

OCTOBER 10, 2022 (ABOUT 3 YEARS AGO)

Gone in a Flash: a brief history of HTTP Adaptive Streaming



BY [CHRISTIAN PILLSBURY](#) • 16 MIN READ • [VIDEO EDUCATION](#)

HTTP Adaptive Streaming (HAS) is unquestionably and by orders of magnitude the dominant way of streaming media on the internet. Part history, part “hindsight is 20/20” revisionist history, and part technical discussion, the goal of this post is to give an understanding of **any** HAS implementation and how all of them address the same set of problems and solve them in the same way.

By the mid-2000s, streaming media on the internet was very clearly on the rise. In 2002, MLB.com streamed its first baseball game to an audience of 30,000, and within just a couple of years, Major League Baseball Advanced Media (MLBAM) was helping CBS stream the 2005 NCAA basketball tournament. Likewise, by mid-2006, YouTube was seeing an average of 100 million views per day and was one of the fastest growing sites on the web.



In other words, watching videos on the internet (often called OTT or “Over the Top”) was growing, growing fast, and wasn’t showing any signs of tapering off anytime soon. And while these and other examples were showing both technical and even financial success, this rapid growth was also exposing some real problems with the existing technologies’ ability to keep up.

Storm’s a-brewin’ (aka The Problems)

Many things started applying pressure to come up with some new solutions to OTT streaming (including e.g. security considerations), but there were three distinct yet interrelated problems that loomed especially large.



That bank (aka Cost)

Streaming media on the internet was **expensive**. Given how much data is compressed in video compared to other types of content, it should come as no surprise that storing, processing, and shipping that data cost a lot of

money. However, based on the way the underlying technology worked, it was very hard to make those different moving pieces more cost efficient.

Moreover, the existing standards such as RTMP relied on highly specialized servers (and highly specialized clients), which meant folks either needed to build, deploy, improve, and maintain these highly specialized servers (and pay for the highly specialized teams to do the work), or to pay for expensive “off the shelf” licensed solutions like Adobe Media Server (which still needed folks to build, deploy, improve, and maintain them).

Too big to succeed (aka Scale)

Related to and a major contributor to cost was the way the existing technologies scaled. At its core, the reason scaling was a problem was because the underlying technologies (understandably) did not anticipate just how many folks would be watching TV on the internet, potentially all across the world.

RTMP and protocols like it were inherently and deeply session based and stateful between client and server. Because of this, the most common solutions to scale were similar to any stateful architecture. You could spin up beefier servers that could handle the load (but even here, only up to a point). You could spin up multiple redundant servers that could be pooled for requests, which was complex at best and not realistically possible at worst for live media content. None of these solutions were especially great and they also added to the cost. Unfortunately, there weren't many

MUX



Everything, everywhere, all at once (aka Reach)

The internet was changing quickly. Home networks and WiFi routers had already started to become commonplace, which meant issues of bridging private and public networks were becoming commonplace. Network security was finally starting to be taken seriously and led to the rise of

firewalls on these personal routers and home computers, which meant issues of blocked ports and protocols were becoming commonplace. Both of these presented problems with the existing streaming media technologies such as RTMP.

