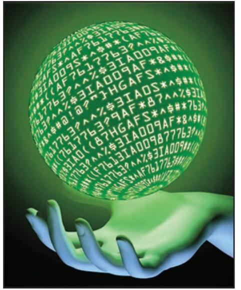

Watching Video over the Web

Part 1: Streaming Protocols

Ali C. Begen, Tankut Akgul, and Mark Baugher • *Cisco*

The average US consumer watches TV for almost five hours a day. While the majority of viewed content is still broadcast TV programming, the share of the time-shifted content is on the rise. One-third of US viewers currently use a digital video recorder like device, but trends indicate that more consumers are migrating to the Web to watch their favorite shows and movies. Increasingly, the Web is coming to digital TV, which incorporates movie downloads and streaming via Web protocols. In this first part of a two-part article, the authors describe both conventional and emerging streaming solutions using Web and non-Web protocols.

Summer 2010, time for the 19th FIFA World Cup. It was an exhilarating month, with 64 matches played by 32 national teams. Unsurprisingly, the World Cup is one of the most watched events worldwide, and this year marked the first time that viewers had the chance to see games broadcast in 3D. But more importantly, many more viewers than ever watched the games over the Web. With recent developments in video streaming technologies and the increase in broadband Internet access penetration, fans enjoyed watching games in high-definition or near-HD quality on their computers, smartphones, and other connected devices, including their TVs. In the US, 45 percent of the daily World Cup TV audience watched the matches in a non-home environment or on a non-TV platform (www.espnmediazone3.com/us/2010/07/espn-xp-world-cup-dispatch-4-through-742010). ESPN3.com reached more than 7 million unique viewers and delivered 15 million hours of content.

Just a few months prior, we witnessed the same phenomenon with the Vancouver 2010 Winter Games. In Canada, where the population is approximately 34 million, almost 4 million unique viewers watched the games on the Internet. CTV, the national TV network that broadcast the games, made 300 events available on the Web. Canadian Internet users consumed a total of 6.3 million hours of live and 0.9 million hours of on-demand content, resulting in a total 6.2 Pbytes delivered in about two weeks (www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000007347). Two years ago, the Beijing Summer Games similarly attracted online viewers, both for live and on-demand viewing, with NBC delivering more than 1,100 years of video content to almost 52 million unique online viewers in the US during the games.

Sporting events aren't the sole type of content attractive to online viewers, though. In the past few years, many providers have started making their regular and premium content such as news, series, shows, and movies available on their Web sites. Despite geographical restrictions on many of these Web sites, consumers saw a proliferation in the amount of content they had access to. Some content providers and TV channels don't impose restrictions on viewer location, and thus transformed from being a regional TV source to a global one, extending their reach for ad revenues in an unforeseen manner.

In this first installment of a two-part series, we describe the impetus behind this shift, focusing

Published by the IEEE Computer Society
1089-7801/11/$26.00 © 2011 IEEE

on several streaming solutions that exist today.

Thirsty for Streams

Consumers' desire to access practically limitless amounts of content any time they want and the drop in delivery costs hastened the deployment of streaming services. New Web sites that handle content aggregation such as Hulu emerged. In May 2010, Hulu had more than 40 million unique viewers in the US, streaming more than 1 billion videos per month (www.comscore.com/Press_Events/Press_Releases/2010/6/comScore_Releases_May_2010_U.S._Online_Video_Rankings). Remarkably, these numbers are steadily increasing. Netflix, the largest subscription service for DVD rental and streaming video, currently has over 20 million subscribers, many of whom use its streaming services on a variety of devices (www.netflix.com/MediaCenter?id=5379).

We can divide Internet video, also known as over-the-top (OTT) services, into distinct categories of user-generated content from mostly amateurs (such as the content served by YouTube), professionally generated content from studios and networks to promote their commercial offerings and programming (such as what you find on ABC.com or Hulu), and direct movie sales to consumers over the Internet (also referred to as electronic sell-through, or EST). In the last category, Netflix, Apple TV, and new undertakings such as UltraViolet are greatly increasing the amount of video offerings on the Internet.

Cisco's Visual Networking Index (VNI) suggests that traffic volumes in the order of tens and hundreds of exabytes (1 billion Gbytes) and zettabytes (1,000 Ebytes) aren't that remote. Over the next few years, 90 percent of the bits carried on the Internet will be video related and consumed by more than 1 billion users. Although some portion of these video bits will be for managed services such as cable TV and IPTV, we can't ignore the amount of bits for unmanaged (OTT) services.

Cable and IPTV services run over managed networks for distribution because these services use multicast transport and require certain quality-of-service (QoS) features.[1] In contrast, conventional streaming technologies such as Microsoft Windows Media, Apple QuickTime, and Adobe Flash, as well as the emerging adaptive streaming technologies such as Microsoft's Smooth Streaming, Apple's HTTP Live Streaming, and Adobe's HTTP Dynamic Streaming, run over mostly unmanaged networks. These streaming technologies send the content to the viewer over a unicast connection (from a server or content delivery network [CDN]) through either a proprietary streaming protocol running on top of an existing transport protocol, mostly TCP and occasionally UDP, or the standard HTTP protocol over TCP.

