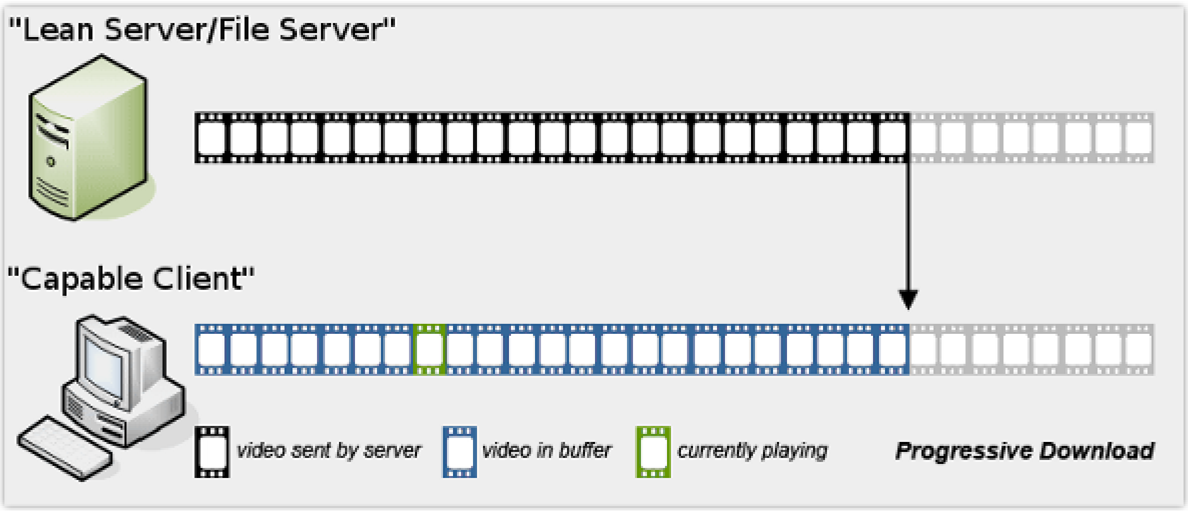
Implementing Adaptive HTTP Streaming Using the Web

TOAST UI Follow
May 31, 2019 · 14 min read

Around the world, people are consuming video contents at a rate that has never been observed before. *Flash*, once the go-to video player technology on the web, has mostly been replaced by the standard HTML5 video, and most of new video streaming services have been built based on the HTML5 specs. During my constant struggle of searching for optimized methods to deliver high quality video streams without buffering, the method that stood out the most was using the existing Hypertext Transfer Protocol (HTTP) to implement the Adaptive HTTP Streaming.

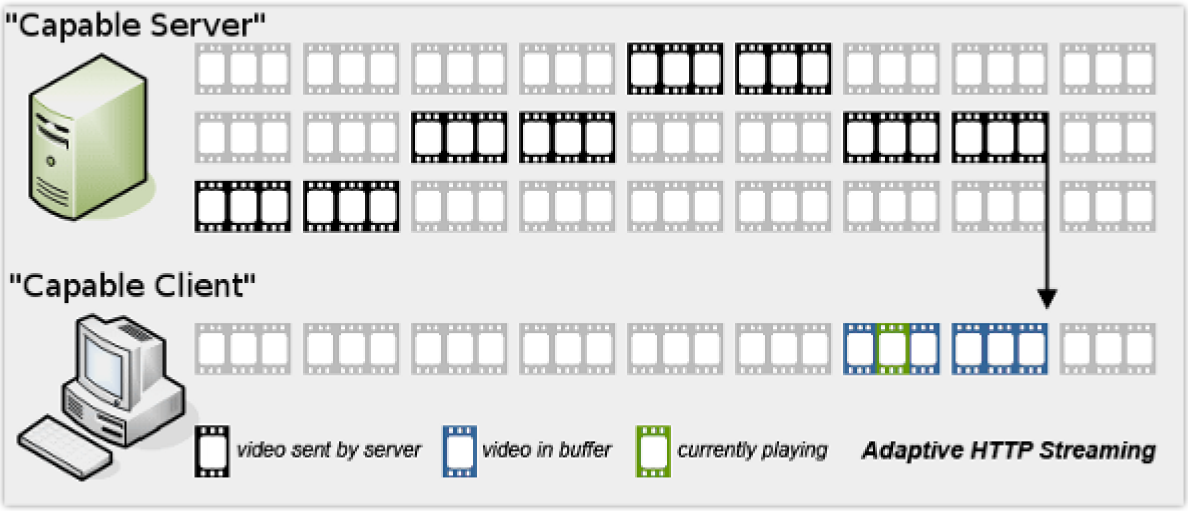
What is Adaptive HTTP Streaming?

