8. What Makes a Download a Stream?

Frank Bauer and Philipp Kurth

Introduction

There is a vibrant discussion about streaming, data transport through the cloud and various compression technologies for audiovisual media. However, there is one startling detail missing in the debate: when is a download a stream, and why do container formats matter? At first glance, this might seem like a strange comparison because it seems obvious: streamed media is found above all on Netflix and other streaming platforms, while video downloads are familiar from private video sharing, stock media sites (like Shutterstock) or illegal portals. Nevertheless, the truth is more differentiated. Moreover, the term download is a complicating factor. When we use the term streaming, we are usually referring to networked streaming, for example, over the internet. Yet it is worth taking into account the entire lifetime of a video from its inception (e.g., when captured by a camera) to its end user delivery. In particular, it is notable that video data is streamed from the very beginning, before any networks get involved. When a camera sensor records the image data, information is continuously streamed over an internal bus to the video encoder hardware and the storage device. The resulting file is not generally in the optimal format for either download or network stream, and will thus be converted into an appropriate format. In this article, we will illustrate the fundamental technical aspects of video compression and the role that container formats play. We will use the latter to argue how the structure of the file can hint towards the intended use, i.e., when a downloaded file can be regarded as a streaming medium. However, first we need to define the meaning of "download" for this text.

By Definition a Download

We define a download in the context of a network connection. Downloading is the process of receiving data from a remote machine. The opposite is an upload, whereby the data is sent to a remote machine. We will often call the remote machine a Server or a Host (as the machine is hosting the data we want to download) and the receiving machine a Client.

Since typical streaming media are not stored on a local machine but on a remote system (like the Netflix servers, or more precisely a Content Distribution Network (CDN) that distributes the content for Netflix), it is by definition a download. Streaming is often deemed a particular form of download. The differentiating factor commonly used is the volatility of the data on the consumer's machine. A downloaded file is permanent, while the data sent during streaming is transient and automatically deleted as soon as it has been consumed or the playback stopped.

The distinction is also often cited in legal arguments, as either download or stream has very different implications on liability in Europe. One long-standing interpretation is this: consider an illegally published video file. If a user accidentally downloaded the file, that user would be liable for downloading it. If a viewer accidentally streamed that same file, the person hosting the content would be accountable for any infringement. This distinction was made in a particularly dicey German case, where thousands of internet users were sent a cease and desist letter with fine for allegedly downloading illegally published pornographic content (Solmecke 2013). The sending law firm classified the consumed media as downloads, while defenses often argued it was streaming. Some declared that it was a progressive download (middle ground between a download and a stream), whereby you can watch the content while it is being loaded. During the download, the data is temporarily stored in the browser's cache,[1] where it will remain even when playback has finished. Caches are cleared eventually, and, on iOS or Android, they are not even accessible to the user. Nonetheless, the fact that the media remained on the user's machine even after the playback had ended caused some legal representatives to classify this behavior as downloading and not streaming. Details like this complicate the arguments and require in-depth technical knowledge to find appropriate rulings in court. It is important to note that the judgment of the courts differs widely on the interpretation of Article 5 of the EU-Directive 2001/29/EC concerning streaming media (Hugenholtz 2018).

Hence, applying this volatile definition of streaming is dangerous. As we have described, most modern browsers allow users to watch downloadable media while the download advances, in some cases even without storing the files in the users' download folder, but instead in temporary, internal locations that (depending on browser settings) will be erased once the session is closed. In contrast to a stream, such progressive downloads are not customizable (i.e., in language or quality selection) and deliver the exact same content to all users.

Some tools can convert streaming media (for example, from YouTube) and assemble it into a working, local video file. When considering this aspect, we could argue that the classification of download versus stream is not an inherent property of the distributed data but something that depends on the tools a consumer uses.

This assumption renders a label like “streaming platform” invalid, as the platform (the remote site hosting the video data) can never determine whether the delivered files are a stream or a download.

There is a more straightforward way to distinguish streaming media from downloadable content, which is based solely on the published data. This approach could help to overcome the ambiguities of the previously introduced definitions. To apply it, we need to understand networked streaming and dig into the depths of codecs and containers.

Codecs and Containers—A Primer

In our experience, many people confuse a video codec with the container format. However, the two serve different purposes. The video codec determines the way content is compressed, while the container determines the arrangement of that compressed data, for instance within a file. This distinction is essential for our discussion. We want to attribute the label streaming or download based on the properties of a file. As we will see, the decision is mostly informed by the container format used and not the codec.

A video (compression) codec is something like Motion-Jpeg (MJPG), MPEG-2, H.264, H.265 (also called HEVC), the upcoming H.266 (also called VVC), ProRes, RedCode or ARRIRAW. MPEG-2 and the H.26x-series are continuous improvements of the same principle of compression focused on end-user consumption. ProRes, RedCode and ARRIRAW are compressions for a different kind of market and mainly target the production cycle of a video. Compression is an essential aspect of networked video. Generally, the amount of data for a video is enormous. A single frame of uncompressed 4K, UHD (3840 x 2160 pixels, 10-Bit color depth per channel, no chroma subsampling, no audio) video amounts to about 60 MB in storage. A 90-minute movie (without sound) would result in a 5 TB file. For comparison, a Blu-ray can store up to 50 GB (=0.05 TB), which includes audio as well. This amount of data is not playable in real-time over a typical consumer network. With a fast 1000 MBit/s network, the download of that video would take about 10 hours to transfer. However, the average bandwidth is lower, especially on mobile networks, where a vast amount of video is consumed. According to Fenwick and Khatri (2020), the average German download bandwidth in 2020 on a mobile connection was only 30 MBit/s, increasing the download time to about 15 days. In that same report, the users gave Germany a video experience score of 74 out of 100 points. That disconnect (an average download time of 15 days versus four-star satisfaction) underlines the importance of compression for practical use of video data and is the necessary foundation for the rise of YouTube, Netflix and other streaming service providers.

Still, why do we need containers? Container formats combine and arrange the compressed video data with audio, subtitles and other information. Without containers, we would not be able to consume compressed video at all. Well known representatives are QuickTime (.mov), Matroska (.mkv) (Matroska 2020), Audio Video Interleave (.avi) or the creatively named MPEG-4 Part 14, better recognized as mp4.

A video encoder will usually convert the video from an input format into the desired output. Since most video compression is not lossless, this step will negatively impact the quality of the video. Non-professional users often apply this recompression without need, due to the confusion between container and codec. For example, a video is stored in a Matroska format, but the editing software cannot import it. Users will then use tools like FFmpeg or Handbrake to convert the video into a supported format, let us assume QuickTime. Default settings of that tool will recompress the video and store it in the new container. Though, if the video was initially compressed with H.264, for instance, the recompression was unnecessary, as both containers (Matroska and QuickTime) support H.264 compressed video. The right conversion is re-muxing, basically repackaging the video data from Matroska to QuickTime, as this would not change the video quality.

It is crucial to know that containers and codecs have a symbiotic relationship. A regular video codec applies some kind of compression scheme to the frames of a video. Typically, the output of a codec is self-sustained, sequential packages of frame data (not an entire video). But the compressed video data generated by a codec is useless without a container. Container formats, on the other hand, know how codecs generate packages and can use that information to structure the data within a file. This fact allows us to change the container format (which is just a management data structure) without recompressing the video (which would change the codec, triggering a recompression and degrading quality).

The Principles of Networked Streaming

As we have seen, it is a challenge to find a good definition of a download. Since we aim to distinguish it from a stream, it is instructive to take a close look first at a typical streaming experience, as well as the technical challenges that are handled by the streaming service to provide it. We have chosen one that is probably most familiar—end user streaming.

The User's Perspective

Consider a consumer streaming their favorite show over a streaming platform such as Netflix, HBO or Amazon Prime Video. Users, in general, have different ways at their disposal to consume the content offered. They can choose between a browser, desktop or mobile app to watch the content. For the sake of this example, we will assume a mobile app on an iPhone 6s[2] will play the video.

1. The user selects an episode that has been partly watched on the desktop app before.
2. After a brief moment, the playback resumes at the last watched point, with the previously selected language and subtitle settings.
3. Because the viewer had slept through the last minutes, they rewind the video to catch up. Shortly after, the playback resumes from the rewound position.
4. A couple of seconds later, the video and audio automatically adopt a higher definition.
5. As that episode finishes, the next episode starts.
6. The user chooses to skip the show's intro scene, causing the video to jump forward.

Behind the Scenes

At the backend, the place that will deliver the content to the viewer, the same interaction looks like this.