Historically, progressive download, which uses HTTP over TCP, has been quite popular for online content viewing due to its simplicity.

Historically, progressive download, which uses HTTP over TCP, has been quite popular for online content viewing due to its simplicity. In progressive download, the playout can start as soon as enough necessary data is retrieved and buffered. Today, YouTube delivers more than 2 billion videos daily with this approach. However, progressive download doesn't offer the flexibility and rich features of streaming. Before the download starts, the viewer must choose the most appropriate version if there are multiple offerings with different resolutions for the same content. If there isn't enough bandwidth for the selected version, the viewer might experience frequent freezes and rebuffering. Trick modes, such as fast-forward seek/play or rewind, are often unavailable or limited. These limitations are likely to inhibit the growth of large volume (including HD) movie distribution on the Internet. A new approach, which we refer to as *adaptive streaming*, is emerging to address these shortcomings while preserving the simplicity of progressive download.

Adaptive streaming is a hybrid of progressive download and streaming. On one hand, it's pull-based, as is progressive download: the adaptive streaming client sends HTTP request messages to retrieve particular segments of the content from an HTTP server and then renders the media while the content is being transferred. On the other hand, these segments are short, enabling the client to download only what's necessary and use trick modes much more efficiently, giving the impression that the client is streaming. More importantly, short-duration segments (for example, MPEG4 file fragments) are available at multiple bitrates, corresponding to different resolutions and quality levels, so the client can switch between different bitrates at each request. The client player strives to always retrieve the next best segment after examining a variety of parameters related to available network resources, such as available bandwidth and the state

of the TCP connections; device capabilities, such as display resolution and available CPU; and current streaming conditions, such as playback buffer size. The goal is to provide the best quality of experience by displaying the highest achievable quality, starting up faster, enabling quicker seeking, and reducing skips, freezes, and stutters.

Because adaptive streaming uses HTTP, it benefits from the ubiquitous connectivity that HTTP has to offer. Today, practically any connected device supports HTTP in some form. It's a pull-based protocol that easily traverses middleboxes, such as firewalls and NAT devices. It keeps minimal state information on the server side, which makes HTTP servers potentially more scalable than conventional push-based streaming servers. To the existing HTTP caching infrastructure, adaptive streaming is no different than any other HTTP application. Individual segments of any content are separately cacheable as regular Web objects using HTTP or any RESTful (conforming to the Representational State Transfer constraints) Web protocol. This allows distributed CDNs to greatly enhance the scalability of content distribution.

Media Streaming

Transmission of content between different nodes on a network can be performed in a variety of ways. The type of content being transferred and the underlying network conditions usually determine the methods used for communication. For simple file transfer over a lossy network, the emphasis is on reliable delivery: added redundancy protects packets against losses, or retransmission can recover lost packets. When it comes to audio/video media delivery with real-time viewing requirements, the emphasis is on low latency and jitter, and efficient transmission; occasional losses might be tolerated in this case. The structure of the packets and algorithms used to transmit real-time media on a given network collectively define the media streaming protocol. Although various media streaming protocols available today differ in implementation details, we can classify them into two main categories: push- and pull-based protocols.

Push-Based Media Streaming Protocols

In push-based streaming protocols, once a server and a client establish a connection, the server streams packets to the client until the client stops or interrupts the session. Consequently, in push-based streaming, the server maintains a session state with the client and listens for commands from the client regarding session-state changes. Real-time Streaming Protocol (RTSP), specified in RFC 2326, is one of the most common session control protocols used in push-based streaming.

Push-based streaming protocols generally utilize Real-time Transport Protocol (RTP), specified in RFC 3550, or equivalent packet formats for data transmission. RTP usually runs on User Datagram Protocol (UDP), a protocol without any inherent rate-control mechanisms. This lets the server push packets to the client at a bitrate that depends on an application-level client/server implementation rather than the underlying transport protocol, and makes RTP a nice fit for low-latency and best-effort media transmission.

In conventional push-based streaming, the server transmits content at the media encoding bitrate to match the client's media consumption rate. In normal circumstances, this ensures that client buffer levels remain stable over time. It also optimizes the use of network resources because the client usually can't consume at a rate above the encoding bitrate; consequently, transmitting above that rate would unnecessarily load the network. Moreover, transmission above the encoding bitrate might not be even possible for live streams in which the stream is encoded on the fly. However, if packet loss or transmission delays occur over the network, the client's packet retrieval rate can drop below its consumption rate, which might drain its buffer and eventually result in a buffer underflow that interrupts the playback. This is where bitrate adaptation comes into play.