Adaptive HTTP Streaming, as the name suggests, is adaptive, and therefore dynamic. The main purpose is to adapt (react) to the user's network status to stream the content. While the Real Time Streaming Protocol (RTSP) / Real Time Messaging Protocol (RTMP) Streaming are based on a similar concept, since the technology was not based on HTTP, it required additional cost and work to maintain the service. The general purpose Progressive Download (PD) plays the selected content once the source for the video has been specified while the content is being downloaded. The HTML5 is capable of using this method to stream videos.



(Image Source: http://abt.net/multimedia.php)

One downfall of PD is that since it pins down one source for the video with a designated resolution, depending on the user's network quality, the user may run into buffering, and if the network quality remains poor, the user constantly has to deal with buffering. Adaptive Streaming, on the other hand, was designed to target this particular problem. The underlying idea is as follows. You encode and store the video content with multiple resolutions, and furthermore, you thinly slice the video into multiple data units, instead of storing the entire video in one big file. Then, when the user plays the video, the stored video strategically uses the network information to provide the user with optimized streaming service. With this kind of technology, because the file has been encoded with multiple sources, the user is given access to optimized and selected source for the file, and because the file has been split into numerous chunks, the user can change the quality of the video with relative ease. For example, if the user's current network is performing rather poorly, the user has the option of streaming bits of the content with 480P, and changing to higher resolution once the network stabilizes.



(Image Source: http://abt.net/multimedia.php)

You may have experienced, when watching Netflix or Youtube, that the quality of the video starts off poor but gradually becomes better. It is a strategy used within the protocol because at first, the network status may be unknown, so the protocol streams with lower quality and once it has identified the network quality, it delivers the appropriate quality content. The benefit of adaptive streaming is clear from both ends of the service provider and the user. The service provider can manage the traffic of the service, and the user can watch videos without interruptive buffering. For the users who are willing to compromise and wait through the buffering for high resolution, the user has the option of doing so using the provided UI.

The Overall Flow

Server-Side

Unlike PD, there are couple of preparations required to implement adaptive streaming. Each video file must be encoded with supporting resolutions, and since there are multiple files, information regarding each chunk must be provided to the client. The overall flow of adaptive streaming is as follows.

1. When uploading the video, the video is sliced into thin segments.
2. Each segment is encoded according to the resolution designated by the requirements of the service. During this process, the number of segments increase proportionately to the number of resolutions available.
3. A Manifest, a file that contains information on corresponding media segment and the respective resolution, is provided to the client.

When the video file is sliced into segments, different tools are used for different Codecs. As for the documents that provide information regarding segmented video file, there are two formats: Apple-HLS and MPEG-DASH.

Client-Side

The standard for allowing Adaptive Streaming over the web client is the Media Source Extensions (MSE), and it is used to transfer streaming data to the player.