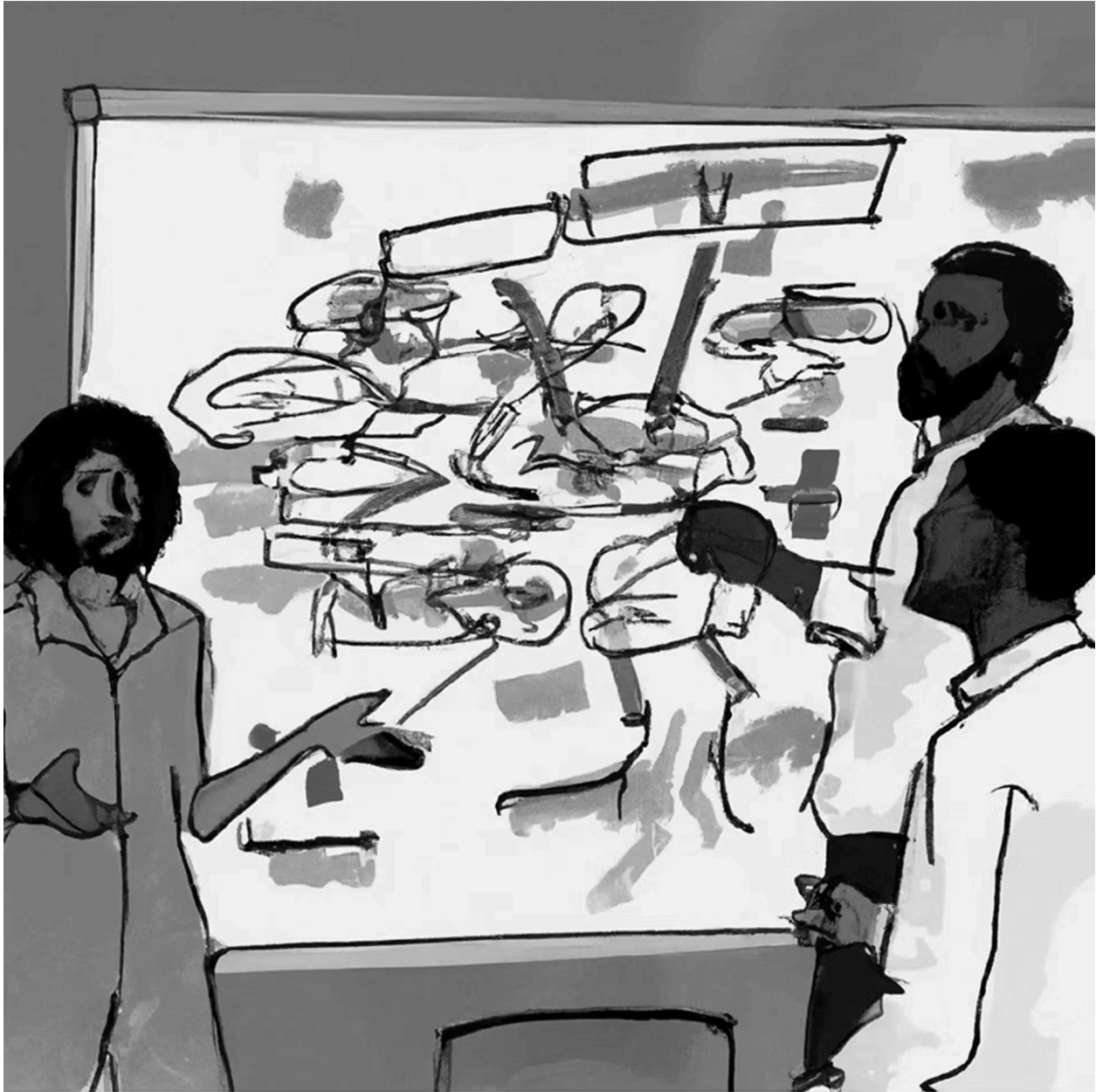
Not only that, more people in more places were getting internet connections, yet the infrastructure for high speed internet wasn't anywhere near as widespread. In the United States alone, by early 2005 only 30% of Americans on the internet had any kind of broadband connection, many of whom were on the comparatively much slower ADSL (Asymmetric Digital Subscriber Line) "broadband" of the time, which capped out between 1.5-9mbps under the most ideal conditions. This was a problem for things like RTMP because it wasn't designed to "adapt" the size of the encoded and streamed content based on the very specific conditions of a very specific client/player. Instead, there was just "one version" of the media. This also meant that handling multiple subtitles or audio dubs was complex when solvable at all, and, of course, heavily impacted cost and scale concerns.

Finally, more and more types of devices were "connected," and more and more environments were being used for playback. While this time predated today's world of modern smartphones and tablets and the "Internet of Things", folks were still already trying to watch video on the internet using things like their WAP-enabled (Wireless Application Protocol) phones with less than stellar results.

A ray of sunshine (aka HTTP Adaptive Streaming, or "HAS")

There was a growing consensus that the prevailing technologies for streaming media weren't going to cut it and would likely need to be rethought (almost) entirely whole cloth. I've focused primarily on RTMP, which was especially popular, but at their root, all of the dominant standards and implementations broadly resembled one another and resulted in the same sets of problems (and many, though not all, of these

are cropping back up in “realtime” protocols such as WebRTC, but that’s an entire blog post unto itself). Not only that, given the volume of people and dollars that were solving these problems, there was enough market pressure to prompt major changes quickly.



And it did. In late 2008, just a couple of years after the explosive growth mentioned earlier, Microsoft announced production-ready support for a new, HTTP-based adaptive streaming media standard in IIS, called (Microsoft) Smooth Streaming (MSS). In just over 2 years, Apple’s HTTP Live Streaming (HLS), Adobe’s HTTP Dynamic Streaming (HDS), and the Motion Picture Expert’s Group’s (MPEG) Dynamic Adaptive Streaming over HTTP (DASH) had all been released. While there were plenty of differences

between these standards, they're all strikingly similar in their core implementation details.