1. An authenticated user of the service sends a request to continue the last played episode of a show. Querying the database for the user and the selected content reveals the last frame that was played by the consumer's device. The returned information also includes the playback application and device, as well as language and subtitle preferences. The last video was delivered with 5.1 sound as 4K UHD content compressed with H.265. However, the current request originates from a mobile app installed on an iPhone 6s. That hardware contains specialized decoder hardware for H.264 video only. While the quality and size of H.265 would be better, the energy profile of the app is worse, resulting in higher battery drain and less playback time. For this reason, the server will select an H.264 format for delivery. A similar choice is made for the audio content, as a 5.1 signal often wastes capacity. Consequently, stereo audio is selected. Since there is no information on the currently available bandwidth[3] for the mobile device, an average quality setting is applied.
2. The server assembles all the data needed with the correct quality and starts to send it to the client for playback. The system anticipates the next frame that will be requested by the player and preloads the data into a temporary cache. During playback, the client app continuously requests the next frames for playback. Those are delivered and deleted from the server cache. This deletion, in turn, triggers the preload of the next frames, and so on.

3. After a while, the client app requests an older frame that was not anticipated. The system clears the cache, reads the requested data from disc and sends it to the client.
4. Server and player app both monitor the time it takes for the data to arrive at the client side. As it is evident that the bandwidth can handle a higher quality stream, the currently loaded frame data in the server's cache is replaced with a higher quality version of the same moment in the video. From this point forward, all frames are delivered with the higher quality settings.
5. Later, the last frames of the video are delivered to the application. The system will also send a notification with those frames suggesting the following auto-played content to the player application. At the same time, the playback data for that video is preloaded and stored into the system cache. A short while later, the player requests the suggested content, which is met with the delivery of the cached data. Since a new episode has started, the system sends a notification that informs the player of the end position of the show's intro sequence.
6. Another brief moment later, that playback position is requested by the player. This forward seek forces the system to clear the cache, load the corresponding data for delivery, and continue the delivery from there.

Is It a Stream?

We have called this chapter "The principles of networked streaming," which implies that we have described a stream and not a download. But why is that? We think this example represents the prototypical streaming experience and exposes two features that are unique to streaming:

- The ability to move through the video and start at an arbitrary position without any discernible delay.
- The ability to adjust the quality (and content, like the audio language) on the fly.

The first ability in particular may be surprising, as a downloaded file can be started and repositioned to arbitrary locations within the video, too. The difference is that this feature can only be offered once the download file is fully transferred over the network. When played directly over a network, playback can often only start at the very beginning of the download file or after a significant portion of the file has been transferred. The source location of the file is an important distinction. In this paper, we assume that the files reside on a network and not a local disc.

The second property, arbitrary quality selection, is more prominent. A downloadable file usually provides a single video stored in a defined codec (like H.265) and quality. If it is at all possible to influence the codec or quality, the selection will take place at download time by choosing different files. Potentially, this constraint has some negative consequences when buying content on one device and consuming it on another. Let us assume the purchase was completed on a Desktop PC. The storefront that sells the video determines that the best format is a maximum quality video encoded with H.265 and sends the corresponding download link to the customer. When that customer chooses to use the iPhone 6s mentioned above for playback, the codec will waste battery life as it is not hardware accelerated.

This effect is also very prominent for older home-made videos in a user's library or older content from portals like Instagram, YouTube or Twitch. When the videos were new, the device (for example an iPhone 6s) or the video provider stored the content in the most efficient way, in that case H.264 compressed. However, later, when new hardware and codecs arrive, the same file is no longer optimized. In that case, playing it back on new hardware (that has H.265 acceleration but not H.264) can negatively affect the battery life. While current generation hardware still has a H.264 decoder along with new hardware for H.265, it is mostly just a matter of time until that outdated part of the hardware is removed.

While obsolescence is a problem on streaming platforms as well, they are better equipped to handle the initial example, whereby the content is bought and consumed on different hardware. A streaming platform can deliver a different, adaptive version of the same content. Both users and the service provider can thus adjust the playback quality to the capabilities of the hardware or the available network capacity. In 2020, during the Corona Crisis, many of Europe's employees worked from home and saturated the internet bandwidth with teleconferences, another form of live video streaming. During this time, Netflix, YouTube, Amazon and most other large streaming providers reduced the maximum quality of the content delivered for streaming (BBC 2020). It was possible to implement this change in a short amount of time because of the existing infrastructure for adaptive streaming quality. Providers simply did not offer higher bandwidth versions of their content. Without this prerequisite, all videos would have had to be recompressed in order to lower the consumed bandwidth.

This adaptivity requires a streaming platform to store multiple versions of the same video. But how does the server switch that video (and audio) mid-stream? Moreover, we have seen that the server does not deliver the entire file, but seems to have packages at its disposal that get transferred sequentially. To explain, we need to take a look at the inner workings of a container.

Container or File Formats

As we have mentioned briefly, there are different compression algorithms (also called video codecs) for video data like MPEG-2, H.264, H.265 or H.266. Please note that these only apply to video, not to subtitles, audio or other metadata tracks. An encoder compresses the video into a bitstream representing the final data. However, there is no H.264 file format. That video bitstream is generally stored in a container format (typically mp4).

Most compressed or aggregated files rely on containers. TIFF, for example, is a container format for image data. It supports various compression algorithms such as Deflate, LZW or even JPEG.[4] The TIFF container specifies the compression format and some metadata about the image (i.e. size or bit depth) as well as the compressed data. We call TIFF a container as it supports a variety of compressions and multiple images within the same file, such as a preview image, compressed with JPEG, and the actual raw data using LZW compression.

Another widely used container format is ZIP. The files stored inside a .zip file are compressed with an algorithm—typically Deflate (Deutsch 1996; for available methods see pkware 2012). The container additionally stores the file metadata (like names, size, type), the file hierarchy (or folder structure), and the applied compression method.

Tracks

Like ZIP or TIFF, video containers combine several different types of data (often attributed the term tracks or streams) in one file. Commonly, these are the video track and one or more audio tracks, as well as subtitles and control or metadata tracks. Before we come back to our base observation (download versus stream), we will take a brief look at the different track types usually present within a container (see Fig. 8.3 for a sample of tracks stored for a typical consumer video).

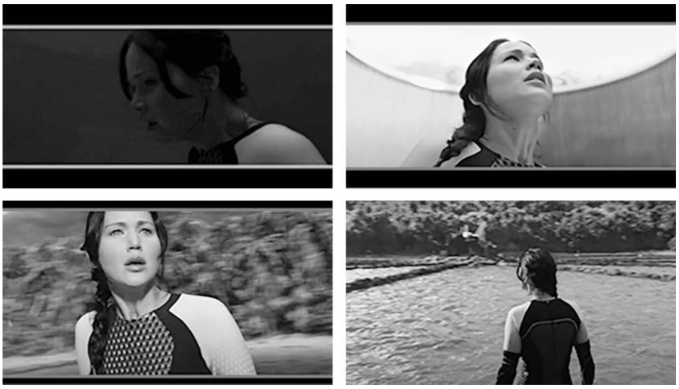


Figure 8.1: *The Hunger Games: Catching Fire*. Transforming the frame from CinemaScope to IMAX.

Control Tracks are not typical for consumer media but are used in theatres to automate the curtain or live effects like smoke, lights or rumble. When movies like *The Hunger Games: Catching Fire* (Francis Lawrence, 2013) change the aspect ratio of the image (in this case from CinemaScope to IMAX, see Fig. 8.1) right in the middle, a home theatre version will simply change the height of the image or the size of the letterbox. However, in cinema, the change in aspect ratio often results in a wider or, in this case, higher image, which necessitates the curtains to draw open mid-movie. The screener or other staff do not control this; instead, it is the control track.

Metadata Tracks are another type of data that is uncommon in consumer media. In production, data like camera settings, lenses and focus are vital and an integral part of a so-called pro workflow; these are included in metadata tracks.

Text Tracks are common in consumer media. They typically store subtitles and most containers support multiple subtitle tracks. Depending on the specific container format, there may be support for compressing the text track. Compared to audio or video, the size of text tracks is negligible.

Audio Tracks, along with the video, consume the most space, in some cases (depending on the compression and the number of audio channels and sampling rate) even more than the actual image signal. Containers can typically store multiple audio tracks for the same file, either for localization or different formats like DTS-HD, Dolby Digital or a Stereo Downmix. There are also different supported audio compression formats like mp3, AAC, or even lossless compressed PCM-data.

The Video Track is the final track type. Most containers support multiple video streams as well; nevertheless, this feature is rarely used, as in most cases the video is considered the primary track, which is augmented by audio, subtitles and other information.

Download Versus Stream

It may be surprising, but this separation into single, autonomous tracks is the first property that helps us to distinguish the behavior of a download from that of a stream.