1. The client requests a manifest file that contains information regarding segmented data of the video to the server.
2. The client parses the manifest file to receive necessary information including the resolution and quality, and identifies the path to obtain the corresponding segments (e.g. CDN, URL).
3. The client measure the user's bandwidth, and selects the optimized video quality according to the manifest, and downloads it. (The bandwidth is measured again as the segments are being downloaded.)
4. Provides the downloaded segment data to the MSE buffer.
5. MSE decodes the data and provides the video object to the player to stream. (goto 3)

Apple-HLS, MPEG-DASH

The two iconic technologies used to provide Adaptive HTTP Streaming are Apple's own HTTP Live Streaming (HLS) and MPEG's standardized Dynamic Adaptive Streaming over HTTP (DASH). HLS and DASH can be viewed as protocols for the manifest specs that include the contents information. While they both have different pros and cons, DASH is preferred due to the extensible nature and the open structure.

Apple-HLS

Developed with the Mac products as primary targets, HLS is natively supported in Mac's Safari as well as in mobile Safari, and it directly uses the HTML5 video source as HLS Manifest file to stream the video. Because it was built to mainly support Apple products, it lacks cross product support, but it is also supported in Microsoft Edge and Android browsers natively. Since HLS is supported natively, if you use the HTML5 Video source as HLS Manifests, you can stream the video adaptively without much extra work. However, such native implementation falls short in that it cannot be controlled strategically within the service.

Previously, according to the specs, only MP2TS was allowed for a media container. The MP2TS container had a problem that as the size of the segment increased due to the packet header, the entire overhead became heavier due to the header, and it also suffered compatibility issues in Codec browsers. Especially since Chrome did not support MP2TS, to normally play videos over MP2TS,the file had to be converted into mp4 using a process known as a demux. However, in 2016, the specs for the HLS changed to allow the usage of MP4 containers. The HLS uses the M3U8 playlist, used to create mp3 music files using the manifest. Because M3U8 documents the types of contents using a format that was restricted in terms extensibility, it presented the content information in main M3U8 and a sub M3U8 structure.

- Developed from Apple
- In certain browsers including Safari, HLS could be used directly using the media source from the HTML5 Video.
- Since mobile Safari does not support MSE, only HLS can be used.
- Media Container Format : mp2ts, mp4 (2016)
- Uses M3U8 for manifests

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000, RESOLUTION=720x480
http://ALPHA.mycompany.com/lo/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000, RESOLUTION=720x480
http://BETA.mycompany.com/lo/prog_index.m3u8

#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=500000,
RESOLUTION=1920x1080
http://ALPHA.mycompany.com/md/prog_index.m3u8
```

```
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=500000,
RESOLUTION=1920x1080
http://BETA.mycompany.com/md/prog_index.m3u8
....Continues....
```

MPEG-DASH

The MPEG-DASH is a standard format agreed upon by MPEG and ISO. In other words, it has not been developed by a certain vendor, and can be used as a standard. Since there are no browsers that natively support the DASH, there are no browsers that allows for a direct implementation of the HTML5 video source without performing preparation tasks beforehand. In order to use it, you have to configure the adaptive streaming as specified by the service using the MSE, which will be explained in greater detail later. The main characteristics of this protocol is that there are no restrictions on media formats, and since the Media Presentation Description (MPD,) the manifest, is built based on the XML, it is descriptive enough to present not only the main content but also additional adds within the video. Due to the descriptiveness of the MPD, unlike M3U8, it is able to retain all of the information in one manifest file.

- Standard agreed by MPEG and ISO.
- DASH is able to use MSE to adopt the browser’s native play-functionality.
- DASH is not restricted by the media container format.
- Addition of ads is easy and intuitive.
- Because the manifest is based on XML, it is capable of great descriptiveness, and provides wide range of information using one MPD. (Including the DRM information)

