Anton Venema
Chief Technology Officer

Anton is a leading expert on real-time communications solutions, and the visionary lead architect behind LiveSwitch Cloud and LiveSwitch Server.

Live Broadcasting

Delivering ultra-low-latency media at massive scale with LiveSwitch and WebRTC

Introduction

In the early days of the internet and personal computing, it wasn't uncommon to wait for a video to download completely before being able to open and play it. Internet speeds, compression standards, media quality, and delivery protocols have evolved significantly since then, often competing with each other as technologists work to deliver content faster, cheaper, and more reliably.

The demand for on-demand content has skyrocketed since then. High-traffic sites like YouTube, Vimeo, and Netflix stream billions of pre-recorded videos every single day. The technology to support this has evolved alongside, with HTTP-based streaming technologies in the forefront of the current delivery mechanisms.

More recently, live streaming over the internet has exploded in popularity. A generation of "cord-cutters" walking away from expensive television subscription services combined with an exponential growth in internet speeds and new opportunities for interactivity within a live broadcast has driven many live content producers to distribute online. Social media heavyweights like Facebook and Twitter are pushing hard into this space, encouraging their users to generate new content and hosting major events like the 2020 presidential debates.

As the drive towards interactivity increases, so does the demand to reduce latency on the generation and distribution of live content. Traditional techniques using HTTP streaming, which have been adapted for live broadcast, generally involve latency that exceeds what is reasonable to drive a great user experience in a live interactive broadcast.

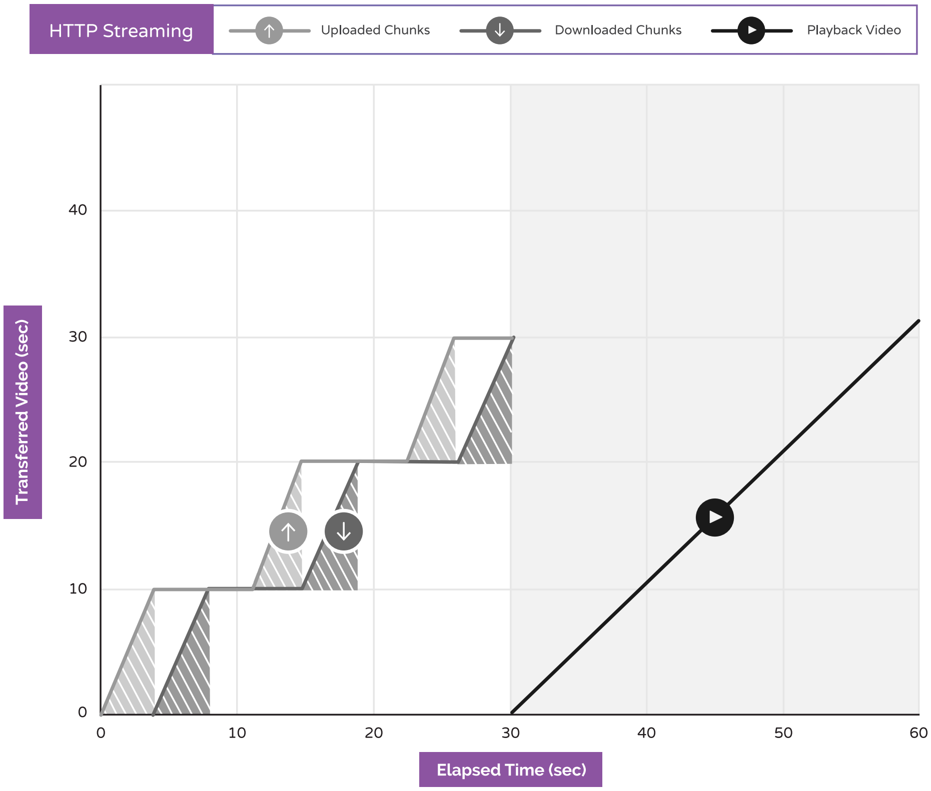
A better approach is to use WebRTC-based streaming with efficient server scalability to drive latency to sub-second values. With plugin-free support now from every major browser vendor on desktop and mobile combined with an intelligently designed media server farm, it's possible to scale to millions of concurrent users while maintaining just a few milliseconds of latency.

Before going into the details of RTC streaming, it's useful to understand a bit about HTTP streaming and how it works.