When transferring the file over the network, a stream only contains the tracks that are currently in use (the selected language and appropriate codec settings for the hardware platform) and not all the tracks available. This recombination is ordinarily achieved by re-muxing the tracks on the fly on the streaming server. In contrast, a download usually includes multiple tracks, at least for audio or subtitles.

Distributors often want to include multiple languages for audio, or multiple video qualities and codecs to cover a wide range of audiences. This diversity was established on DVD and later Blu-ray, as they were not marketed to single countries, but—amongst others—to the whole of Europe. While economically not as advantageous as in times when movies were shared over physical discs, this behavior persists in the "cloud age" of content distribution, as consumers have got used to having, amongst others, the native audio track. As with discs, the container file for a purchased download needs to package all the offered content when the file is transferred to the customer. This packaging of all the available content obviously increases the file size. Even if an individual consumer will never listen to the French audio track, it needs to be present for a one-size-fits-all solution, as there is no chance to adaptively change the language otherwise. In the case of a container, this means that the file includes all tracks, and the player software selects the one that corresponds to the viewer's preferences. In theory, it would be possible to include a video that was compressed with different codecs as well. While not generally done, a container could, in principle, contain both an H.264 compressed and another H.265 video track. When played on older mobile hardware, that device could then choose H.264 to conserve battery power, while modern phones would play the H.265 version.

This need for multiple audio, video or subtitle tracks constitutes an advantage of streamed data where bandwidth is concerned. Viewers can choose the language they prefer at any point during playback (as they are accustomed to doing in physical media distribution). While the streaming server has access to all the available tracks that are present in a download, only the ones that are needed for the current playback are transferred to the consumer. We discussed this behavior above, in the section on the principles of networked streaming. There, the quality was automatically changed mid-stream, but the same applies to language or subtitle track selection. The server hosts multiple video and audio tracks, and either assembles them on the fly or has a collection of preassembled files to fit the needs of the playing software.

This on-the-fly assembly indicates another feature of container tracks that we have not discussed so far: track synchronization and frame interlacing. The latter in particular is a valuable property when we are freely combining different versions of audio and video as described above.

Track Synchronization and Interlacing

At first glance, a track seems to be comparable to a file in a ZIP container—one for video, one for audio and another one for subtitles. However, a closer look reveals that video files are different. The various tracks have to play in sync, as viewers are very susceptible to the desynchronization of audio and video. Humans readily notice an offset of 50 ms (for a 60 Hz video that is equal to 3 frames) or less. With subtitles, slight differences are not as noticeable, but the effects/control track may be sensitive as well (depending on the effect).

In that regard, the tracks in a video container cannot be considered independent like a file in a ZIP archive. They have to share a common time code that is managed by the container format. While it sounds trivial to maintain a common time code, this is not an easy task to accomplish. Audio and video codecs are separate entities that are not matched. Hence, the time representations stored within the audio or video track are not compatible. Containers unify this time representation, as they "understand" the time present in all the supported codecs. This understanding is the main reason why containers are limited to particular known codecs. The relevant management information is stored in another essential section of the file: the header.

Headers usually encode global information like the author, length of the content, and the list of available tracks with their corresponding types and storage/compression format.

Each track itself starts with a subheader, which includes codec-specific information as well as an index that maintains the entry points[5] for each frame. There are different ways to layout data within a container, which directly influence whether or not a file is playable over a network or not. Since tracks are encoded with independent codecs, the most obvious way to store the data is in sequential order—each track containing the frames in sequential order. This layout is shown in Fig. 8.2a. Admittedly, there is an immediately apparent problem. How do we keep the separate files in sync? To understand why this is problematic, we need to look at data transfers from a disc to a player or over a network. From a high-level technical point of view, this is the same. We need to open a file (from a disc or a network resource) and start to load the data into memory to decode and play it. The main difference is speed. While reading local data is very efficient on a modern PC, a network is typically slow. To understand the difference, we use the size of Blu-ray Discs as an example. A movie in Blu-ray quality[6] consumes about 50 GB (including all its audio tracks).

When a downloaded 50 GB file is read from a disc for processing, we are usually limited by the bandwidth of the hard drive. We calculate the approximate time to read the entire file based on the fastest single SSD internal storage as of 2020—an NVMe SSD. When attached over PCIe 3.0[7] with four lanes, the maximum bit rate is approximately 31 GBit/s (Wikipedia contributors 2020). Since this bit rate includes the so-called protocol overhead,[8] the bit rate available for the actual data is less. According to a recent benchmark test by a PC-gaming (incidentally one of the most demanding tasks a PC can handle) website (PCGamer 2020), the fastest available SSD can read up to 27 GBit/s. Thus, the system could read the entire file into memory in roughly 15 s.

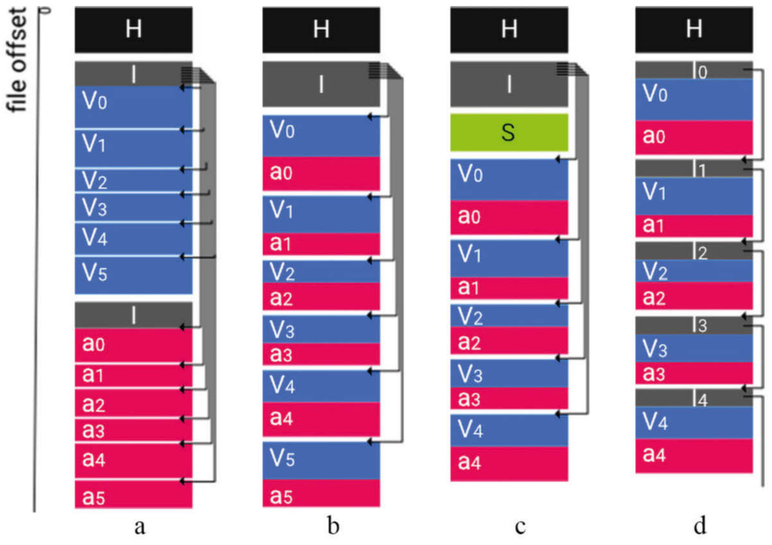


Figure 8.2: a) Tracks in sequential order. b) Interlaced tracks. c) Including subtitles. d) Interlaced tracks without global index

On the other hand, we have a file transfer over the network. A fast consumer-grade cable connection in 2020 can handle up to 1 GBit/s. This bandwidth increases the time for the entire file read to about six and a half minutes.

Let us revisit Fig. 8.2a with that knowledge and the assumption that the video data is approximately twice the size of the audio track. The player needs to download about two-thirds of the file before it has access to the Index Record (IR) for the audio track. This storage layout introduces playback latency (both on the hard drive as well as on the network). However, while the latency on the local drive is about 10 s (which is quite large), the time required on the network is still about 4 minutes—a period most people would consider too long.

Luckily, the situation is not as dire in real life. The header section stores the file offset where we can find the start and types of all the available tracks. When we play the file over a network, we can thus read that header section to find the start of the video and audio tracks and request that section of the file from the server, reducing latency to the bare minimum.

Unfortunately, it introduces another issue. We have to make two separate requests for non-sequential sections of the file—one for the video section of the file and one for audio. It is important to note that non-sequential file reads are slower than continuous reads. This speed reduction may be no problem for a local disc that plays one video at a time. However, a typical video server (or the CDN server that distributes the video) handles hundreds of clients simultaneously, where unordered disc access will pose a significant problem.

The problem stems from the fact that we need information at once from two different tracks, stored at two different locations within a file. Each contains the entire data, like a file in a folder. Fortunately, since the container manages the data, we can change that layout and combine the tracks so that everything we need in order to play a single frame of the movie (or a short sequence) is stored as self-contained and sequentially ordered (see Fig. 8.2b) in a frame package often called a segment. This technique is called data-interlacing, not to be confused with an interlaced scan (in contrast to a progressive scan) often used to enhance the perceived frame rate.

Interlaced data is not desirable for all types of tracks. For instance, a subtitle track does not contain information for every frame but the text for specific time codes. When storing the data interlaced, we would have to repeat the visible text for each frame in order to maintain the guarantee that each package contains all the essential information for a single frame. This constraint would introduce redundant information that increases the file size. In this case, it is preferable to store the entire subtitle stream before the video and audio packages (see Fig. 8.2c). This type of storage increases the start-up time slightly, but considering the small size of a subtitle track, this is barely noticeable.

This flexibility of arbitrarily rearranging the content of a movie is due to the use of containers. Furthermore, while containers were not developed to support streaming, the layouts they allow indirectly influence whether or not media is playable over a network; which in turn is a precondition to networked streaming and will help us to classify the intent of a file.