```
<?xml version="1.0"?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
profiles="urn:mpeg:dash:profile:full:2011" minBufferTime="PT1.5S">
    <!-- Ad -->
    <Period duration="PT30S">
        <BaseURL>ad/</BaseURL>
        <!-- Everything in one Adaptation Set -->
        <AdaptationSet mimeType="video/mp2t">
            <!-- 720p Representation at 3.2 Mbps -->
            <Representation id="720p" bandwidth="3200000"
width="1280" height="720">
                <!-- Just use one segment, since the ad is only 30
seconds long -->
                <BaseURL>720p.ts</BaseURL>
                <SegmentBase>
                    <RepresentationIndex sourceURL="720p.sidx"/>
                </SegmentBase>
            </Representation>
            <!-- 1080p Representation at 6.8 Mbps -->
            <Representation id="1080p" bandwidth="6800000"
width="1920" height="1080">
                <BaseURL>1080p.ts</BaseURL>
                <SegmentBase>
                    <RepresentationIndex sourceURL="1080p.sidx"/>
                </SegmentBase>
            </Representation>
....Continues....
```

Media Source Extensions (MSE)

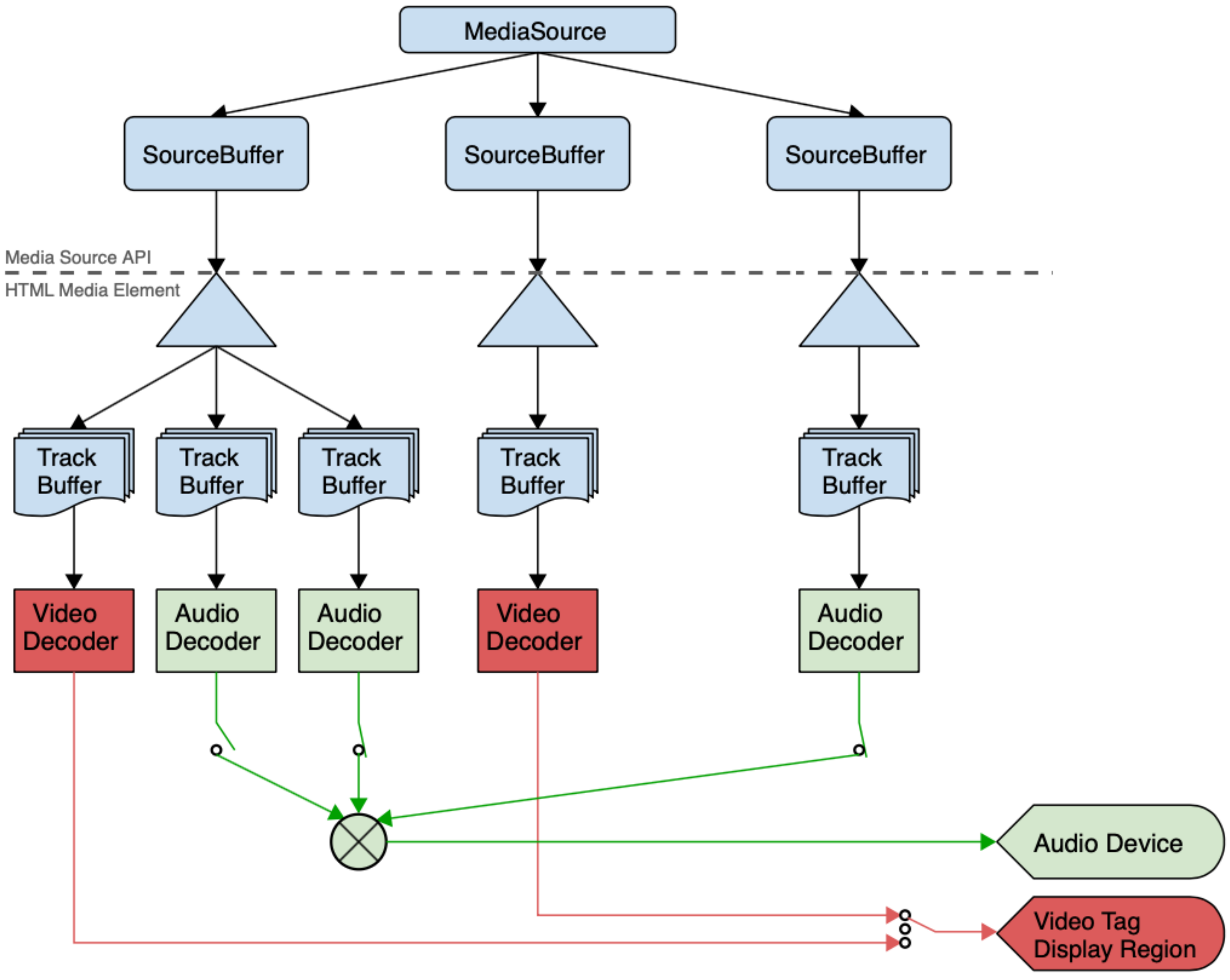
While for HLS, the media source can be used directly on both Safaris, for DASH, generally MSE is used to manually extend the media source. Actually, both DASH and HLS are built to provide the data of the streamable video to the video player, it is not directly related to streaming. MSE uses the DASH or HLS manifest to obtain the necessary media information and is used directly by the HTML5 video to stream the content. (In Safari, the manifest can be used as the source without alteration.) MSE is a new interface that allows developers to directly manipulate the source for the HTML5 video using the `HTMLMediaElement` instead of the `source` tag. The developer first takes the data of the to-be-played video over the HTTP, and uses the SourceBuffer object to provide the HTMLMediaElement with the necessary buffer chunk. This method gives developers control over providing data required to play an HTML5 video. Before we take a look at the actual implementation of MSE, let's go over couple of core concepts.

Segments