To prevent a buffer underflow, the server can dynamically switch to a lower-bitrate stream. This, in turn, reduces the media consumption rate at the client side and counteracts the effect of network bandwidth capacity loss. Because a sudden drop in the encoding bitrate can result in a noticeable visual quality degradation, this reduction should occur in multiple intermediate steps until the client's consumption rate matches or drops below its available receive bandwidth. When network conditions improve, the server does the opposite and switches to a higher-bitrate stream — again, in multiple intermediate steps to avoid a sudden network overload. Provided that the stream isn't live and the network capacity allows for a higher-bitrate transmission, the server can also choose to send packets at a higher rate than the media encoding rate to fill up the client's buffer. By dynamically monitoring available bandwidth and buffer levels and by adjusting the transmission rate via stream switching, push-based adaptive streaming can achieve smooth playback at the best possible quality level without pauses or stuttering.

Bandwidth monitoring is usually performed on the client, which also computes network metrics such as round-trip time (RTT), packet loss, and network jitter periodically. The client can use this information directly to make decisions about when to switch to a higher or a

Bitrate Adaptation in 3GPP

The 3rd Generation Partnership Project (3GPP) is a collaboration between telecommunications associations whose scope is to produce technical specifications for 3G mobile systems based on Global System for Mobile Communications (GSM) networks. Popular push-based streaming servers such as RealNetworks Helix or Apple QuickTime use the bitrate adaptation mechanisms defined in 3GPP standards.

In its Release 6, 3GPP defined mechanisms for the media client to inform the media server about dynamically switching the transmission bitrate.[1,2] The algorithms that adapt the bitrate are implementation-specific. These algorithms collect real-time statistics about available bandwidth and client buffer levels. In a typical implementation, the process goes as depicted in Figure A.

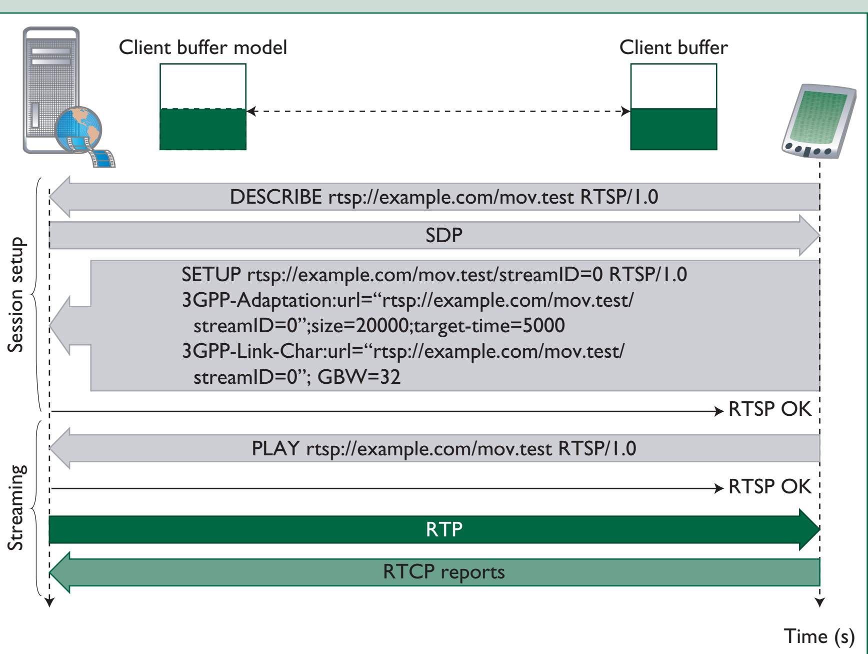

Figure A. Example 3GPP streaming session setup with bitrate adaptation.

During session setup, the server sends a list of available streams and their properties such as bitrate and codec to the client via Session Description Protocol (SDP), specified in RFC 4566, in response to the client's RTSP `DESCRIBE` message. After receiving the session description, the client picks from within the available audio and video streams that best fit with its link speed and decoding capabilities. Then, the client sends RTSP `SETUP` messages (one for each unique audio and video stream) to the server to prepare it to send out the selected streams. The RTSP message header contains a `3GPP-Adaptation` parameter that informs the server about the client's buffer size (size attribute) and minimum required buffering (target-time attribute) to ensure interrupt-free playback. It also contains a `3GPP-Link-Char` parameter that provides information about the client's guaranteed receive bandwidth (GBW attribute).

After the server obtains information about the client's buffer and receive bandwidth, it sets up a buffer model to simulate the changes in the client's receive buffer levels. In this model, the server tries to satisfy the minimum required buffer level while preventing any overflows. Because network conditions and buffering requirements can change over time, the client keeps notifying the server via periodic RTCP receiver reports. These messages carry information such as available free space in the client's receive buffer, playout delay (the time difference between the presentation time of the next frame in the decoder buffer and the time the RTCP message is sent), and estimated network bandwidth. Using this information, the server can maintain an accurate model of the client's receive buffer and make informed decisions about when to switch to a higher- or lower-bitrate stream dynamically.

The upshift and downshift buffer level thresholds for the stream are usually preprogrammed on the server. For instance, if the server detects that the client's buffer level is below the next downshift threshold, it switches to the next lower encoding rate available to prevent the buffer from draining any further. On the other hand, if the buffer level exceeds the next upshift threshold, the server switches to the next higher encoding rate, provided that the network bandwidth can support that rate. If not, the server might choose to slow down the transmission rate to prevent a buffer overflow.