HTTP Streaming

HTTP streaming, used by Wowza, Red5Pro, DaCast, IIS Media Services, and others, operates by splitting up a media recording into lots of "small" (measured in seconds) chunks and transcoding each chunk into a range of bitrates. Clients can then download these individual chunks over HTTP and select a different bitrate for each chunk depending on what is available and how well the local network is keeping up.

The primary problem with HTTP streaming for live broadcasts is that it introduces several seconds of latency, often as much as 30 seconds, as chunks are buffered by the playback device. Proprietary protocols like Apple's HTTP Live Streaming (HLS), Microsoft's Smooth Streaming, and Adobe's HTTP Dynamic Streaming (HDS), as well as international standards like Dynamic Adaptive Streaming over HTTP (DASH) all use this same basic technique, and all suffer from the same inherent problem.



While this may be acceptable for pre-recorded or on-demand content, it is highly disruptive for live content. The total time before playback can begin is a combination of upstream latency, server processing time, downstream latency, and this chunk buffering time.

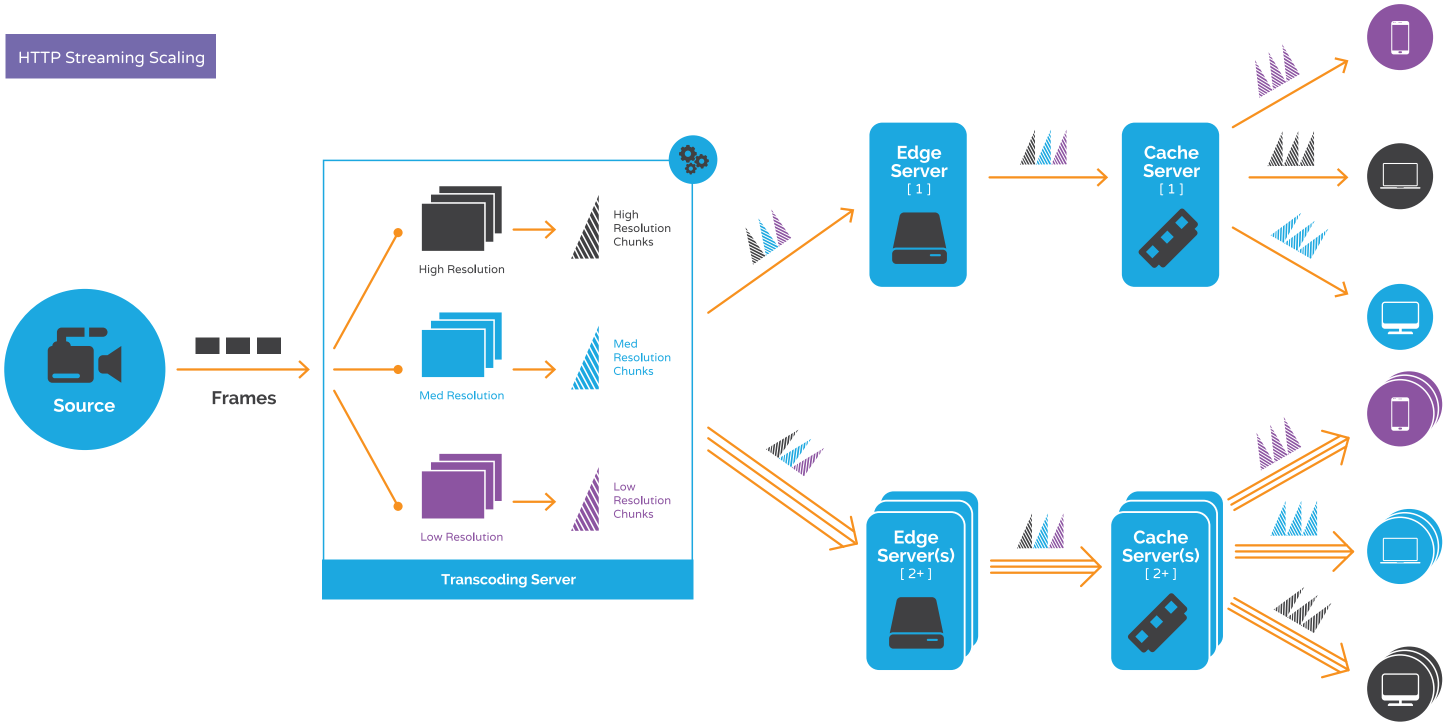
Scalability

Scaling an HTTP stream requires a network of content distribution servers to which encoded chunks can be delivered from a media server responsible for processing the inbound stream. More specifically:

1. Content is recorded, encoded, and uploaded to a media server over an RTP-based connection.
2. The media server decodes the inbound stream and re-encodes it at varying bit-rates in "small" file chunks.
3. These chunks are distributed out to content distribution servers [1 thru "n"]. The number and geographic location of these servers can vary, but ideally should be located as closely as possible to the content subscriber to minimize latency.
4. Clients start downloading these chunks and buffering them for playback. This is where the most latency is introduced, anywhere from 10-30 seconds.
5. Client-side code analyzes the network traffic determine which bitrate to use for the next downloaded chunk.