Live Video Encoding

There are situations when it is impossible to maintain a global file index (I in Fig. 8.2c is a global file index that stores the file offsets for each segment available in the file): for example, live video encoding, often found when live-streaming content or when a camera records data. Typical one-to-many live streams are found on Platforms like Twitch (a YouTube competitor with a strong emphasis on live streaming). It was initially focused on watching others playing video games and allows mostly text-based interactions with viewers in real-time. However, the scope was expanded quickly to various live broadcasts from people playing music to doing mundane activities like sleeping.

Another non-persistent, many-to-many live-streaming derivative is commonly used when using Skype, Teams or Zoom.

But why do we need a global index at all? The answer is found in the properties of unordered data. This lack of order manifests as the observation that segments can have different sizes (i.e. due to compression, see Fig. 8.2). Since the video of a live stream is never fully encoded, we do not know the size or count of the individual segments. Thus, it is impossible to calculate the start position of any segment, which precludes us from constructing the index before the entire video is encoded.

This is where things get complicated. The size of the index itself influences the location of the first segment, which would be the first entry in the index. That size depends on the number of segments we need to store, which is unknown when we start a camera recording or a live stream. Even when the video is already known, the number of segments is undetermined before the compression is finished, as, in simplified terms, an encoder creates a segment at fixed time intervals, or whenever the content of the scene changes rapidly (see section on inter-frame compression for more details).

A straightforward way of dealing with this problem is to use a progressive index, as shown in Fig. 8.2d. Instead of having a global index at the beginning of the stream, each segment contains an index that references the previous and following segments. That index has a fixed size (as only two segments are referenced) enabling the encoder to write this structure on the fly. When playing back that live stream, users have two choices: starting at the live edge (the last segment that was encoded just before they started watching) or, hypothetically, from the beginning of the recorded stream.

When joining the live stream at the current time, the server delivers the last completed segment. As it contains interlaced data, it is immediately playable.

Moreover, it contains the address of the next segment as well, which is then requested by the playback software. There is no need for a global index.

Playing the recorded stream from the beginning is also latency-free. The server immediately delivers the first segment of the stream (which is found immediately after the header). That segment references the second one, which references the third, and so on.

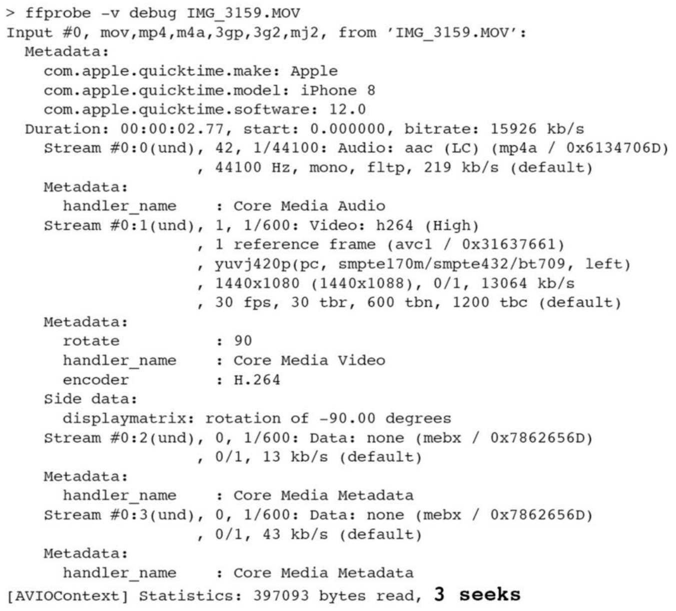
The lack of arbitrary positioning in live streaming, as we have described it, violates one of our key properties of streaming that we postulated earlier. To explain this, we need to subclassify the world of networked streaming into volatile streams (which are never recorded by the producer but sent and consumed) and recorded streams. A traditional live stream is volatile. Visitors can only join at the live edge and not start from the beginning or at any other position. For the moment, we will ignore this type of stream.

Recorded streams are another matter. Streamed content is delivered to viewers on the live edge and simultaneously stored on the streaming server for later consumption. However, the resulting recordings do not exhibit a global index, which may introduce lag on later networked playback. This is usually fixed by re-muxing the recording after it finished in order to create a global index for later playback.

Seeking

Latency is introduced when playing existing videos from an arbitrary position without a global index over a network connection or other slow media like a DVD. This lag was noticeable on older iTunes purchases, where the entire media file was downloaded to the customer before playback was allowed. It is still evident when uploading and sharing unprocessed videos from most smartphones on a personal website. Since those recordings do not contain a global index, playback from an arbitrary position is not immediately possible. Fig. 8.3 shows this behavior (last line on the output) for an unprocessed video recorded on an iPhone 8. The decoder (the playing software) needs to seek three times before playback can start from the beginning. And it gets worse when starting at a random position.

Let us assume we want to start the playback at segment 600 (random access[9] to the 600th element). The software needs to start reading the first element (just as if the client was starting from the beginning of the video) to find the position of the second, then the third and so on (see Fig. 8.2d). In contrast to playing from the start, when the entire segment is read from disc, this seeking operation only reads the beginning of the segment where the index is stored to determine the location of the next segment. Applied to a file played over a network, the system has to either download all the segments up to 600 before playback can start, or the server has to read arbitrary file positions, behavior we established above as bad on the server side.



```
> ffprobe -v debug IMG_3159.MOV
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'IMG_3159.MOV':
  Metadata:
    com.apple.quicktime.make: Apple
    com.apple.quicktime.model: iPhone 8
    com.apple.quicktime.software: 12.0
  Duration: 00:00:02.77, start: 0.000000, bitrate: 15926 kb/s
    Stream #0:0(und), 42, 1/44100: Audio: aac (LC) (mp4a / 0x6134706D)
                    , 44100 Hz, mono, fltp, 219 kb/s (default)
    Metadata:
      handler_name    : Core Media Audio
    Stream #0:1(und), 1, 1/600: Video: h264 (High)
                    , 1 reference frame (avc1 / 0x31637661)
                    , yuvj420p(pc, smpte170m/smpte432/bt709, left)
                    , 1440x1080 (1440x1088), 0/1, 13064 kb/s
                    , 30 fps, 30 tbr, 600 tbn, 1200 tbc (default)
    Metadata:
      rotate          : 90
      handler_name    : Core Media Video
      encoder         : H.264
    Side data:
      displaymatrix: rotation of -90.00 degrees
    Stream #0:2(und), 0, 1/600: Data: none (mebx / 0x7862656D)
                    , 0/1, 13 kb/s (default)
    Metadata:
      handler_name    : Core Media Metadata
    Stream #0:3(und), 0, 1/600: Data: none (mebx / 0x7862656D)
                    , 0/1, 43 kb/s (default)
    Metadata:
      handler_name    : Core Media Metadata
[AVIOContext] Statistics: 397093 bytes read, 3 seeks
```

Figure 8.3: Abbreviated output from *ffprobe* showing the number of seek operations for a video recorded on an iPhone 8.

This approach is also problematic when seeking within a video clip, as we need to download all frames until we find the seek position. This restriction would—mainly for streaming media—introduce a significant latency, as starting playback at a random position or seeking through a file would cause the system to transfer all the data between the current and the target positions.

Servers often optimize this and do not send the complete data. In those cases, the server has to perform unnecessary random file accesses when processing the index section of every segment up to the one that will be played. This problem, of course, has a known solution: the global file index, where we can quickly look up the position of the segment we need to play first without seeking through the entire file.

As the first example of an end user's streaming session implied, this behavior (starting playback at random locations and jumping within the content without delay) is an essential aspect of the overall streaming experience. However, this behavior hinges on the ability to maintain a global index. When not dealing with live streaming, this is the premier property a video container needs to offer in order to be classified as streaming media.

The live encoding described above still did not contain a global index, as the generated stream was just a linked collection of segments. Each segment knows the ones immediately before and after, but no other. However, recorded streams will also build a global index while recording, either in a separate file or in memory. In this case, the file system is the container. Individual segments are stored in separate, self-contained container files along with a corresponding global index file. That global index is queried whenever viewers want to play a non-sequential position from the stream. The result is the container file for the segment that needs to be played next. From there on, the segments play sequentially without needing further look-ups in the global index. This way, only seek operations cause a limited number of non-continuous file reads.

File System Containers

For the most part, we have assumed that a segment is a part of a large container for the entire movie. Nevertheless, as we have just described, a segment could be considered a short clip from the played video, which we could store in a single file instead.

In this scenario, those files are containers around at least one segment, with references to the previous and next file in the video. A global index is also required. Its purpose is the association of playback time with segment files. This file is often referred to as a playlist and can change dynamically, for example when live streaming.

This representation of a container may be unusual, but mirrors the same structure as we have discussed before, as the file system is just another means to arrange data.

The Influence of a Codec

So far, we have directed our focus onto the characteristics of containers to identify properties that are important for streams. However, the choice of codec settings for a video normally restricts, or at the very least influences, the way we can use it. While specific codec settings might work perfectly well for downloaded videos, the same settings might negatively impact the user experience when streamed over a network. These settings hence further inform our decision on whether a given file is intended for download or streaming.