How stuff works (aka Shared Implementation Details)

1. They each used one or more text files to describe relevant aspects of the media content as a whole. These are often called **manifest** files (or **playlist** files in the case of HLS), and would be the first thing a client/player requests from a media server.
2. Different types of related media content could be described in the manifests. Perhaps the most prevalent (though not only) example are providing different **renditions** (also called **representations**) of the same audio or video content that a client/player could choose from based on e.g. network conditions.
3. All media content was made available as a series of **segments**, or small time slices of the media, that could be individually requested, buffered, and played by the client/player.
4. Each of these used HTTP GET requests as the primary protocol and method for client/server interaction, both for the manifests and the media segments.

```

<?xml version="1.0" encoding="UTF-8"?>
<SmoothStreamingMedia
  MajorVersion="2"
  MinorVersion="0"
  Duration="2300000000"
  TimeScale="1000000"
>
  <Protection>
    <ProtectionHeader SystemID="{9A04F079-9840-4286-AB92E65BE0885F95}">
      <!-- Base 64-Encoded data omitted for clarity -->
    </ProtectionHeader>
  </Protection>
  <StreamIndex
    Type="video"
    Chunks="115"
    QualityLevels="2"
    MaxWidth="720"
    MaxHeight="480"
    TimeScale="1000000"
    Url="QualityLevels({bitrate},{CustomAttributes})/Fragments(video={start_time})"
    Name="video"
  >
    <QualityLevel
      Index="0"
      Bitrate="1536000"
      FourCC="WVC1"
      MaxWidth="720"
      MaxHeight="480"
      CodecPrivateData="270000010FCBEE1670EF8A16783BF180C9089CC4AFA11C0000010E1207F840"
    >
      <CustomAttributes>
        <Attribute Name="Compatibility" Value="Desktop" />
      </CustomAttributes>
    </QualityLevel>
    <QualityLevel
      Index="5"
      Bitrate="307200"
      FourCC="WVC1"
      MaxWidth="720"
      MaxHeight="480"
      CodecPrivateData="270000010FCBEE1670EF8A16783BF180C9089CC4AFA11C0000010E1207F840"
    >
      <CustomAttributes>
        <Attribute Name="Compatibility" Value="Handheld" />
      </CustomAttributes>
    </QualityLevel>
    <c t="0" d="19680000" />
    <c n="1" t="19680000" d="8980000" />
  </StreamIndex>
</SmoothStreamingMedia>

```

Microsoft Smooth Streaming (IMSC) Manifest File

```

#EXTM3U
#EXT-X-VERSION:5
#EXT-X-INDEPENDENT-SEGMENTS

#EXT-X-STREAM-INF: BANDWIDTH=2516370, AVERAGE-BANDWIDTH=2516370, CODECS="mp4a.40.2, avc1.640020", RESOLUTION=1280x720
v0/2516370.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=1640580, AVERAGE-BANDWIDTH=1640580, CODECS="mp4a.40.2, avc1.640020", RESOLUTION=960x540
v0/1640580.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=4755240, AVERAGE-BANDWIDTH=4755240, CODECS="mp4a.40.2, avc1.64002a", RESOLUTION=1920x1080
v0/4755240.m3u8

```

```
#EXTM3U
#EXT-X-TARGETDURATION:6
#EXT-X-VERSION:3
#EXT-X-PLAYLIST-TYPE:VOD
#EXTINF:5,
https://fake.mux.com/1640580/0.ts
#EXTINF:5,
https://fake.mux.com/1640580/1.ts
#EXTINF:5,
https://fake.mux.com/1640580/2.ts
#EXTINF:5,
https://fake.mux.com/1640580/3.ts
#EXTINF:5,
https://fake.mux.com/1640580/4.ts
#EXTINF:5,
https://fake.mux.com/1640580/5.ts
#EXTINF:5,
https://fake.mux.com/1640580/6.ts
#EXTINF:5,
https://fake.mux.com/1640580/7.ts
#EXTINF:5,
https://fake.mux.com/1640580/8.ts
#EXTINF:5,
https://fake.mux.com/1640580/9.ts
#EXTINF:4.4176109999999994,
https://fake.mux.com/1640580/10.ts
#EXT-X-ENDLIST
```

```
#EXTM3U
#EXT-X-TARGETDURATION:6
#EXT-X-VERSION:3
#EXT-X-PLAYLIST-TYPE:VOD
#EXTINF:5,
https://fake.mux.com/2516370/0.ts
#EXTINF:5,
https://fake.mux.com/2516370/1.ts
#EXTINF:5,
https://fake.mux.com/2516370/2.ts
#EXTINF:5,
https://fake.mux.com/2516370/3.ts
#EXTINF:5,
https://fake.mux.com/2516370/4.ts
#EXTINF:5,
https://fake.mux.com/2516370/5.ts
#EXTINF:5,
https://fake.mux.com/2516370/6.ts
#EXTINF:5,
https://fake.mux.com/2516370/7.ts
#EXTINF:5,
https://fake.mux.com/2516370/8.ts
#EXTINF:5,
https://fake.mux.com/2516370/9.ts
#EXTINF:4.4176109999999994,
https://fake.mux.com/2516370/10.ts
#EXT-X-ENDLIST
```

```

<?xml version="1.0" encoding="utf-8"?>
<MPD
  mediaPresentationDuration="PT634.566S"
  minBufferTime="PT2.00S"
  profiles="urn:hbbtv:dash:profile:isoff-live:2012,urn:mpeg:dash:profile:isoff-live:2011"
  type="static"
  xmlns="urn:mpeg:dash:schema:mpd:2011"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011 DASH-MPD.xsd"
>
  <BaseURL>./</BaseURL>
  <Period>
    <AdaptationSet
      mimeType="video/mp4"
      contentType="video"
      subsegmentAlignment="true"
      subsegmentStartsWithSAP="1"
      par="16:9"
    >
      <SegmentTemplate
        duration="120"
        timescale="30"
        media="$RepresentationID$/$RepresentationID$_$Number$.m4v"
        startNumber="1"
        initialization="$RepresentationID$/$RepresentationID$_0.m4v"
      />
      <Representation
        id="bbb_30fps_1024x576_2500k"
        codecs="avc1.64001f"
        bandwidth="3134488"
        width="1024"
        height="576"
        frameRate="30"
        sar="1:1"
        scanType="progressive"
      />
      <Representation
        id="bbb_30fps_1280x720_4000k"
        codecs="avc1.64001f"
        bandwidth="4952892"
        width="1280"
        height="720"
        frameRate="30"
        sar="1:1"
        scanType="progressive"
      />
      <Representation
        id="bbb_30fps_1920x1080_8000k"
        codecs="avc1.640028"
        bandwidth="9914554"
        width="1920"
        height="1080"
        frameRate="30"
        sar="1:1"
        scanType="progressive"
      />
    </AdaptationSet>
    <AdaptationSet
      mimeType="audio/mp4"
      contentType="audio"
      subsegmentAlignment="true"
      subsegmentStartsWithSAP="1"
    >
      <Accessibility schemeIdUri="urn:tva:metadata:cs:AudioPurposeCS:2007" value="6" />
      <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main" />
      <SegmentTemplate
        duration="192512"
        timescale="48000"
        media="$RepresentationID$/$RepresentationID$_$Number$.m4a"
        startNumber="1"
        initialization="$RepresentationID$/$RepresentationID$_0.m4a"
      />
      <Representation
        id="bbb_a64k"
        codecs="mp4a.40.5"