References

1. "Transparent End-to-End Packet-Switched Streaming Service (PSS); Protocols and Codecs," 3GPP TS 26.234, Dec. 2010; http://ftp.3gpp.org/specs/html-info/26234.htm.
2. P. Fröjdh et al., "Adaptive Streaming within the 3GPP Packet-Switched Streaming Service," *IEEE Network*, vol. 20, no. 2, Mar. 2006, pp. 34–40.

lower-bitrate stream, or it can communicate this information along with its buffer levels to the server via receiver reports and let the server make those decisions. Such reports are usually transmitted via RTP Control Protocol (RTCP). See the "Bitrate Adaptation in 3GPP" sidebar for more information.

Pull-Based Media Streaming Protocols

In pull-based streaming protocols, the media client is the active entity

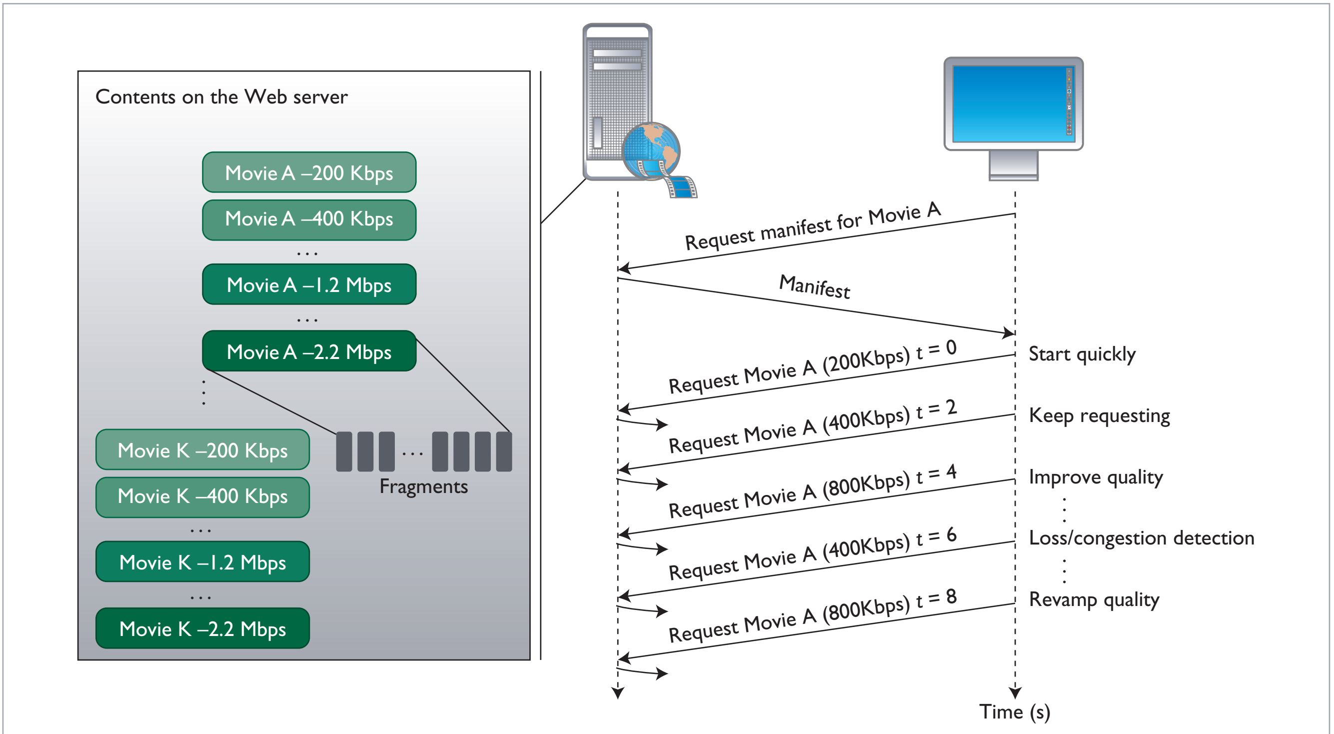

Figure 1. Example bitrate adaptation in pull-based adaptive streaming showing how a client reacts to network conditions.

that requests content from the media server. Therefore, server response depends on the client's requests when the server is otherwise idle or blocked for that client. Consequently, the bitrate at which the client receives the content depends on the client and available network bandwidth. As the Internet's primary download protocol, HTTP is a common protocol for pull-based media delivery.

Progressive download is one of the most widely used pull-based media streaming methods available on IP networks today. In progressive download, the media client issues an HTTP request to the server and starts pulling content from it as quickly as possible. Once the client fills up a minimum required buffer level, it starts playing the media while continuing to download content from the server in the background. As long as the download rate isn't smaller than the playback rate, the client buffer stays at a sufficient level to continue an uninterrupted playback. However, if network conditions degrade, download rate can fall behind the playback rate, which might cause an eventual buffer underflow.