Before diving into the specifics of codec settings, we first have to highlight a feature of most consumer-grade lossy video compression: inter-frame compression using I-, P- and B-frames.

Inter-Frame Compression

Fortunately, video data is more than a collection of random images over time. Most frames have a temporal consistency, i.e. the difference between two neighboring frames over time is likely to be small or smaller than the actual stored values. When compressing a video, we can take advantage of this property, as the number of bits we need to encode information depends on the number of different values we need to store. Seven bits allow us to store 128 unique values, while 8 bits double that amount to 256.

Consider two greyscale frames and their brightness values p0 and p1 of the same pixel [compare to Fig. 8.4a]. For this example, we need at least 10 bits for each in order to store the given brightness values p0=612 and p1=532 (10 bits can store up to 1024 distinct values). If we calculate the difference d=p1-p0 = -80, we only need 10 bits to store p0 and 8 bits (7 bits for the value 80 and one bit for the sign) to store the same information. If we also allow a small error (a slight brightness deviation that the viewer will most likely not notice), for example, a maximum error of 8 (out of 1024, which is approx. 0.8 % error), we can save an additional 3 bits when storing the value of d. Hence, we only require 15 bits for storage instead of the original 20.

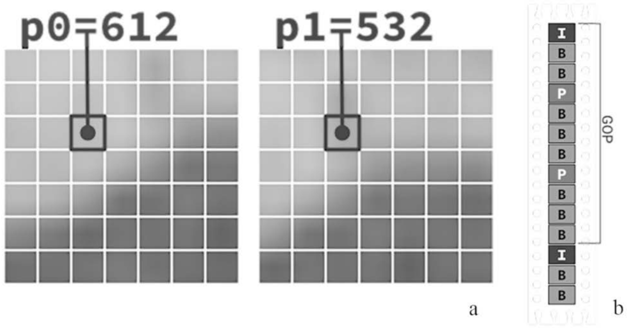


Figure 8.4: a) Two neighboring frames over time. b) Example of a GOP.

Let us apply this technique to an uncompressed 4K, UHD (3840 x 2160 pixels, 10-Bit color depth per channel, no chroma subsampling) video. Two uncompressed frames take up approximately 60 MB of storage. With this simple compression trick, we would reduce the size down to about 44 MB.

This type of compression is no longer lossless and is the basis for most modern video compression algorithms like MPEG2, H.264 or H.265. In 2019, this bit compression gained some attention in the final season of *Game of Thrones* (HBO, 2011–2019). Some dark and moody scenes were mastered for UHD, 10-Bit broadcasting but were then compressed by streaming Platforms (in Germany Sky and Amazon Prime Video) to an 8-Bit format. The high compression combined with color depth reduction resulted in very noticeable banding and block compression artefacts (see Fig. 8.5; we have increased the brightness, since the original image is extraordinarily dark and artefacts are not as visible against the white background of the page), unusual for an otherwise high-quality production. The scene lighting and composition, in this case, is very hard for compression algorithms, as the image content in Fig. 8.5a covers only the darkest range of brightness values. In contrast to the bright page, it almost looks like a completely black image. Difference-based compression techniques always introduce an error in the color and brightness channels of the image. Such errors are more visible in dark scenes with nuanced colors, as humans notice small brightness variance in the dark much better. However, while more pronounced when streaming (due to the limited available bandwidth), this particular problem is not intrinsic to streaming.

Nevertheless, this restriction is another fascinating point of compression: it may affect creative choices. The producers (and streaming providers) had to deal with some negative feedback due to those artefacts, which could have been circumvented by filming with more traditional lighting, where dark scenes are mostly blue, but still cover a broad brightness spectrum (more akin to Fig. 8.5b). Negative backlash due to technical restrictions can force producers to rethink an otherwise compelling choice.

There is also an economic impact for the stream provider. A popular show like *Game of Thrones* in its final season attracted millions of viewers watching the content simultaneously. This number of views is a strain on the available bandwidth (costing money), which necessitates more robust compression to reduce the load on the servers. At the same time, it will also decrease the perceived quality of the source material, which in turn may drive some customers away from the platform.

The GOP

Most video compression codecs derived from MPEG make use of delta encoding by specifying different frame types within a video. In simple terms, those are:

- **I-frames:** store all pixel values as they are. Thus, an I-frame can be decoded without knowledge of the prior or following frames.
- **P-frames:** are encoded relative to one preceding I-frame or another (earlier or later) P-frame. They make use of delta encoding.
- **B-frames:** are encoded using multiple preceding (or following) frames as a reference.

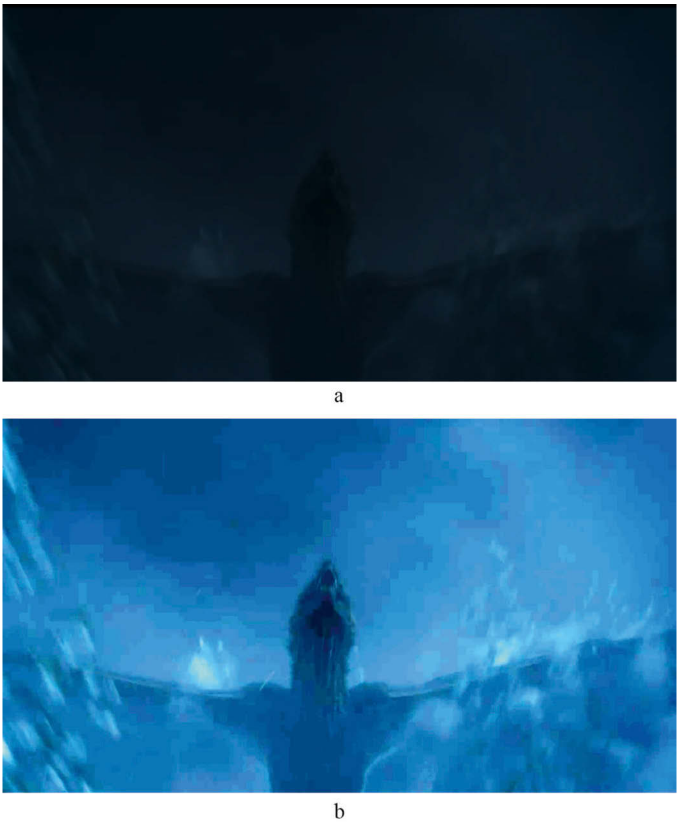


Figure 8.5: a) *Game of Thrones,* Season 8, Episode 3 at 21:09. b) The same frame with increased brightness.

All dependent frames are arranged into a GOP (Group of Pictures, see Fig. 8.4b), which constitutes the smallest, self-sustained unit of video. Please note that one or more GOPs (and not frames) are stored within the segments of a container along with the corresponding fragment of the audio track, as well as index information. Usually, frames can only be decompressed when the entire GOP is available (in a Network that would mean the entire GOP had to be transferred).

This GOP structure has a vital implication for video playback. It is no longer possible to start playback at any arbitrary position, as the decoder needs the first I-frame inside a GOP or (e.g. with H.265) the entire GOP. GOP size (which is mostly a measure of the number of I-frames) has contradicting implications. Increasing the GOP size (by reducing the number of I-frames or, in other words, increasing the time between two I-frames) will reduce the size of transferred data while the stream is playing. However, it will increase the data that needs to be preloaded before the stream can start, as the decoder will need to download the entire GOP that includes the desired playback position.

The decision on how much time, or rather how many P/B-frames follow a given I-frame is determined by the compression settings. When encoding a video with H.264, using the popular FFmpeg library, an I-frame is inserted every 250 frames by default (FFmpeg 2020). While this works well for videos intended for download and reduces the bandwidth consumed while streaming media, it has specific implications for the user experience in streaming. For example, when watching a movie over a streaming platform and manually jumping to a random point in time, the server has to send the entire GOP that includes the specified time. The playback can only start after its successful transmission.

While latency may be acceptable, transmission errors in the I-frame may result in decoding errors until the next full I-frame is received. It is important to note that especially streaming media is often sent using UDP, a network protocol that does not offer any guarantee on correctness, completeness or order. This lack of correctness makes streaming data prone to unrecovered transmission errors which (when randomly affecting an I-frame) will have a lasting visual impact until the next I-frame is received.