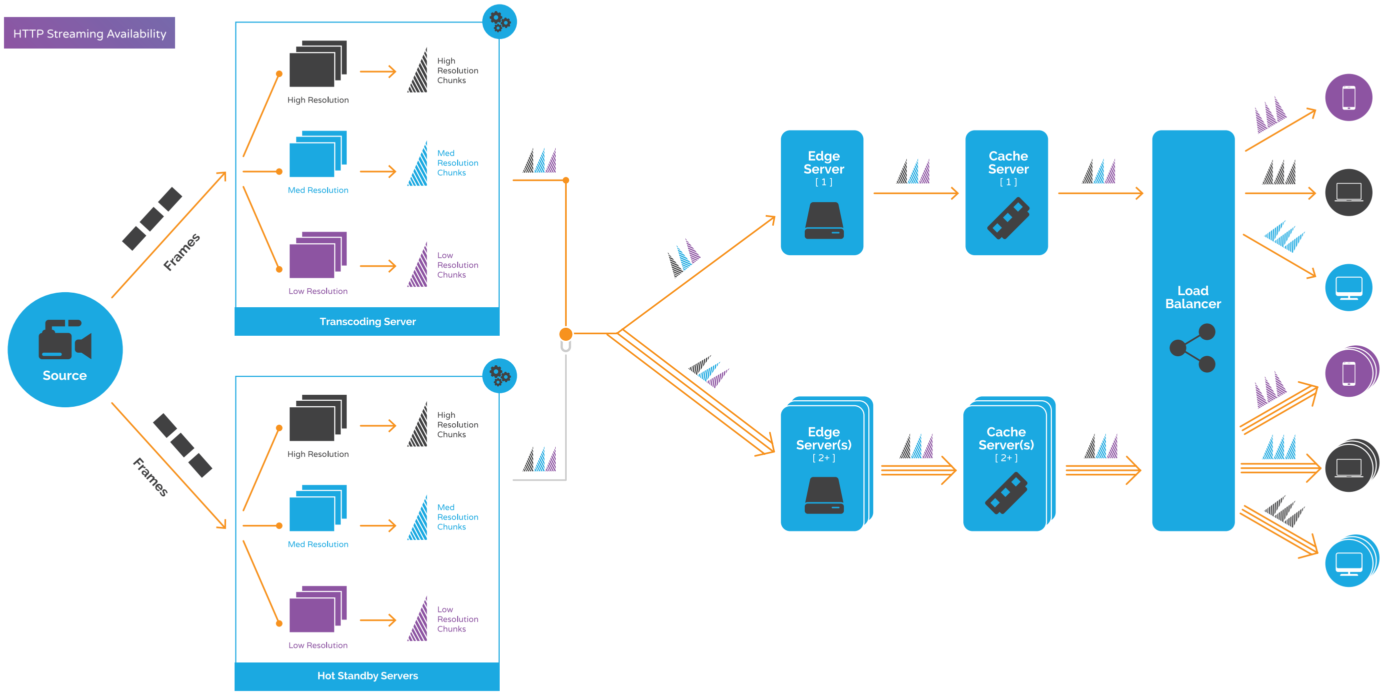
To improve performance, the content distribution servers can and should use in-memory caching to avoid unnecessary repeated disk I/O. An extra file forwarding tier between the media server and content distribution servers can be added to increase backend scalability.

HTTP Streaming Scaling



Availability

Making an HTTP stream highly available means uploading the content to at least two different media servers and ensuring that each content distribution node runs with at least two servers with the same content available on each. Since everything is HTTP-based, load balancing on the download side is relatively straight-forward - just add an HTTP load balancer. A highly-available load balancer can ensure that if a content distribution server goes down, requests are simply routed to another server.