Similar to methods used in push-based streaming, pull-based streaming protocols use bitrate adaptation to prevent buffer underflow. Figure 1 shows an example implementation, where the media content is divided into short-duration media segments (also called fragments), each of which is encoded at various bitrates and can be decoded independently. When the client plays the fragments back to back, it can seamlessly reconstruct the original media stream. During download, the media client dynamically picks the fragment with the right encoding bitrate that matches or is below the available bandwidth and requests that fragment from the server. This way, the client can adapt its media-consumption rate according to the available receive bandwidth.

Although media fragment structures differ among implementations, the basic principle for fragment construction is the same. When audio and video aren't interleaved, each audio frame usually consists of constant duration audio samples in the milliseconds range, and each frame is usually decodable on its own for common audio codecs. Therefore, one can easily stuff audio data into a media fragment by combining a sufficient number of audio frames to match the fragment duration. For video, on the other hand, the frames aren't necessarily independently decodable due to temporal prediction commonly applied between the frames. Therefore, for video, the partitioning for fragment construction is performed at the group of pictures (GoP) boundary instead. In video coding, a GoP is a frame sequence that starts with an intra-coded frame (I-frame) that can be decoded independently, followed by predicted frames that depend on other frames. If the predicted frames within a GoP depend only on the frames within that same GoP, we call it a closed GoP — otherwise, it's an open GoP. Because a closed GoP is self-contained (meaning it can be

decoded independently from other GoPs), one can construct a fragment from it rather straightforwardly. The encoder simply adjusts the number of frames in the GoP such that the total duration of its frames is equal to the desired fragment duration. However, if the fragment duration is large compared to a typical GoP size, we'll want to pack more than one GoP into a fragment.

Let's look more closely at two different implementations of pull-based adaptive streaming protocols that are based on multi-bitrate fragments.

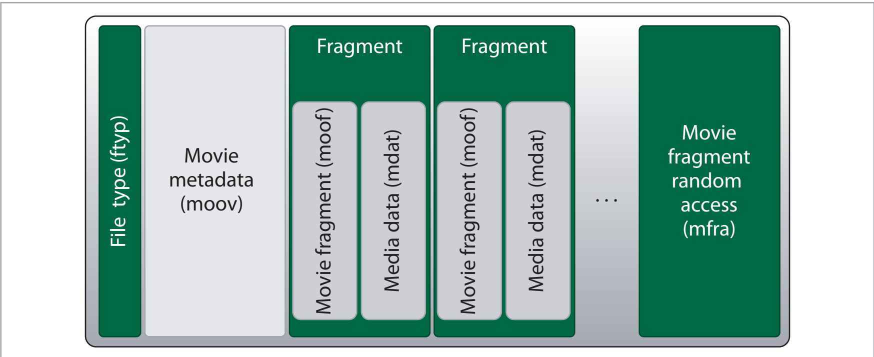

Figure 2. Fragmented MP4 file format structure. The audio and video data are in mdat boxes, which together with the metadata form a fragment that is retrieved in an HTTP GET request. (Image adapted from http://alexzambelli.com/blog/2009/02/10/smooth-streaming-architecture/.)

```
GET /sample/v_720p.ism/QualityLevels(1500000)/Fragments(video=160577243) HTTP/1.1
```

Figure 3. Example HTTP request message for downloading a fragment.

Microsoft Smooth Streaming. Microsoft's Smooth Streaming implementation is based on Protected Interoperable File Format (PIFF; http://go.microsoft.com/?linkid=9682897), which is an extension of the MPEG4 (MP4) file format specification (www.iso.org/iso/catalogue_detail.htm?csnumber=51533). In Smooth Streaming, all fragments with the same bitrate are stored in a single MP4 file. Therefore, each available bitrate has a separate file. Fragments are usually two seconds long and typically contain a single GoP.

Figure 2 shows a fragmented MP4 file. A fragmented MP4 file is composed of a hierarchical data structure. The most basic building block of this hierarchy is called a box; it can contain audio/video data and metadata. Each box type is designated by a four-letter identifier. The file starts with an ftyp box that describes the version information for the specifications with which the file complies, followed by a moov box that describes the media tracks available in the file. The audio/video media data for a single fragment is contained within a box of type mdat. In a fragmented MP4 container structure, an mdat box is immediately preceded by a box of type moof, which contains metadata for that fragment. The moof box can also contain signaling information, such as a sequence counter for the fragment, the number of samples within the fragment, and each sample's duration.

For the client to request a fragment with a specific bitrate and start time, it first needs to know which fragment(s) are available on the server. This information is communicated to the client at the beginning of the session via a client-side manifest file. In addition to bitrates, this file describes codecs, video resolutions, captions, and other auxiliary information for the available streams.

Once the client downloads the manifest file, it uses the information in this file to make HTTP GET requests to the server for the individual fragments. Each fragment is downloaded via a unique HTTP request-response pair. The HTTP request message header contains two pieces of information: the bitrate and the requested fragment's time offset (see Figure 3).

When the server gets an HTTP-encapsulated request from the client for a particular media fragment, it first needs to determine which MP4 file to search for that fragment. This information is contained in another manifest file, the server-side manifest, which maps MP4 files to the bitrates of the fragments they contain. The server looks up the bitrate information it receives from the client in the manifest file and determines the corresponding MP4 file to search.