Assume a brightness that is encoded with one I-frame followed by five P-frames: IPPPPP, IPPPPP, . . . The source material contains the following brightness values for a pixel p over time: 612, 532, 613, 600, 640, 700, 1010 (compare to Fig. 8.4a). Those values are delta-encoded on the transmission side, resulting in the values 612 (first I-frame), -80, 1, -12, 28, 88, 1010 (next I-frame). This sequence is transferred to the viewer; however, due to transmission errors, the transmitted values are changed to 200, -80, 1, -12, 28, 88, 1010. The new sequence would result in decoded brightness of 200, 120, 201, 188, 228, 288, 1010. In this case, the error is visible for 6 frames before the stream can correct itself with the next I-frame. The recovery time from data errors is a critical consideration when compressing video, especially in an unreliable network environment. In the worst case, this amounts to a recovery time of nearly 10 s for the example GOP size of 250 frames. Since streaming is built for networks, this recovery time is another indication that helps us to differentiate a streamable file.

Reality Check

Please note that in reality neither MPEG2, H.264 or H.265 store the actual color difference. The process is more involved than that, but this simple idea is sufficient for our purpose (Richardson 2004; Schwarz et al. 2014). The actual compression algorithm does not work on a per-pixel basis but (similar to JPEG compression) on a group of 16 x 16 neighboring pixels (hierarchically subdivided into smaller blocks down to the size of 4 x 4 pixels). For example, the H.264 algorithm tries to identify the movement of blocks of pixels and stores the movement vectors as well as color change coefficients per block. This block-wise compression significantly reduces the size of the resulting images, way more than our simple example. The number of allocated bits the encoder can use to store the block differences similarly determines the quality of the compression.

Properties of a Stream?

We set out to decide whether a given video source is a download or a stream by just looking at the properties of the transferred media. In the section on the principles of networked streaming, we defined certain key behaviors of a networked streaming experience, namely adjustable quality and content as well as fast playback from arbitrary positions.

A simple solution was found for the random starting positions. The key was the availability of a global index. This index should be stored at a known, fixed position within the file (e.g., after the fixed-size header). But many downloadable files also contain this global index.

In the passages on GOP, we also mentioned that error correction is an essential aspect for a stream. In our example, we stated that an easy solution would be increasing the I-frame count. However, there are better and more complex techniques we have not talked about so far. Most streamed media include checksums and recovery data along with the transmitted segments, in order to enable recovery from degrading transmissions. If either property (global index or error correction) are not available, we can determine that the media is not suitable for streaming. Since those properties are also present in most downloadable content, we need another criterion to distinguish streaming media.

Another impact of the GOP size was latency, which should be as small as possible for streaming. Hence, codec properties like high I-frame counts are another indication of a stream that we will apply.

An additional critical distinction is related to the ability to change the quality and content of a stream dynamically. This is made possible by the container's ability to manage several interlaced tracks in self-contained segments. In fact, considering a segment as a self-contained entity (as described in the paragraphs on file-system containers) with easily customizable content exactly tailored to the users' needs is the chief criterion we will adopt to identify streamed media.

Divide and Conquer

The critical implication of self-sustained segments is not the fact that they are stored in single files, but that the entire content is separated into smaller segments. Each of those segments should only contain the minimum amount of data that is needed for playback, i.e. only one video and audio track, not all the available languages. However, the selected language or video quality or any other track property can change with each delivered segment (when necessary). During that change, seamless playback (without long delays and repeated or missing content) is maintained.

We can determine the self-sustainability of segments when looking at the file during playback on the networking level. While some downloadable files can be played back during the download, they are delivered with all the available tracks and qualities included. Even when the downloading client allows viewers to jump forward, the data packages themselves are not minimal and have no relation to the content, e.g. if they contain multiple audio tracks. This behavior is called progressive download, not streaming. It is important to note that from the user's perspective a progressive download behaves just like a stream, although the delivered packages are not, and cannot be customized to the specific user settings.

Conversely, a streaming video will always deliver the bare minimum of data in small successive and mostly self-contained segments. Each segment provides all the information needed to play a short fraction of the movie, even if previous segments have been lost. Hence, splitting the video into minimal, self-sustained parts is our primary indication for streamable media.

Applications

Now the requirements for streaming media have been set, we can investigate some sample streaming applications. The introductory example from the section on the principles of networked streaming already constituted a stream, as we have derived our entire premise from this example. However, there are other examples we will try to classify by applying our rules.

MPEG-TS

MPEG Transport Stream (W3C Working Group 2016) is a standardized container format that is best known for digital broadcasting like DVB or IPTV. It constitutes a format that can transfer multiple simultaneous streams by packaging the content in small .ts files. Let us check our three properties for streaming against the format's specification:

- Global Index: MPEG-TS is often used for television broadcasting, where changing the playback position is not relevant. Streams will always start at the live edge. The purpose of the global index was maintaining a list of references to all playable positions, in this case, only a single one, the live edge. This information is kept by the Service Information (SI) that is sent along with the Transport Stream.
- Error Correction: As broadcasts are sent over inherently unreliable connections like a radio transmission, error correction is an essential aspect of the format.
- Self-Contained, Minimal Segments: With MPEG-TS a segment has to be assembled from multiple network packets carrying the different tracks (audio, video, subtitles and metadata). Packets include a particular record, the so-called reference clock (PCR), which keeps the packets in sync. From our high-level description, all packets that contain data for the watched stream and are sent after one PCR constitute one segment of data and contain the bare minimum of information. However, this assumption of minimum data is problematic when talking about broadcasting. Since the data connection (for example over the air) is not a 1:1 relation (where the broadcasting station sends custom data to each individual viewer), all the available tracks are sent into the air. The video player will then receive all packets (including ones with unnecessary data), and process those that are needed for playback. When sent over a network connection, however, this problem with MPEG-TS does not arise, as the server can assemble a transport stream that fits the needs of an individual viewer and only includes packages of interest.

MPEG-TS fits our definition for a stream perfectly. The critical observation is that MPEG-TS can be used in such a way that only minimal data is sent. Each segment can contain different combinations of tracks without causing the playback to be interrupted. That is why we classify the format as a stream, as it could be used for streaming.

Here is an appealing implication. The data format of a Blu-ray Disc—MPEG Transport Stream for BD-ROM or .m2ts (Blu-ray Disc Association 2005)—is derived from MPEG-TS with only minor modifications. This relation makes the data stored on a Blu-ray Disc streaming media. Nevertheless, for the intention of this paper, we are concerned explicitly with media sent over a network.

Cloud Gaming

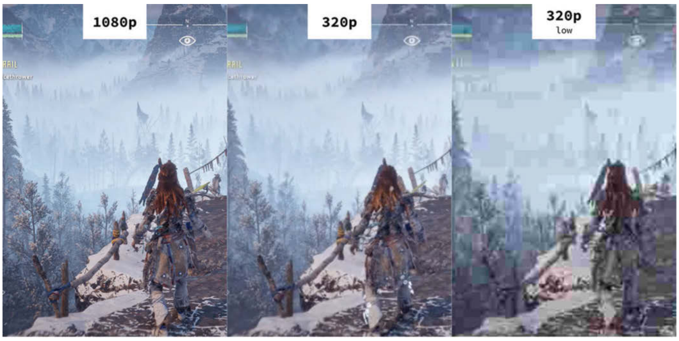
Recently, multiple platforms started to provide live cloud gaming. The idea is that users do not install and run video games on their local machine but on a remote server. Cloud gaming allows users to experience high-definition video games on machines with low hardware specs. The local machine will record the user input and send it to a server. That server, a powerful gaming machine, renders high-definition video and audio for the given input, encodes it as a live stream and sends it back to the client. Gaming, nonetheless, poses distinct requirements for the streaming service. In particular, the most crucial factor to immersive gaming is low latency.

Providers of game streaming, therefore, advertise round trip times[10] of less than 70 ms (Fox 2020). For comparison, the latency in gaming on a local machine is in the 40 ms range, depending on the game settings.

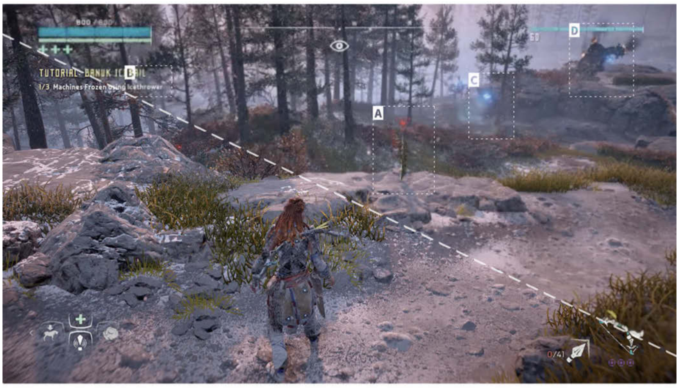
The 70 ms mentioned above comprise roughly the following steps: the user input is captured and sent to the server. The server's graphics card waits until previous frames have finished rendering, then synthesizes the new image that incorporates the user input. This last step is identical to gaming on a local machine. However, instead of presenting the result on a display, the frame is compressed and sent to the client. Cloud gaming is a highly individual and interactive streaming experience, as most of the generated images and experiences are unique to an individual player.