```

```
        bandwidth="67071"
        audioSamplingRate="48000"
    >
    <AudioChannelConfiguration
        schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011"
        value="2"
    />
</Representation>
</AdaptationSet>
</Period>
</MPD>
```

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns="http://ns.adobe.com/f4m/1.0" version="3.0">
  <id>myvideo</id>
  <label>English</label>
  <lang>en</lang>
  <streamType>recorded</streamType>
  <duration>100</duration>
  <mimeType>video/mp4</mimeType>
  <baseUrl>http://example.com/</baseUrl>
  <bootstrapInfo profile="named" id="boot1" fragmentDuration="4">
    (BASE64 encoding of bootstrap information)
  </bootstrapInfo>
  <bootstrapInfo profile="named" id="boot2" fragmentDuration="4">
    (BASE64 encoding of bootstrap information)
  </bootstrapInfo>
  <bootstrapInfo profile="named" id="boot3" fragmentDuration="4">
    (BASE64 encoding of bootstrap information)
  </bootstrapInfo>
  <bootstrapInfo profile="named" id="boot4" fragmentDuration="4">
    (BASE64 encoding of bootstrap information)
  </bootstrapInfo>
  <bootstrapInfo profile="named" id="boot5" fragmentDuration="4">
    (BASE64 encoding of bootstrap information)
  </bootstrapInfo>
  <media url="myvideo_250" bitrate="250" bootstrapInfoId="boot1" />
  <media url="myvideo_500" bitrate="500" bootstrapInfoId="boot1" />
  <media url="myvideo_900" bitrate="900" bootstrapInfoId="boot1" />
  <media url="myvideo_1300" bitrate="1300" bootstrapInfoId="boot2" />
  <media url="myvideo_2100" bitrate="2100" bootstrapInfoId="boot3" />
  <media
    url="myvideo_audio1"
    bitrate="192"
    bootstrapInfoId="boot4"
    type="audio" label="Espanol"
    lang="es"
    alternate="true"
  />
  <media
    url="myvideo_audio2"
    bitrate="192"
    bootstrapInfoId="boot5"
    type="audio"
    label="Chinese"
    lang="zh"
    alternate="true"
  />
</manifest>

```

A reasonable question in response to this can be captured in a single word: Why? Why did every solution work basically the same way, at least in the core details? Why did each of these very large, well funded institutions all end up implementing different versions of the “same thing” (this includes DASH, which, while released under the ISO/IEC, had many large

companies backing it, and even small ones like Digital Primates, where I used to work)? Why were these shared implementation details **the** right solution, especially when there were clear and obvious tradeoffs (overhead and speed of packet delivery being the most glaring) compared to what came before?