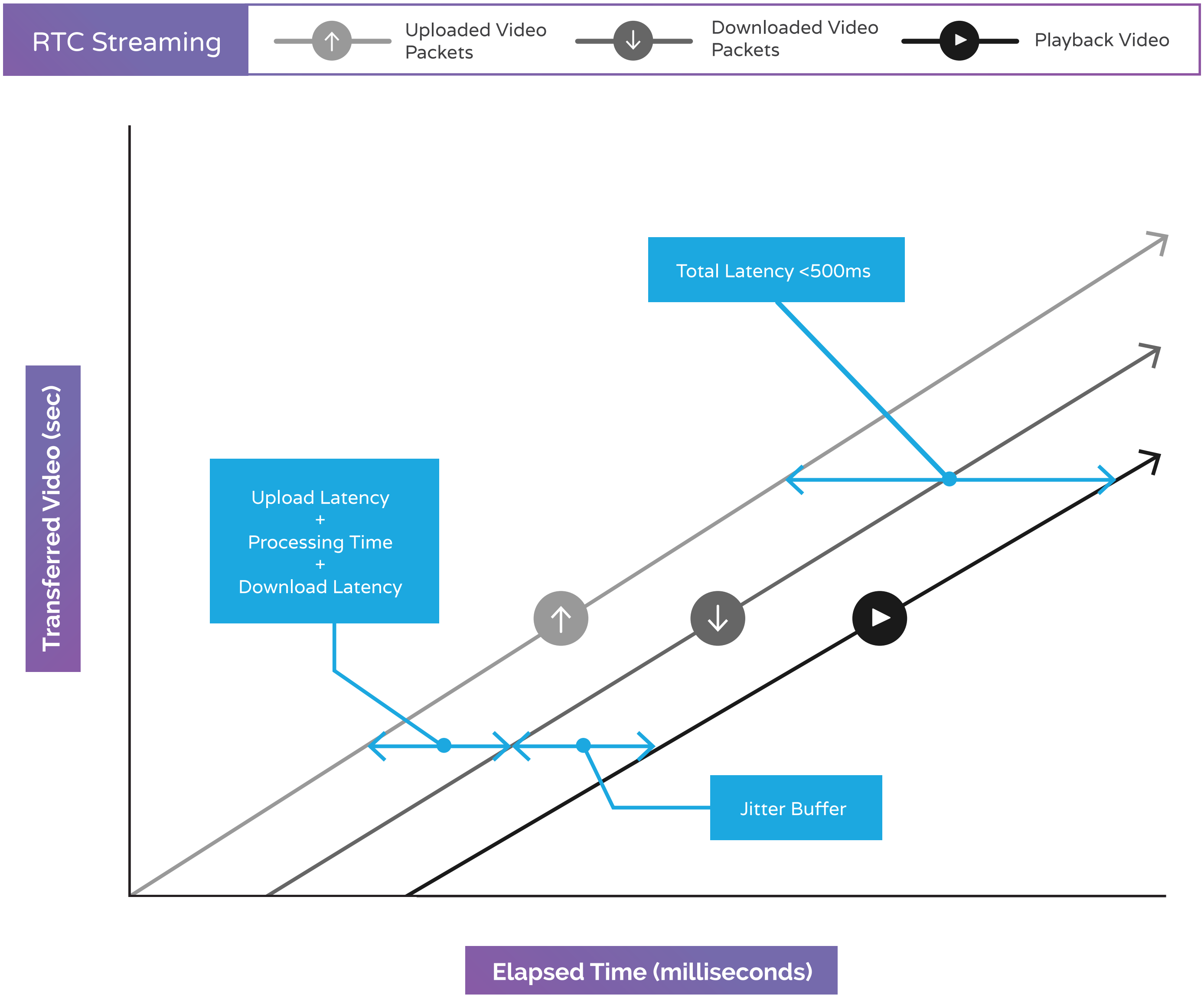
Challenges

As mentioned earlier, the primary challenge with HTTP streaming for live broadcast is the added latency. Apple's HLS, for example, uses a default chunk size of 10 seconds. Assuming 3 chunks need to be downloaded before playback starts, this translates to a minimum 30-second delay on the feed. For live applications, especially interactive or time-sensitive ones, this extra latency can severely degrade the user experience.

Reducing the chunk size is possible (as low as 1 second), but buffering at least a few chunks is still required, generally at least 10 seconds total, and the cost advantages of HLS start to go away as you do this. To support switching bitrates, each chunk has to be playable independently of any other chunk. This means that each chunk has to start with a keyframe (a complete image), so reducing chunk size and latency has the side effect of significantly increasing client bandwidth requirements, and increasing the number of server requests. Since all bitrates share the same chunk size, this most negatively impacts the higher-quality streams by requiring client bandwidth to vastly exceed what may be necessary.