The 30 ms difference between local and remote gaming is primarily due to the data transmission; the compression and decompression steps also factor into the additional time cost. It follows that extreme care has to be taken in order to make the compression and decompression as fast as possible. Fig. 8.6a shows some real-world compression examples captured with *PS4 Remote Play* on a local network. As is typical for adaptive quality streaming, the low bandwidth version (320p) both reduces the resolution (320p instead of 1080p) and employs more aggressive compression. The high resolution 1080p is (except for latency) visually comparable to the HDMI output of the console. At first glance, the change down to 320p does not have a significant impact. The image is not as sharp, but for a limited bandwidth network, this signifies an acceptable compromise. Even the "320p-low" version[11] would still be watchable as pure video content. Please note that this aggressively recompressed version shares the same artefacts we saw in Fig. 8.5. However, for games, the quality of the regular 320p compression is often not sufficient. Fig. 8.6b illustrates this problem. The change in compression is not especially visible in this example (the image is separated diagonally). Nevertheless, essential on-screen elements like markers (A), or text (B) are harder to read or not noticed at all. Enemies are less visible as well (C) and often merge with the surroundings (D) due to the image space frequency reduction of the compression at mid-range. When faced with limited bandwidth, the alternative would be a sharp image, at a lower image rate, brief stutters (when new images do not arrive in time) or even a complete picture loss. For gaming, missing images or lost connections are a worse outcome than reduced fidelity.

Similar to regular streaming, the chosen quality depends on the available bandwidth. Yet it is not purely about bandwidth; the latency cost of the compression



a



b

Figure 8.6: a) PS4 Remote Play screen capture of *Horizon Zero Dawn* (Sony Interactive Entertainment, 2017) with different compression settings, b) a detailed look at the impact of compression when playing.

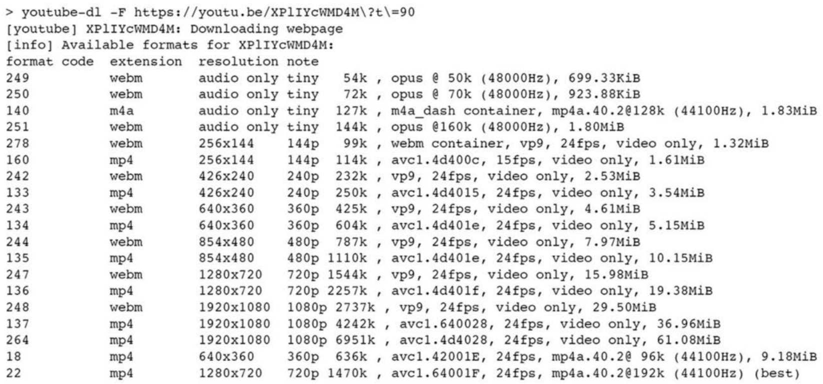
also factors in this calculation. Increasing the visual quality can reduce latencies, as the encoding/decoding steps perform faster. At the same time, this is contradicted by the increased transmission time (less compression results in larger files). In that regard, game streaming needs to strike a careful balance when choosing the right quality settings; just like its video streaming counterpart, however, it relies on adaptive streaming quality.

As we have mentioned, we classify cloud gaming as streaming. Similar to live streaming, it is impossible to change the playback time by jumping ahead or back. This restriction is due to the application, not to the delivered files. As we described above for MPEG-TS, this implies that the global index will only reference the segment that plays the live edge. The segments themselves are self-sustained packets containing the game's audio and video for a minimal GOP size. Whenever possible, a game stream will use a GOP that contains a single frame (which has to be an I-frame) to reduce latency to the bare minimum. This high I-frame frequency will also result in faster compression and better error handling, as we have described above.

We briefly touched on the topic of volatile streams in the section on live video encoding. The generated segments are entirely transient in this example. The gaming machine (in our example a PS4) will render the sound and images for the given point in time, encode them and package them into a segment—all within the memory of the machine. Those segments are sent to the player over the network, decoded and presented on a display. The data is never stored in the process. In this case, the container format is specified by the transmission protocol, which will have to handle the arrangement of data. Lost segments will not corrupt the entire stream, but are simply dropped during playback.

YouTube

YouTube is famously known for streaming, as it was one of the first real video streaming platforms. However, in this section, we pick an aspect offered by third-party websites, which allow users to download the videos for local playback. Those sites may use tools like FFmpeg[12] that can take a stream as input and store it in a single file. In general, tools like FFmpeg act like video players. Instead of presenting the content on a display, they convert it for storage.



```
> youtube-dl -F https://youtu.be/XPlIYcWMD4M\?t\=90
[youtube] XPlIYcWMD4M: Downloading webpage
[info] Available formats for XPlIYcWMD4M:
format code  extension  resolution note
249          webm       audio only tiny   54k , opus @ 50k (48000Hz), 699.33KiB
250          webm       audio only tiny   72k , opus @ 70k (48000Hz), 923.88KiB
140          m4a        audio only tiny  127k , m4a_dash container, mp4a.40.2@128k (44100Hz), 1.83MiB
251          webm       audio only tiny  144k , opus @160k (48000Hz), 1.80MiB
278          webm       256x144    144p   99k , webm container, vp9, 24fps, video only, 1.32MiB
160          mp4        256x144    144p  114k , avc1.4d400c, 15fps, video only, 1.61MiB
242          webm       426x240    240p  232k , vp9, 24fps, video only, 2.53MiB
133          mp4        426x240    240p  250k , avc1.4d4015, 24fps, video only, 3.54MiB
243          webm       640x360    360p  425k , vp9, 24fps, video only, 4.61MiB
134          mp4        640x360    360p  604k , avc1.4d401e, 24fps, video only, 5.15MiB
244          webm       854x480    480p  787k , vp9, 24fps, video only, 7.97MiB
135          mp4        854x480    480p 1110k , avc1.4d401e, 24fps, video only, 10.15MiB
247          webm       1280x720   720p 1544k , vp9, 24fps, video only, 15.98MiB
136          mp4        1280x720   720p 2257k , avc1.4d401f, 24fps, video only, 19.38MiB
248          webm       1920x1080  1080p 2737k , vp9, 24fps, video only, 29.50MiB
137          mp4        1920x1080  1080p 4242k , avc1.640028, 24fps, video only, 36.96MiB
264          mp4        1920x1080  1080p 6951k , avc1.4d4028, 24fps, video only, 61.08MiB
18           mp4        640x360    360p  636k , avc1.42001E, 24fps, mp4a.40.2@ 96k (44100Hz), 9.18MiB
22           mp4        1280x720   720p 1470k , avc1.64001F, 24fps, mp4a.40.2@192k (44100Hz) (best)
```

Figure 8.7: Available streams for the previously mentioned *The Hunger Games*-video.

The first definition of a stream we encountered was characterized by the volatile nature of the data. In that regard, the tool used was the deciding factor whether or not the data was a stream. Consider the YouTube example: when videos are played in a browser (like Safari, Google Chrome, Edge or Firefox), that definition will classify the experience as streaming, as the segments received from YouTube are played and immediately discarded. If the same input is consumed through FFmpeg, and watched at a later point, it is classified as a download, because the same transferred data was then stored. But why would the same content be classified differently? The data sent from YouTube is always a streaming format that meets all our criteria. YouTube hosts the same content as multiple single files that are created using different codecs for compression (see Fig. 8.7). Those files are created in such a way that the server can switch seamlessly between segments from different files during playback in order to change the quality. Like the video player inside the browser, FFmpeg can start capturing the stream at any point in time, without delay, and change the quality (or content if available) at any point. The transferred segments are self-contained and minimal, and the built-in error correction is still applied (as it is when playing the video on the YouTube website). We can even configure FFmpeg to generate a copy of the streamed segments by storing the data as, for instance, an MPEG Transport Stream. However, the resulting files will lose the ability to change quality or content seamlessly. The decisions made when the file is played (even when played using FFmpeg) are an integral part of the created file.

In that sense, FFmpeg transforms the data into a new format. While the source data remains a stream, the data has undergone another transformation that changed its purpose from being a stream to make it into a download. For example, when storing the result as a .mp4 container, the file will (at least when using the default FFmpeg settings) lose the global index. Without it, it is impossible to play the file over a network without noticeable delay.

This transformation happens all the time to videos moving through the cloud. They start as source material from a camera, are edited to a completed movie and then distributed as downloads to streaming companies, broadcasters or cinemas. A streaming service will then transform the video to create a format that is suitable for streaming. In this final illustration, the end user receives the streaming data and converts it back into a downloadable format. Nevertheless, the process of receiving the data was still streaming. This result contradicts the initial assumption, as the resulting file is no longer volatile, but a permanent file on the receiver's hard drive. It is only if that file is then reoffered over a network, such as by a conversion website, that the following transfer will constitute a download.