Segments are thin slices of encoded video data. The information regarding this chunk is obtained by DASH or HLS, and each segment can be provided to the users to view. There are two different types of segments: **initialization segment** and the **media segment**. The **initialization segment** contains information required to decode the media segment that contains actual video information, and the initialization segment includes codec initialization data, tracking ID, and timestamp offset. The **media segment** is a packet version of the actual video that includes the respective timestamp on the media timeline. The media segment is aware of its place on the media timeline, so it is not necessary to provide the segments sequentially. Therefore, without the initialization segment, no matter how much media segment data you throw at users, it will not be able to be played.

MediaSource Object and SourceBuffer Object

`MediaSource` is the source of media data for the `HTMLMediaElement`. You can view it as a type of media that is played on the video player. The developer uses the `SourceBuffer` to pass the media segment to `MediaSource`, and the `HTMLMediaElement` obtains necessary data from the `MediaSource` to play the video. Structurally, `HTMLMediaElement` uses the `MediaSource` and every `MediaSource` has a `SourceBuffer`.



(Image Source: https://www.w3.org/TR/media-source/)

Connecting the HTML5 Video with MSE

Upon googling, I found that there are only few documents on MSE, and since it is a relatively new specification, in order to fully understand MSE, it is best to turn to the W3C spec document. Fortunately, the spec documentation is concise and readable. If you are looking for information on just using the MSE interface, feel free to skip past the algorithmic documentation. The overall content of the spec documentation can be seen generally at the bottom of the page with example codes. I will copy and past the example codes with a comment to explain each line, but first, let's look at some of the crucial points.

The main piece of the MSE, technology that provides the streaming source to the Video (HTMLMediaElement), is the MediaSource. Therefore, it is our first task to create the MediaSource object, and then we can connect it to the Video object.

