called transcoding. As you can expect, a higher quality video needs more bits per frame, and having a range of quality levels allows YouTube to serve your video to a range of clients with widely different network capacities. You can see the different quality levels available for a video using the settings button at the right bottom corner of a YouTube video. The quality levels are labeled by a number ranging from 144 pixels to 2160 pixels. This number reflects the number of pixels in one

[1] There are a few different standards, including UHDTV, which are all grouped into the informal term 4K.
[2] The first frame has to be intra-coded by definition because there is no previous frame.
[3] This is roughly 416 seconds in a 24 frame per second video.

dimension of the 2D array, typically the shorter dimension, e.g., 144p refers to a 144*256 array, while 1080p refers to a 1080*1920 array.

Once the stored-video streaming server has created multiple different quality levels for a video, the *stored-video streaming problem* is the problem of dynamically selecting what quality level to use for a client depending on the network conditions at the client's end. These network conditions can vary from one client to another depending on where the client is geographically located and can also vary within a single client's video session because the client moves from (say) WiFi to LTE.

4 Some strawman designs for stored-video streaming

Next, we'll discuss some strawman designs for the stored-video streaming problem assuming video is stored at different quality levels at the stored-video streaming server. Each of these strawman designs has some pros and some cons and analyzing these designs will help us understand what it is that users really care about in stored-video streaming. A real stored-video streaming system picks some compromise between multiple conflicting user requirements—or more ambitiously, lets the user pick their own compromise.

The simplest solution to the stored-video streaming problem is to treat it like downloading a file. The client first downloads the entire video at the highest quality to their local disk. Once the file has downloaded, the client plays it using their video player. The pro of this approach is that the user gets the highest quality video. However, the user needs to wait till the entire video is downloaded, which isn't particularly feasible for a long video like a two-hour movie. Furthermore, the user may not even watch the entire video and lose interest in between. From the video provider's perspective, this represents wasted egress network capacity out of the streaming server that could have been more gainfully used for another client.

The second strawman design for the stored-video streaming problem is to send the lowest quality video to the client one chunk at a time. Then, at the client, we overlap the playback of downloaded chunks with the download of new chunks. In other words, when the client is watching a particular chunk, subsequent chunks in the video are being downloaded from the server. This design implies that the user needs to wait only till the first chunk is downloaded, which can be pretty quick because it is only a few seconds of video at the lowest quality level. The benefit of this approach is low startup delay for the client and the ability to stop sending video for the server if the client disappears or pauses playback. The downside is that the video quality is quite poor.

For the third strawman, let's modify the second strawman to send the highest quality video that the client's network can support. How do we determine this highest quality video? For simplicity, let's assume the client's total available capacity to the server for stored-video streaming is C bits/second. Let's assume that for each quality level L of the video, the stored-video streaming server provides us an estimate of the number of bits per second that it would take to transmit video at that quality level (C_L). Then, we want to pick the highest level L such that $C_L < C$.

Why is this? This is because C_L is the number of bits per *second of video* in the video stream, i.e., the ratio of the video's file size to its duration. Then C/C_L represents the *number of seconds of video downloaded every second* at the client. The video player is playing one second of video every second, assuming you are not speeding up playback. Then so long as C/C_L is greater than 1, the video player will not starve for content to play back. This translates to the requirement $C_L < C$.

In summary, for strawman 3, we want to pick the highest L such that $C_L < C$. Any higher and the video player will starve; any lower means that we can do better. While this works in theory, the problem in practice is that C is unknown and can change over time as the client's network conditions change. If C improves this is not a concern, but if C decreases, the video player will starve for content, leading to the spinning wheel that many of us are familiar with on YouTube. YouTube's spinning wheel is an indication that the video player has exhausted all the video content that it can play back and is waiting for more content to be delivered over the network. Hence, the problem with strawman 3 in scenarios of varying network conditions is the presence of *rebuffering* or *buffer stall* events that can annoy the user.

For strawman 4, we can use strawman 3 until there is a rebuffering event. At this point, we can switch to a lower video quality until C improves again. But time-varying video quality isn't particularly pleasant either. It may be preferable to watch a consistently medium-quality video rather than a video that oscillates between the

highest and lowest video quality levels.

5 User requirements in stored-video streaming

Our discussion of the 4 strawman designs in the previous section should have given you a sense of what users actually care about in stored-video streaming. They care about at least four things and maybe more.

1. Startup delay: How long does the user have to wait before they can play even the first frame of the video?
2. Average video quality: What was the video quality of the video watched by the user averaged over the duration of the video session?
3. Rebuffering events: How many times (and for how long each time) did the playback stall because the video player no longer had any content to play?
4. Video quality variations: How often and how much did the video quality change over the course of the video playback?

These requirements are in conflict with each other. Minimizing startup delay would call for downloading low quality video (strawman 2), which conflicts with requirement 2. Similarly, switching between video quality levels as strawman 4 does may give you high average video quality at the cost of a non-uniform video experience.

Video streaming systems balance these four concerns by picking some compromise between them and are more involved than the four strawman designs described here. As a result, we won't describe the algorithms within these systems here, but only discuss the compromises made by two specific systems here.

In some cases, the compromise is pretty steep. For instance, the Buffer-Based Algorithm [2] explicitly tries to maximize average video quality and minimize the number of rebuffering events. It makes no mention of the other two goals. On the other hand, MPC [3] defines a quality of experience metric (QoE) that uses a weighted combination of the four goals defined here, where the weights could in theory be set by the user.

6 Differences from video conferencing and live video streaming

To summarize, we'll briefly compare stored-video streaming with two other video-based systems in widespread use on the Internet, especially in a post-pandemic world: video conferencing and live video streaming.

Live video streaming can be treated using the same techniques in stored-video streaming if a sufficiently long latency is imposed between when the event happens in the real world and when it is available for viewing on a video player, e.g., a 10 second delay between when the home run was hit in the ballpark and when you see it on your computer. With this sufficently long latency, the live video can be uploaded onto a streaming server as it is being captured, which then stores the video and streams out the video to clients as described earlier. Of the four strawman designs, strawman 1 alone is ruled out because the entire video is never available at any given point in time (it wouldn't be live if so). This means the only possible solutions need to send video one chunk at time.

The biggest difference between stored-video streaming and video conferencing is that video conferencing is interactive. What this means is that the latency between when the video frame is generated at one end of the video conference and when it is viewed at the other end of the video conference is crucial. Otherwise, you have the following scenario: A says something at time t and expects that B will get it at a time $t + \delta$ later, where δ is small. If B does not receive it by then, A will assume something went wrong and repeat what was said. Meanwhile, B probably hears the earlier version of the statement and responds, all in all leading to a very confusing user experience, commonly called lag. This frame latency requirement differentiates video conferencing from streaming.

In stored-video streaming, the frames are all available at time 0 and the video streaming system can always give the network a "head start" over the user's video playback by increasing the startup delay and downloading a large number of frames at time 0. A limiting case of the head start strategy is strawman 1, which sends down the entire video so that the video playback is always behind what the network has downloaded. This is not possible in video conferencing because the frames are not even all available at time 0.

References

[1] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*, 2017.

[2] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *SIGCOMM*, 2014.

[3] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *SIGCOMM*, 2015.