After the server determines the correct file, the next step is to locate the requested fragment in that file. This is achieved via indexing data. As Figure 2 shows, an MP4 file also contains boxes of type mfra that contain a fragment's location and presentation time. The media server uses the time offset information it receives from the client to find a match with a fragment index in the MP4 file. Once the server locates the fragment in the file, it sends the contained moof box followed by the mdat box, and these make up the fragment on the wire.

Apple HTTP Live Streaming. Apple's HTTP Live Streaming implementation (http://tools.ietf.org/html/draft-pantos-http-live-streaming) follows a different approach for fragment (referred to as media segment in the implementation) storage, which is based on the ISO/IEC 13818-1 MPEG2 Transport Stream file format. As

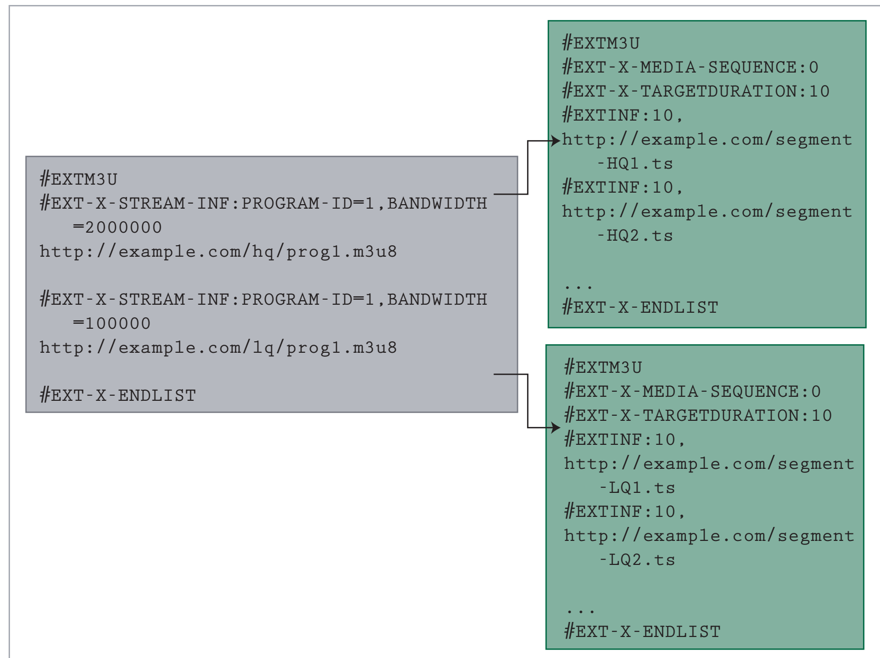

Figure 4. Example showing HTTP Live Streaming playlists.

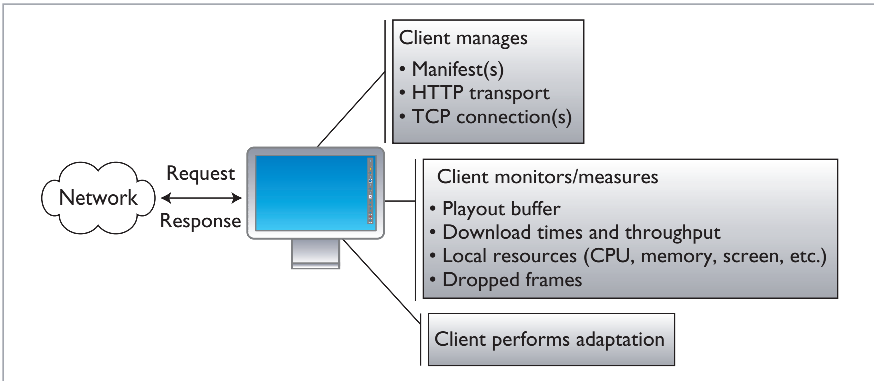

Figure 5. A client-side pull-based adaptive streaming implementation.

opposed to Smooth Streaming, each media segment is stored in a separate transport stream file. A stream segmentation process reads a continuous transport stream file and divides it into equal-duration segments of 10 seconds by default. Longer fragment duration reduces the number of fragments. Longer fragments also allow better compression because they have more temporal redundancy. However, it also reduces the granularity at which fragment-switching decisions can be made, which means it might not be as good in terms of adapting to dynamic bandwidth changes.

Similar to Microsoft's Smooth Streaming implementation, a client-side manifest file keeps a record of the media segments available for the client. The manifest file format is an extension to the MP3 playlist file standard; Figure 4 shows an example of the manifest files organized in a two-level hierarchy. Each file starts with an `EXTM3U` tag that distinguishes it from an ordinary MP3 playlist. The file on the left is a higher-level file that links to two other lower-level files on the right. The link for each lower-level file is specified in a URI that's always preceded by an `EXT-X-STREAM-INF` tag. This tag has an attribute named `BANDWIDTH` that indicates the corresponding lower-level file is a manifest for an alternate encoding of the segments it lists. In the example, the stream has two alternate encodings for each segment — that is, one low-quality (100 Kbps) encoding and one high-quality (2 Mbps) encoding.