```
var mediaSource = new MediaSource();
video.src = window.URL.createObjectURL(mediaSource);
```

The Video and MediaSource are connected using the Object URL, and window.URL.createObjectURL function can be used to create the Object URL for MediaSource Object. Then, once it is connected to the Video, the MediaSource object emits a `sourceopen` that lets you know that it is ready to receive streaming data, and with that, additional tasks can be assigned. Once the MediaSource is fully prepared, let's create a SourceBuffer object, and then repeatedly bring in segments and transmit the streaming data to Video through the SourceBuffer.

```
var sourceBuffer = mediaSource.addSourceBuffer('video/webm;
codecs="vorbis,vp8"');
```

`addSourceBuffer` takes the codec information as the input and returns the SourceBuffer object that can decode the respective codec. Later, the task of providing the SourceBuffer with the media segment information from the server is carried out by SourceBuffer object's appendBuffer method. The response type for when using Ajax to receive media segment data is the ArrayBuffer.

```
var xhr = new XMLHttpRequest;
xhr.open('get', url);
xhr.responseType = 'arraybuffer';
xhr.onload = function () {
  sourceBuffer.appendBuffer(xhr.response);
};
xhr.send();
```

After using the appendBuffer, the MediaSource initiates a task of decoding the data internally, and during this stage, no new buffer data can be provided. Then, we can use the Video object's progress events and etc. to constantly receive buffers to provide it.

So far, we have briefly looked at the flow of the core interfaces. Now, let's go over the example code in the W3C spec documentation with comments.

```
<video id="v" autoplay> </video>

<script>
  var video = document.getElementById('v');

  // Create a new MediaSource.
  var mediaSource = new MediaSource();

  // When the MediaSource is connected to the Video and is ready to
receive streaming data, it emits a sourceopen event.
  mediaSource.addEventListener('sourceopen', onSourceOpen.bind(this,
video));

  // Connect the MediaSource that we just created to the video
object.
  video.src = window.URL.createObjectURL(mediaSource);

  // This handler will run once the mediaSource is open.
  function onSourceOpen(videoTag, e) {
    var mediaSource = e.target;
    // Disregard any unnecessary events from the sourceopen. We need
a sourceBuffer.
    if (mediaSource.sourceBuffers.length > 0)
        return;

    // Use the addSourceBuffer method from the mediaSource to create
a sourceBuffer. It will take the codec information as the input.
    // In the example code, the sourceBuffer can take the encoded
data using the webm
    var sourceBuffer = mediaSource.addSourceBuffer('video/webm;
codecs="vorbis,vp8"');

    // Filter the handlers as required by the video object's needs.
    videoTag.addEventListener('seeking', onSeeking.bind(videoTag,
mediaSource));
    videoTag.addEventListener('progress', onProgress.bind(videoTag,
mediaSource));

    // Use the application code to receive the initialization
segment. While the process will the asynchronous,
    // for both initialization and media segments, you need to set
the response time as "arrayBuffer" when making an
    // ajax request.
    var initSegment = GetInitializationSegment();

    if (initSegment == null) {
      // If the initialization segment is null, the video cannot be
played.
      // Use the mediaSource.endOfStream method to end stream. This
method is called when the stream has finished
      // under normal circumstances  and also under errors. If the
stream has finished due to an error, it passes the
      // cause as an input. "network" or "decode" error
      mediaSource.endOfStream("network");
      return;
    }

    // Provides the initialization segment to sourceBuffer.
    // Once the firstAppendHandler initialization segment has
successfully been inserted into the sourceBuffer, this
    // event handler is disregarded. It is just a one-time function
used to insert initialization segment and then to
    // transition into media segments.
    var firstAppendHandler = function(e) {
      var sourceBuffer = e.target;
      sourceBuffer.removeEventListener('updateend',
firstAppendHandler);
```

```
      // Use the following function to start providing media segments
to the sourceBuffer.
      appendNextMediaSegment(mediaSource);
    };

    // Once the sourceBuffer receives the media data, the
sourceBuffer starts decoding the received data, and when
    // the update task has finished, the updateend event occurs
regardless of success or failure of the task.
    // Here, it is used briefly to provide the initialization segment
and the media segment.
    sourceBuffer.addEventListener('updateend', firstAppendHandler);
    sourceBuffer.appendBuffer(initSegment);
  }