RTC Streaming

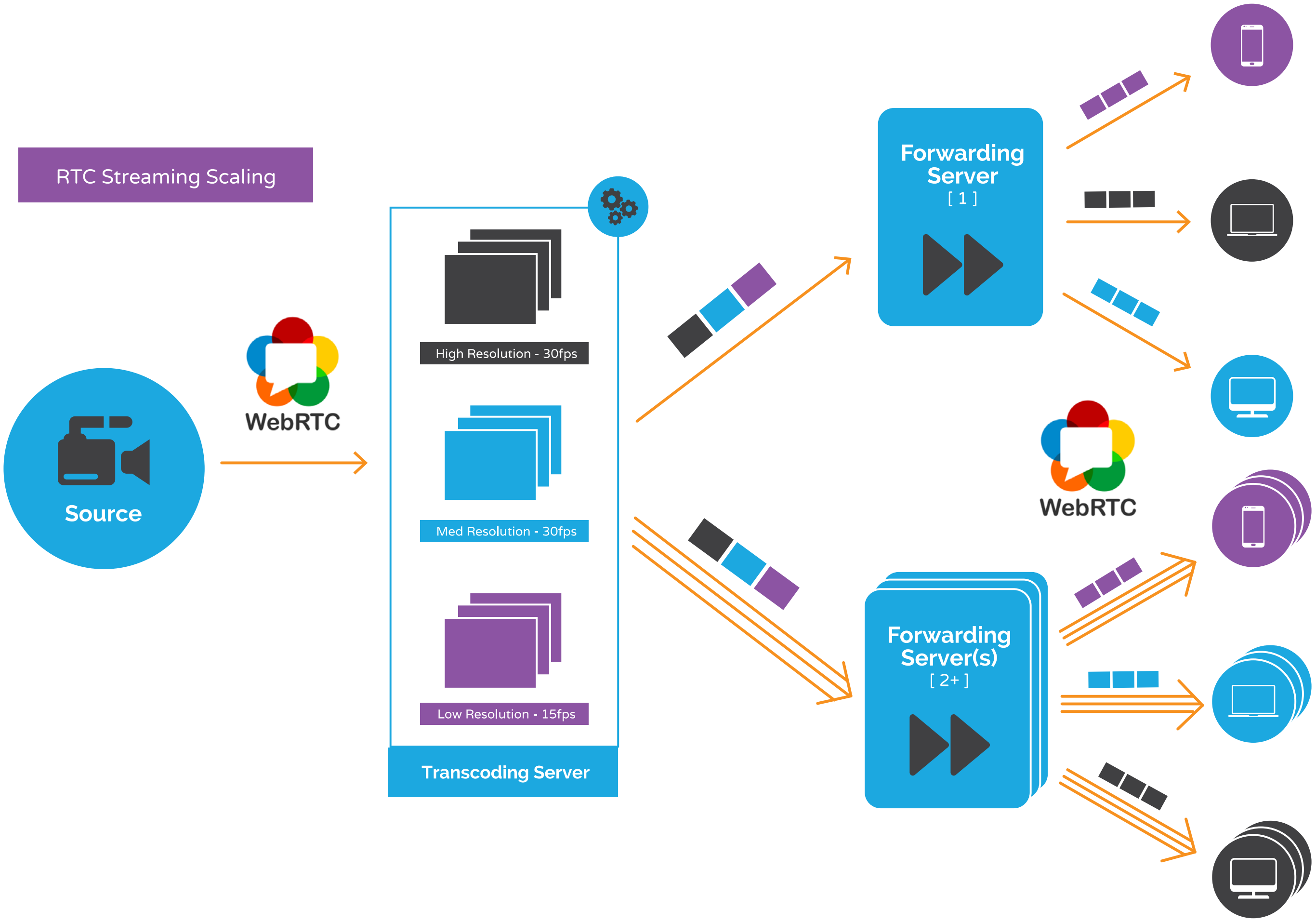
For live content broadcasts with low latency, there's nothing better for the user experience than WebRTC streaming, the primary delivery mechanism in LiveSwitch. While the client still has to buffer a little bit to account for network jitter (varying delay), the buffering time is measured in milliseconds, not seconds. The total time before playback can begin is upload latency plus server processing time plus download latency plus this jitter buffer duration. Even with less-than-stellar internet connectivity, total latency is generally sub-second, and often less than 500ms or even 300ms on stable high-speed connections.