Conclusion

There are a lot of intricacies and small details of various systems at play that contribute to the generic streaming experience. While we have been able to narrow down the inherent characteristics of containers and codecs that qualify video content for potential streaming, we have also left a lot unwritten: e.g., the role of server software or network protocols like HSL, RTMP, Dash or WebRTC, which are an essential aspect when implementing (live) streaming. However, for the purpose of our discussion—classifying media as a stream or download based solely on the properties of the files themselves—this is unnecessary detail.

The most relevant message is the importance of container formats when processing or consuming video. Containers act as a proxy that can combine otherwise unrelated codecs (like MPEG Audio Layer III and H.265) into a working video experience. They bridge the gap between the server and the player software and transport information that allows us to view multiple media types in sync. Without containers it would be impossible to move video and audio through the internet in sync.

This flexibility constitutes a significant trade-off when distributing media. Most modern browsers enable regular and progressive downloads, making it easy to share short clips with the world by simply uploading videos to any web host.

Streaming, nonetheless, requires specialized software on the server and on the part of the client that handles the end customer delivery as well as the media conversion into an appropriate, streamable format. On large streaming providers like YouTube or Twitch, this added complexity on the server is hidden by a specialized and uncomplicated User Interface that accepts any uploaded movie from the client and transcodes or (where possible) re-muxes it into a streamable format with different, default qualities. That server also takes care of recombining tracks into minimal segments. The player software (either the web frontend or dedicated mobile apps) is tuned to that same server to provide the ability to switch resolutions without any noticeable gaps in playback. Using that UI makes streaming as simple as providing downloads, while it generates lock-in to large streaming or video portals.

Although it is technically possible to recreate that service for a personal VLog, most YouTubers or Twitchers do not possess the required skills to implement this; a simple fact that puts leverage into the hands of large streaming platforms, creating an imbalance in power, which is often addressed on the streaming platforms themselves whenever seemingly arbitrary rules are imposed on the creatives.

The concentration of creative media on a few large video platforms also impacts legislative regulations. While it is hard to impose rules or block every website on the internet, it is easy to block or establish laws that force the largest streaming platforms to uphold precepts. This simplification is arguably good when upholding common law, but it also simplifies censorship, as has been evident, amongst other cases, in Turkey (Letsch and Rush 2014) or Germany (van Buskirk 2009).

Indeed, the complexity of streaming, especially at the beginning of user-created videos, is not the main reason for lock-in. Factors like reach and monetization are more critical to creatives. Nevertheless, it is still surprising that a proprietary platform like YouTube has emerged amidst the open web. If more accessible, open technologies like HTTP Streaming or HSL,[13] had been available earlier, this might have played out differently.

There is another advantage of large video platforms that may have a direct impact on the transportability of media. As we discussed above, the codec of a video can impact energy consumption and user experience. A collection of decentralized video downloads (for example privately hosted on a web server or previously downloaded from storefronts or other distributors) would be very hard to convert into more energy-efficient or better compressed, and thus smaller, formats. Large video providers, on the other hand, could have a positive impact by converting videos with large audiences to the best available codec before streaming the content. This centralized approach would limit both the required bandwidth (an economic advantage for distributors as well as consumers) and the energy footprint of their viewers.

Bibliography

BBC. 2020. "Netflix to Cut Streaming Quality in Europe for 30 Days." March 19. https://www.bbc.com/news/technology-51968302 (accessed June 12, 2020).

Blu-ray Disc Association. 2005. "White Paper: Blu-ray Disc Format, 2.B Audio Visual Application Format Specifications for BD-ROM." http://www.blu-raydisc.com/Assets/Downloadablefile/2b_bdrom_audiovisualapplication_0305-12955-15269.pdf (accessed June 16, 2020).

Deutsch, Peter. 1996. "Deflate: Compressed Data Format Specification Version 1.3." https://dl.acm.org/doi/book/10.17487/RFC1951 (accessed July 14, 2020).

Fenwick, Sam, and Hardik Khatri. 2020. "The State of Mobile Network Experience 2020: One Year into the 5G Era." *Opensignal*, May. https://www.opensignal.com/sites/opensignal-com/files/data/reports/pdf-only/data-2020-05/state_of_mobile_experience_may_2020_opensignal_3_0.pdf (accessed June 19, 2020).

FFmpeg. 2020. "Streaming Guide." https://trac.ffmpeg.org/wiki/StreamingGuide (accessed June 12, 2020).

Fox, Jacob. 2020. "Nvidia GeForce Now's Competitive Mode Cuts Input Latency 30 % Lower than Stadia." *PCGamesN*, 26 February. https://www.pcgamesn.com/nvidia/geforce-now-competitive-mode-latency (accessed June 12, 2020).

Hugenholtz, Bernt, ed. 2018. *Copyright Reconstructed: Rethinking Copyright's Economic Rights in a Time of Highly Dynamic Technological and Economic Change*. Alphen aan den Rijn: Wolters Kluwer.

IMAX. 2013. "The Hunger Games: Catching Fire IMAX® Behind the Frame." *YouTube*, 02 October. https://www.youtube.com/watch?v=XPlIYcWMD4M&feature=youtu.be&t=90 (accessed June 12, 2020).

Letsch, Constanze, and Dominic Rush. 2014. "Turkey Blocks YouTube Amid 'National Security' Concerns." *The Guardian*, 28 March. https://www.theguardian.com/world/2014/mar/27/google-youtube-ban-turkey-erdogan (accessed June 23, 2020).

Matroska. 2020. "Matroska Media Container." https://www.matroska.org/ (accessed June 12, 2020).

PCGamer. 2020. "Best NVMe SSD." https://www.pcgamer.com/best-nvme-ssd/ (accessed June 9, 2020).

Pkware. 2012. *Zip File Format, Version* 6.2.0. https://www.loc.gov/preservation/digital/formats/fdd/fdd000355.shtml (accessed June 12, 2020).

Richardson, Ian E. 2004. *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*. West Sussex: Wiley.

Schwarz, Heiko, Thomas Schierl, and Detlev Marpe. 2014. "Block Structures and Parallelism Feautures in HEVC." In *High Efficiency Video Coding (HEVC): Algorithms and Architectures*, edited by Vivienne Sze, Madhukar Budagavi, and Gary J. Sullivan, 49–90. New York: Springer.

Solmecke, Christian. 2013. "Redtube: Wave of Streaming Warning Letters Hits Germany." https://www.wbs-law.de/allgemein/redtube-wave-streaming-warning-letters-hits-germany-16589/ (accessed June 19, 2020).

van Buskirk, Eliot. 2009. "YouTube Blocks Music Videos in Germany." *Wired*, 1 April. https://www.wired.com/2009/04/youtube-blocks/ (accessed June 23, 2020).

Wikipedia contributors. 2020. "PCI Express." *Wikipedia*. https://de.wikipedia.org/wiki/PCI_Express (accessed June 9, 2020).

W3C Working Group. 2016. "Mpeg-2 TS Byte Stream Format." https://www.w3.org/TR/mse-byte-stream-format-mp2t/ (accessed June 16, 2020).

Notes

1 A cache is an automatic system included in most browsers to speed up page loading while surfing. Previously downloaded data remains on the local hard drive. When the same content is viewed again, the data is not loaded from the server but from the local cache.

2 The last iPhone that did not contain hardware to decode H.265.

3 Roughly the speed the device has at its disposal to receive video.

4 While JPEG is commonly referred to as the file format, it is a compression algorithm. The container format found in .jpg-files is called JIFF or Jpeg Image File Format.

5 The location of the track in the file, also known as the file offset.

6 Usually, H.264 is encoded FullHD, 8-Bit color-depth with 4:2:2 chroma subsampling at 30 Hz.

7 PCIe 3.0 is a hardware protocol that manages data transfer from PC peripherals like storage or graphics card to the CPU for processing. Currently, we are transitioning to PCIe 4.0.

8 Data that is sent to control the storage device or as checksums to validate the data integrity.

9 Random access is a term from computer science. It describes the fact that ordered data is not accessed in a predefined pattern (for example, sequentially from start to finish) but in an unpredictable order.

10 Similar to latency, the round trip time indicates the amount of time between the user's input and the playback of the frame showing the result.

11 This is not a compression allowed by PS4 Remote Play. The lowest quality the service provides is the result shown as 320p. For this "320p-low" version, we recompressed the original image using FFmpeg to produce the last example.

12 In combination with the *YouTube-dl* script.

13 At the time, streaming with flash, another proprietary format, was commonly used.