  // It is the function used to provide the media segment after the
initialization segment has been provided.
  // After the first execution, the code runs according to the video
object's progress event.
  function appendNextMediaSegment(mediaSource) {
    // The MediaSource.readyState has three possible values: "open",
"closed", and "ended".
    // During "open," it is currently processing media data, and
"ended" is the same as a wait status.
    // When it is "closed," it can no longer take any media streams.
    if (mediaSource.readyState == "closed")
      return;


    // If there are no more media segments to provide using the
application code, use the endOfStream to finish streaming.
    if (!HaveMoreMediaSegments()) {
      mediaSource.endOfStream();
      return;
    }

    // Since the process of providing the video buffer requires data
decoding, it takes time and CPU cost.
    // Therefore, you must always check that the sourceBuffer is
updating to provide it with a new buffer.
    // If the updating is true, it means that it is still processing
the previous media data.
    if (mediaSource.sourceBuffers[0].updating)
        return;

    // The application code used to get next media segment.
    var mediaSegment = GetNextMediaSegment();

    if (!mediaSegment) {
      // If the media Segment does not exist, it will throw out an
error.
      mediaSource.endOfStream("network");
      return;
    }

    // Provides media data to sourceBuffer.
    // If the MediaSource.readyState is "ended," it will become
"open" again, and the onSourceOpen handler from the
    // sourceopen event can occur again, so you need to take care of
it.
    mediaSource.sourceBuffers[0].appendBuffer(mediaSegment);
  }

  //Use the seeking event handler to provide the media data to the
resulting location.
  function onSeeking(mediaSource, e) {
    var video = e.target;

    // Cancels any buffers being processed in the sourceBuffer.
    if (mediaSource.readyState == "open") {
      mediaSource.sourceBuffers[0].abort();
    }
    // Uses the application code to read the current location of the
video object, and prepares the respective media segment.
    SeekToMediaSegmentAt(video.currentTime);

    // Provides the MediaSource with the updated buffer.
    appendNextMediaSegment(mediaSource);
  }
  // Uses the progress event handler to prepare the new playable
segment data, and provides it to the sourceBuffer.
  function onProgress(mediaSource, e) {
    appendNextMediaSegment(mediaSource);
  }
</script>
```

To briefly summarize the flow of the example code is as follows.

1. Create a new instance of MediaSource and connect it to the Video object using the Object URL.
2. If the MediaSource is ready, the sourceOpen event occurs.
3. Prepare the sourceBuffer to be used in the MediaSource to decode the codec data.
4. Provide the sourceBuffer with an initialization segment, and once the decoding is done, provide it with the media segment.
5. Then, Provide the corresponding media segment according to the progress and seeking event with respect to the video object's timeline.

While the example code only discusses purely using the MSE, in real world situations, you would need to include code to download and parse the manifest from the DASH or HLS form to obtain the necessary media information. Furthermore, you must perform a bandwidth check to determine the optimal quality. Truthfully, checking the bandwidth to select the optimal resolution is a tricky task. Measuring the bandwidth from the client can have deviate quite a lot (especially on the mobile,) so you must carefully consider how to manipulate and stabilize the data. Generally, the EWMA Control Charts is used to assume a placeholder.

Afterword

So far, I have discussed implementing the Adaptive HTTP Streaming using only the web. Furthermore, now the MSE has evolved to cover DRM by connecting to the Encrypted Media Extensions. In the past, without Flash or Silverlight, it would have been impossible to implement such tasks, and now, it can be implemented using only the web standards. As a Front-end developer, looking at the rapidly increasing speed of innovation is kind of scary, but at the same time, I am incredibly excited.

TOAST UI :: Make Your Web Delicious!

Chrome Internet Explorer Edge Safari Firefox Yes 8+ Yes Yes Yes Data Can Be Displayed in Any Format Styling The TOAST…

ui.toast.com

Originally posted at Toast Meetup written by Sungho Kim.

Mse JavaScript Streaming Video Html5