The lower-level files on the right provide the links for the individual media segments available for download. Following the `EXTM3U` tag is an `EXT-X-MEDIA-SEQUENCE` tag that lists the sequence number of the first segment in the playlist. Each segment typically has a unique sequence number that changes in single increments, starting from the first segment's sequence number. The URIs for the individual segments are marked with an `EXTINF` tag. This tag has an attribute separated by a semicolon that indicates the duration of the corresponding media segment. In the example, each segment is 10 seconds long.

For fixed-duration streams, `EXT-X-ENDLIST` marks the end of the playlist. This tag doesn't exist in live streams where the playlist can grow dynamically. In the case of live streams, the client must periodically refetch an updated manifest. The period for refetching the manifest depends on whether the manifest file has changed since the last time it was reloaded. If the manifest has changed, the period is the duration of the last media segment in the playlist (which is specified by the `EXTINF` tag). If the manifest hasn't changed, then the period is a multiple of the duration specified by the `EXT-X-TARGET-DURATION` tag. This tag indicates the maximum `EXTINF` value of any media file that can be added to playlist and is constant throughout the manifest file.

Example Functional Diagram for Client-Side Pull-Based Adaptive Streaming

Figure 5 illustrates a generic client-side pull-based adaptive streaming

implementation. At minimum, the server provides standard responses to HTTP GET requests. The client gets a manifest file that identifies files containing media presentations at alternative bitrates. The client acquires media from fragments of a file over one or more connections according to the playout buffer state and other conditions.

The basic client/server adaptive streaming configuration requires the general functions shown on the client-side of Figure 5:

- The client needs a file, called a client-side manifest or a playlist, to map fragment requests to specific files or to map byte ranges or time offsets to files. In some adaptive streaming schemes, a similar file on the server translates client requests.
- Playout buffer management is a basic function required on any adaptive streaming client, to select fragments from a file at a particular bitrate in response to buffer state and potentially other variables. Typically, an adaptive streaming client keeps a playout buffer of several seconds (between five and 30).
- A transport is needed to communicate requests from the client to the server; the pull-based adaptive streaming schemes that we survey in this article use HTTP GET requests, either to a standard HTTP Web server or to a specialized Web services API supported by a special service on the server (such as a Smooth Streaming Transport application running on Microsoft's Internet Information Services server).
- GET requests use a single TCP connection by default, but some adaptive streaming implementations support using multiple concurrent TCP connections for requesting multiple fragments at the same time or for pulling audio and video in parallel.

The client is preconfigured to request a content at a certain bitrate, or profile, based on the result of network tests or a simple configuration script. When a profile is selected and the client finds the URI associated with it in the manifest, it establishes one or more connections to the server. We've observed that different products employ different strategies — for example, some use only one TCP connection for GET requests to a file whereas others open and close multiple TCP connections during an adaptive streaming session. As the client monitors its buffer, it might choose to upshift to a higher-bitrate profile or downshift to a lower one, depending on how much the rate of video transport does or does not exceed the playout rate. Some adaptive streaming products open a new connection when upshifting or downshifting to a new profile.

As the client monitors its buffer, it might choose to upshift to a higher-bitrate profile or downshift to a lower one, depending on how much the rate of video transport does or does not exceed the playout rate.

Alternative Methods for Bitrate Adaptation

We've described adaptive streaming methods that work based on the principle of switching between alternate encodings of a content as a whole or switching between individual fragments of it encoded at several bitrates. An alternative technology called Scalable Video Coding (SVC) lets clients choose media streams appropriate for underlying network conditions and their decoding capabilities. SVC has been standardized as an extension to the H.264/MPEG4 AVC video compression specification; in its current form, it applies to video streams only.

In SVC, a video bitstream is made up of a hierarchical structure of layers. The base layer provides the lowest level of quality in terms of frame rate, resolution, and signal-to-noise ratio (SNR). Each enhancement layer on top of the base layer provides an improvement for one or more of these scalable quality parameters. Enhancement layers can be independently stored or sent over the network. Therefore, we can modify the overall stream bitrate by selectively adding or removing enhancement layers to and from a stream.

The quality knobs in SVC are referred to as temporal, spatial, and SNR scalability. Temporal scalability provides a way to add or drop complete pictures to and from a stream. It isn't a new concept; in its most basic form, traditional push-based streaming servers have used it in terms of a method known as stream thinning. Spatial scalability, on the other hand, encodes video signal at multiple resolutions. The client can use the reconstructed lower-resolution frames to predict and reconstruct higher-resolution frames with the added information sent in the enhancement layers. SNR scalability encodes the video signal at a single resolution but at different quality levels by modifying the encoding precision. Each enhancement layer increases the precision of the lower layers.

Table 1. Comparison between push- and pull-based streaming protocols.