For the rest of this blog post, we're going to dive into the last point, why HTTP ended up being the de facto standard for HAS, since all of the other 3 are built on top of what was possible by moving to HTTP.

Square pegs in round holes? (aka Why HTTP?)

At the time, HTTP may not have been an obvious choice. As a protocol, it was radically different from things like RTMP. To take just one such example, there was nothing that even remotely resembled a “stream” in HTTP in 2008, let alone anything that was designed specifically to help with streaming **media**. Not only that, at the time, by far the vast majority of “media” HTTP was delivering to clients were image files in web pages. While HTML5 (which would introduce the `<audio/>` and `<video/>` tag to HTML) was co-evolving and certainly on people's radar, its first draft wasn't until mid-2007 and its working draft predates MSS by only a few months.

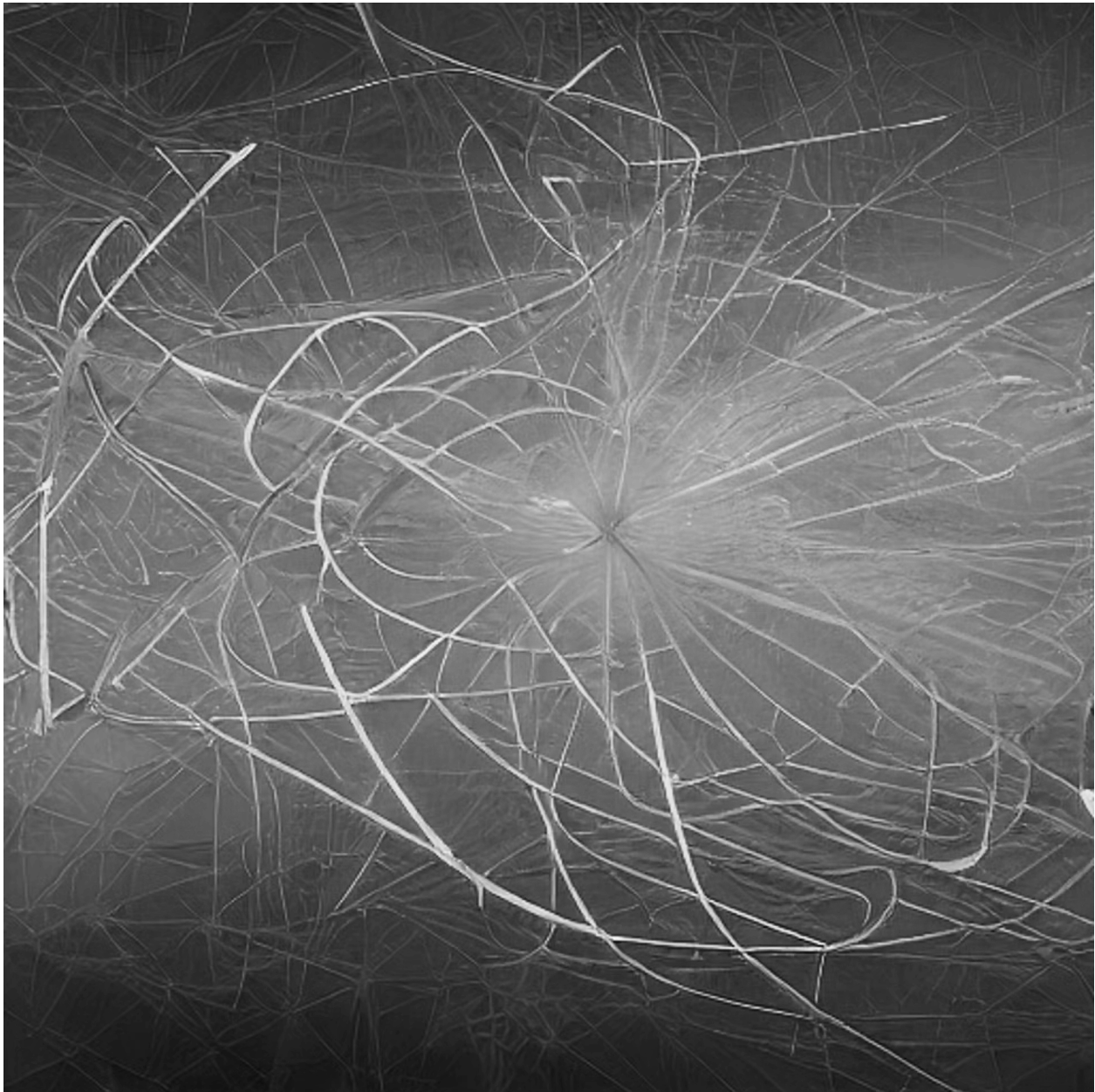
But folks would quickly come to see that the differences in protocols were actually a **strength** as a fundamental replacement, not a weakness. Like the problems mentioned above, the reasons why can again fall into three distinct but interrelated benefits that could help solve the broad categories of problems mentioned above (and some others).

What *can't* it do? (aka Functionality)

In 1997, HTTP/1.1 was released, and by 1999 it had been updated. By this point, there were many features that would help with the problems mentioned earlier, as long as the solution could take advantage of them. Reusable connections for client requests and “pipelining” (allowing multiple

requests to be made before any of them have completed) meant HTTP was getting more efficient. HTTP caching had grown considerably, including beefier cache control mechanisms, which meant both clients and servers could be much smarter about when to send data and when it didn't need to, and did it in such a way that folks didn't need to resolve this problem at a "higher level". This was in part built on top of the evolving set of formalized and well-defined request and response headers, another abstraction, which also included the idea of "content types" to set clear contracts and expectations on "the kind of data" that was being requested and/or provided.