	Push-based	Pull-based
Source	Broadcasters and servers like Windows Media, Apple QuickTime, RealNetworks Helix, Cisco CDS/DCM	Web servers such as LAMP, Microsoft IIS, RealNetworks Helix, Cisco CDS
Protocols	RTSP, RTP, UDP	HTTP
Bandwidth usage	Likely more efficient	Likely less efficient
Video monitoring and user tracking	RTCP for RTP transport	Currently proprietary
Multicast support	Yes	No

A key advantage of SVC lies in its ability to distribute information among various layers with minimal added redundancy. In other words, while a stream that's traditionally encoded at different quality levels has significant redundancy between the encodings, each layer in an SVC-encoded stream has minimal common information between the layers. This makes SVC efficient for media storage at various quality levels. Another advantage of SVC is the graceful degradation of stream quality without client or server intervention when packets from enhancement layers can't be delivered due to abrupt changes in network conditions. This is in contrast to multi-bitrate encoding, which requires switching from one encoding to another via a client or server decision to adjust to changes in network conditions. SVC streams are typically more complex to generate and impose codec restrictions compared to multi-bitrate streams. Thus, the adoption rate for SVC has been rather slow.

Comparison of Push- vs. Pull-Based Streaming Protocols

Table 1 summarizes the differences between push- and pull-based streaming (here, pull-based streaming exclusively refers to streaming over HTTP). One of the main differences between push- and pull-based streaming is the server architecture's complexity. As mentioned earlier, in pull-based streaming, bitrate management is usually a client task, which significantly simplifies the server implementation. Furthermore, pull-based streaming can run on top of HTTP. Therefore, with minor provisions, an ordinary Web server can serve media content in pull-based streaming, although complexities can arise in streaming live content to a large number of clients.

Push-based streaming, on the other hand, requires a specialized server that implements RTSP or a similar purpose-built protocol with built-in algorithms for tasks such as bitrate management, retransmission, and content caching. This may make pull-based streaming more cost-effective compared to push-based streaming.

Despite the lower server costs, pull-based streaming is usually less efficient, overhead-wise, than push-based streaming due to the underlying transport protocol. Compared to HTTP over TCP, RTP imposes a lower transmission overhead. Moreover, because RTP usually runs on top of UDP, the retransmission dynamics and congestion control mechanisms of TCP don't inherently exist in RTP.

Both push- and pull-based streaming protocols allow client buffering both at the beginning of a session and also after trick-mode transitions such as fast forward to play. This is performed to prevent buffer underflows and achieve a smooth playback experience. In adaptive streaming methods, the initial client buffering duration can be substantially less than nonadaptive streaming methods due to the fact the client can begin with a lower-bitrate stream. This allows fast startup and increases responsiveness.

But one of the key benefits of push-based streaming is multicast support. Multicast lets servers send a packet only once to a group of clients waiting to receive that packet. The packet is duplicated along the network path in an optimal way, and a client can join or leave a multicast group on demand. This way, a client can receive only the packets it wants. Pull-based streaming, on the other hand, works based on the unicast delivery scheme, which offers a one-to-one path between the server and each client. Therefore, in the worst case, a server has to send a packet as many times as the number of clients requesting that packet, and, similarly, a network node in the middle has to pass along the same packet multiple times. This reduces network efficiency.

Routing and packet delivery over the network can be made more efficient by using content caching. The content is cached along the network on dedicated cache servers. This way, a client can obtain content from a nearby cache server in the network instead of going all the way up to the origin server. The most efficient use of caching is most likely in pull-based adaptive streaming, where each fragment can be cached in the network independently.

The popularity of watching traditional broadcast TV programming is weakening every day against the Web's onslaught. Consumers can access Web content not just from a TV in the living room but from a variety of devices in a variety of places connected through different types of access networks. Adaptive

streaming methods that can deal with the challenges presented by this variety as well as the scalability of content distribution to large audiences are further accelerating this trend transition, which will certainly impact existing business models and create new revenue opportunities. In the second part of this article, we look into applications for streaming, contrast adaptive approaches with other video delivery paradigms, discuss current standardization efforts, and highlight areas that still require further research and investigation.

Reference

1. G. Thompson and Y.-F.R. Chen, "IPTV: Reinventing Television in the Internet Age," *IEEE Internet Computing*, vol. 13, no. 3, May 2009, pp. 11–14.

Ali C. Begen is with the Video and Content Platforms Research and Advanced Development Group at Cisco. His interests include networked entertainment, Internet multimedia, transport protocols, and content distribution. Begen has a PhD in electrical and computer engineering from Georgia Tech. He is a member of IEEE and the ACM. Contact him at abegen@cisco.com.

Tankut Akgul is in the Service Provider Video Technology Group at Cisco. His research interests are embedded software design, video compression, and multimedia streaming. Akgul has a PhD in electrical and computer engineering from Georgia Tech. Contact him at akgult@cisco.com.

Mark Baugher is in the Research and Advanced Development group at Cisco. He has coauthored several widely used international standards in the IETF and ISMA, and is currently cochairing the Internet Gateway Device Working Committee in the UPnP Forum. Baugher has an MA in computer science from the University of Texas at Austin. Contact him at mbaugher@cisco.com.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.