It also included formalization of things like the `Authorization` header, which, along with the formalization of HTTPS in 2000 and the evolution of its underlying cryptographic protocol (officially reaching TLS 1.2 just months before MSS was released), provided at least some peace of mind for the most egregious of security concerns and content theft.



Some already-present features wouldn't be taken advantage of until much later (range requests to make it easy to store a single media file and just ask for "the relevant bits"). Others would crop up that would help (HTTP/2 PUSH for eager fetching in "Low Latency" extensions to HAS protocols). Still others would simply automatically and transparently benefit HAS, so long as the client and the server supported them (HTTP/3 QUIC to reduce packet size and handshake time as well as further connection pool optimizations).

Been there, done that (aka Maturity)

Most of the functionality described above was the result of a decade of addressing and improving on **generic** issues that were no longer just about HTML and web pages. Moreover, there was no reason to think these improvements or this focus would stop anytime soon.

But it wasn't just the protocol itself that had matured. Standards and architectures had been evolving around HTTP. Roy Fielding finished his doctoral dissertation laying the groundwork for REST in 2000, and it was fast becoming **the** way clients talked to servers for all sorts of use cases, including for binary data uploads and downloads (making more specialized protocols like FTP and Gopher less and less common). One of the primary reasons for this was, even if most servers were still stateful, REST (over HTTP) didn't have to be, which meant it could much more easily scale both vertically and horizontally and could do with far less resource overhead than stateful, session-based connections.

The infrastructure to take advantage of these protocol details had also matured. Load balancing solutions were at this point especially well-trodden for HTTP, working particularly well with (mostly or entirely) stateless RESTful or REST-like server architectures, which was a much clearer, simpler, and cheaper solution to the sorts of scaling complexities mentioned above.

But it wasn't just "on-prem" server solutions that had grown. Akamai's first commercial CDN offering was released in 1999, almost a decade earlier, and it and other CDNs used that time to build smarter and more efficient distribution architectures (using HTTP) for global reach, regardless of where the content was originally coming from. And while there were additional optimizations that would eventually evolve specific to media, the fundamentals were again **generic** solutions to **generic** problems.

Everywhere you look... (aka Ubiquity)

Both because "the web" was growing fast and because HTTP had already evolved (and would continue to evolve) into something much broader than

simply “GET-ing web pages and their related content,” it was also one of the most likely protocols to exist on most devices and across clients and servers. For better or (and?) worse, HTTP was showing clear signs of dominating much of the internet, much like JavaScript is currently showing signs of dominating much of the development space (also for better and worse).

The number of HTTP servers available was massive compared to “traditional” streaming media servers, existed in multiple languages, and included many free options. This also meant they could be “built on top of,” updated, and improved on for whatever use case. This simply wasn’t available for RTMP servers. Tooling, tutorials, testing frameworks, and many other “supporting actors” were widely available. This also meant that engineers who had a solid and deep grasp of HTTP could be brought up to speed and apply that base knowledge to HAS, unlike protocols like RTMP, which were much more domain specific at a “deep” level.

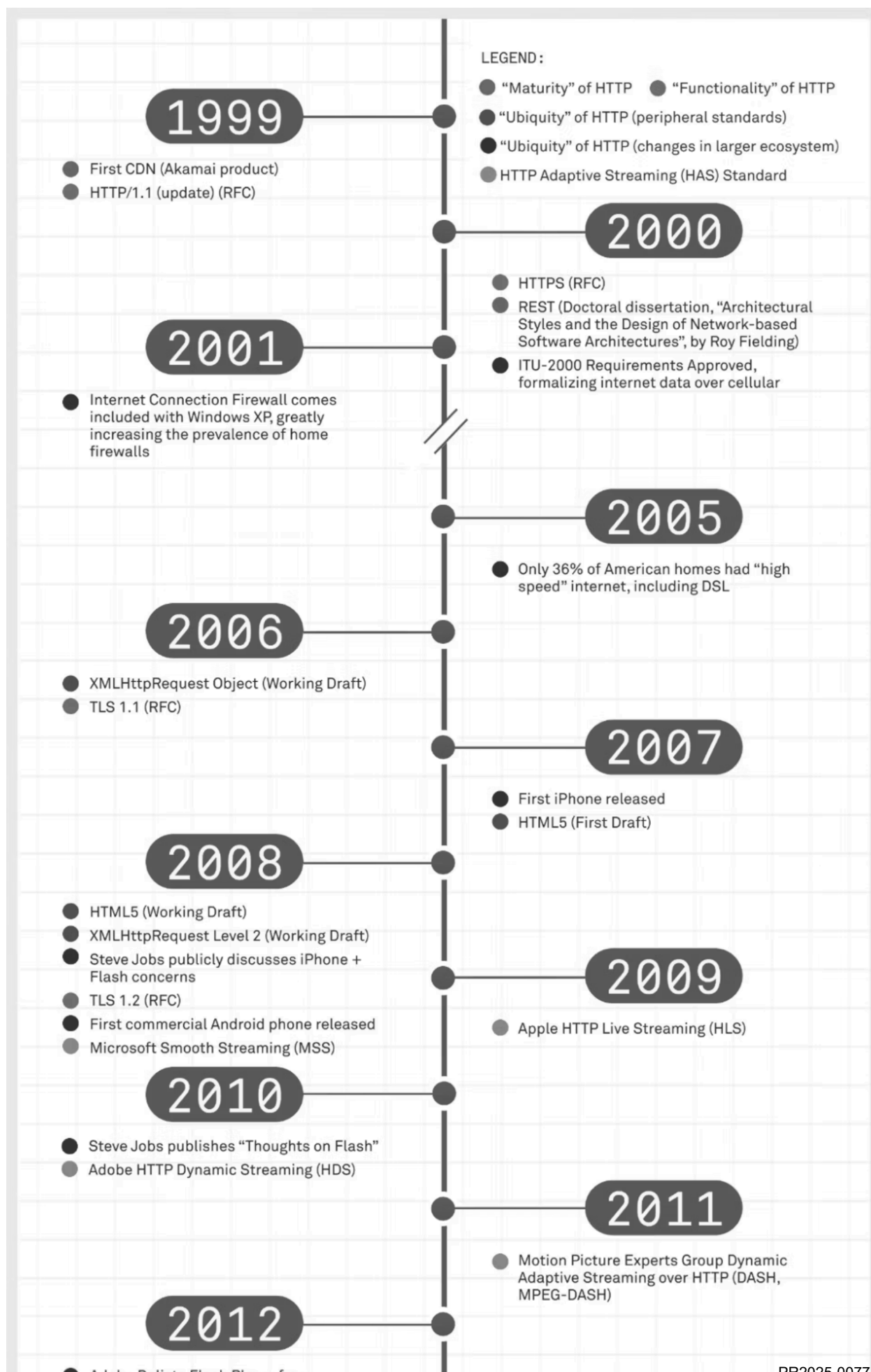
And this same sort of ubiquity was happening on “the client side.” The most common refrain to the “WAP”-enabled phones above was, “this should just work with regular HTTP and HTML,” and that’s what ended up happening with the advent of smartphones like the iPhone and Android. For operating systems, network-centric APIs, and the development environments that corresponded to them, if there was going to be any effort to make “application layer” protocol support easy and convenient, HTTP was going to be on the short list, if not at the top of the list.

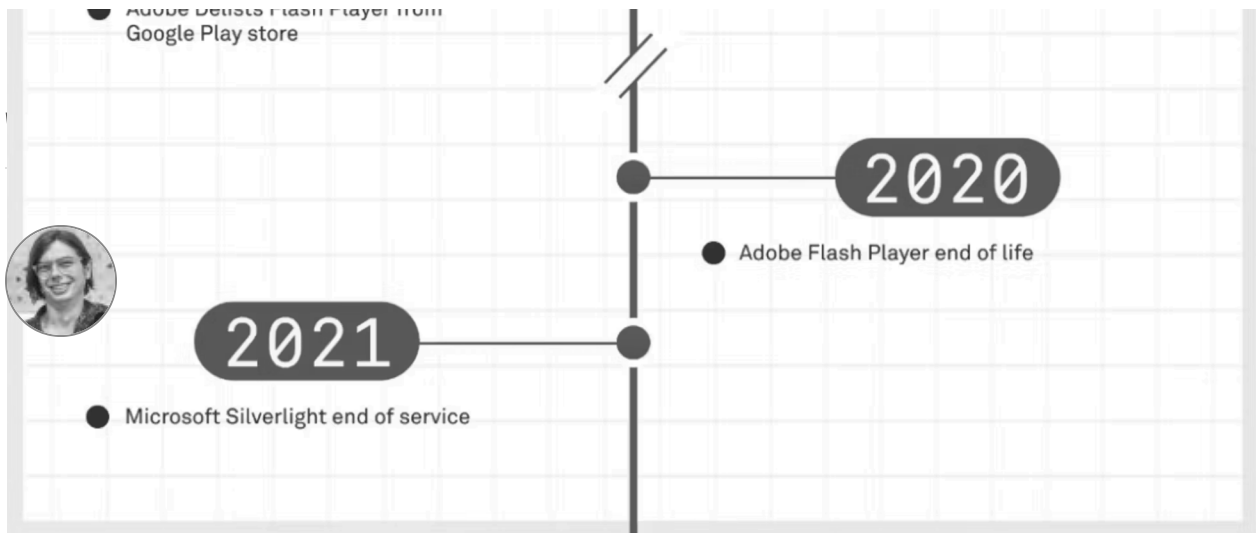
All good things... (aka Conclusion)

Any one of these on their own would help with the various concerns of cost, scale, and reach, but together they re-enforced one another, they practically guaranteed that the benefits of functionality, maturity, and ubiquity would continue, and did so in a way almost entirely independent of its use for streaming media. Since I’m writing this over a decade after even the latest major HAS standard, there’s undoubtedly a bit of hindsight bias here, since we know all of these things ended up being true. But

that just means it was, in fact, a good decision, even if only confirmed in hindsight.

Or rather, it was a good decision so long as folks could figure out a way to take advantage of these things. That's where the other shared details of HAS come in, but that's also for another blog post.





players, I'm spending my time doing deep dives in philosophy, stomping around the Chicago arts scene, and watching TV on the internet with my cat Grits.

Leave your wallet where it is

No credit card required to get started.

SIGN UP

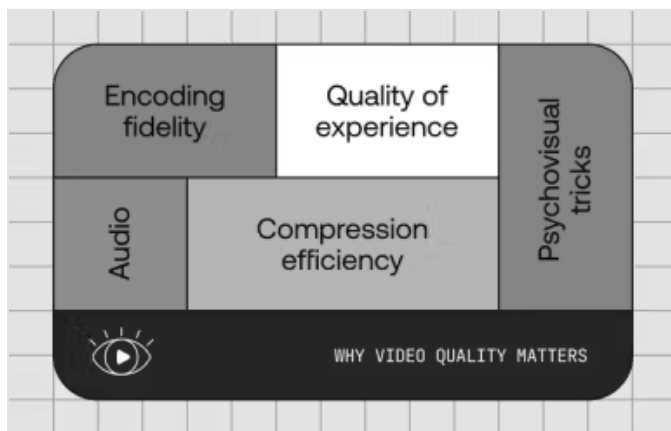
Read more like this





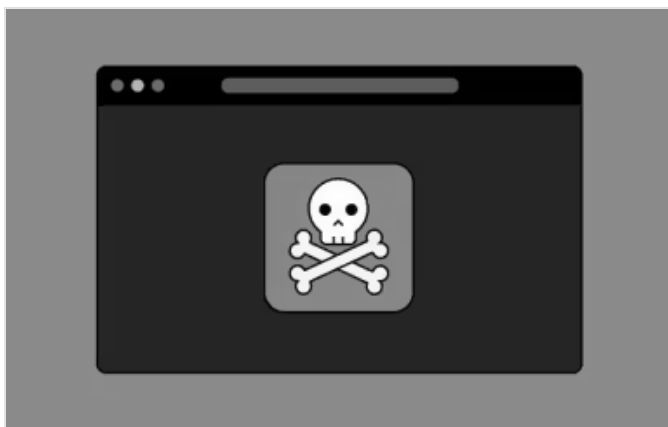
APRIL 2, 2025 • BY DAVE KISS

How to build a Windows 98-style video player in 2025



JULY 26, 2024 • BY JON DAHL

Why video quality matters



JUNE 27, 2024 • BY JUSTIN SANFORD

Being a provider in a pirate's world, or: how to not get your service blocked by an entire country

See all video education posts →

Check out our newsletter

A monthly-ish digest of all the best new blog posts and features

FIRST NAME

EMAIL

SIGN ME UP

Platform

MUX VIDEO

Video API

Features

On-Demand

Live

Interactive

Encoding

Pricing

Pricing Calculator

MUX DATA

Overview

Features

Monitoring

Pricing

MUX PLAYER

Overview

Developers

[Documentation](#)

[API reference](#)

[Webhook reference](#)

[Open source software](#)

[Beta features](#)

[Contact support](#)

Resources

[Video glossary](#)

[Learn about video](#)

[Demuxed](#)

[HowVideo.Works](#)

[Mux vs. Cloudflare Stream](#)

Solutions

[Video for your video platform](#)

[Video for UGC](#)

[Video for connected fitness](#)

[Video for AI](#)

[Video for Vercel users](#)

[Data for broadcasters](#)

[Data for telco operators](#)

[Data for sports](#)

Integrations

Node

Ruby

PHP

Python

Elixir

Go

Astro

SvelteKit

Flutter

WordPress

Next.js

React Native

Remix.js

Laravel

Company

[Blog](#)

[Team](#)

[Customers](#)

[Partners](#)

[Jobs](#)

[Startup Program](#)

[Powered by Mux Program](#)

[Sign up for our newsletter](#)

[Download Press Kit](#)

© Mux, Inc. 2025

[Terms of service](#)

[Security](#)

[Privacy policy](#)

[Sitemap](#)

STATUS: GOOD