Scalability

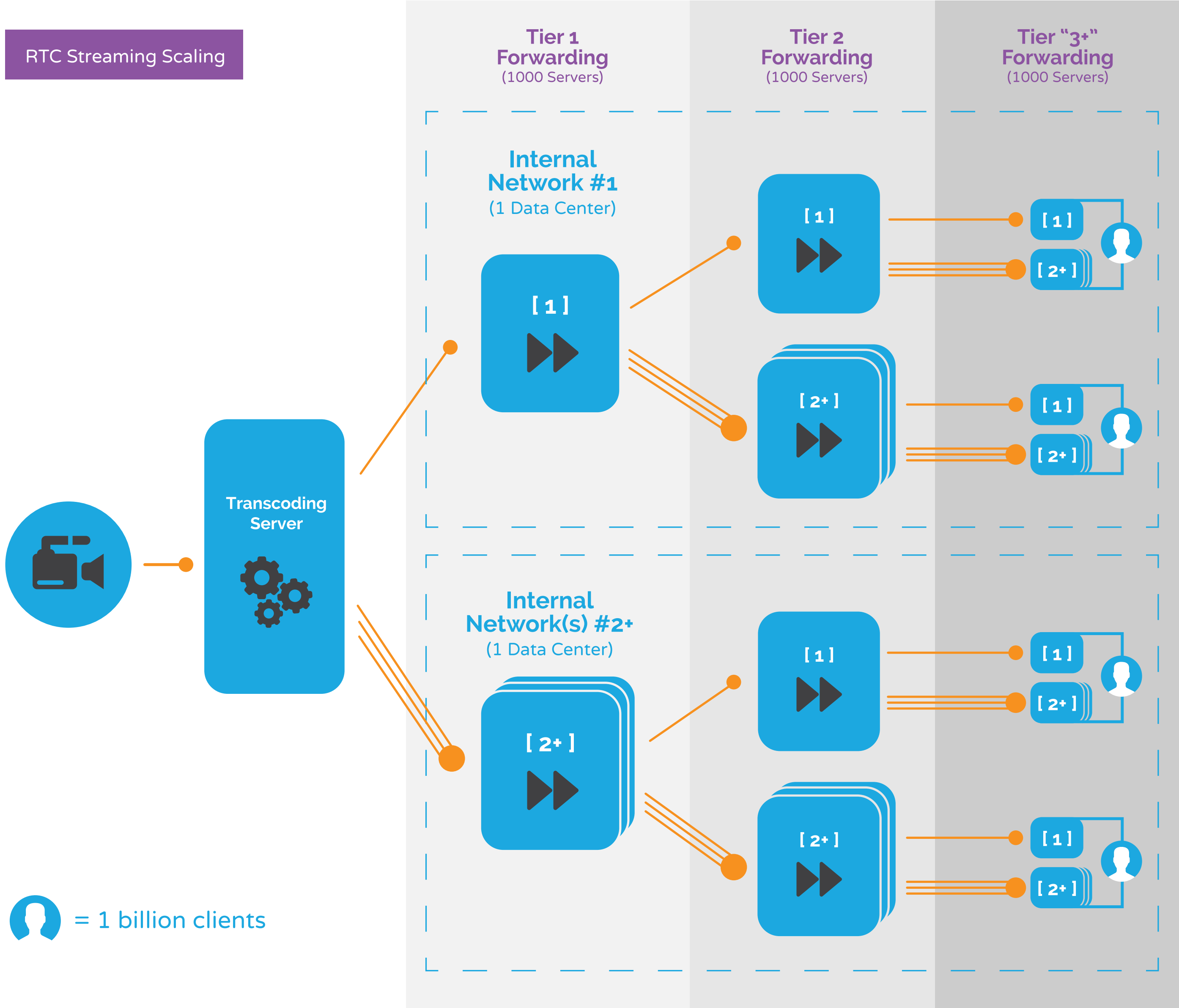
Scaling a WebRTC stream requires a network of media servers capable of forwarding individual media packets from a primary media server that receives and processes the inbound stream. More specifically:

1. Content is recorded, encoded, and uploaded to a media server over a WebRTC-based connection.
2. The media server decodes the inbound stream and re-encodes it at varying profiles in-memory with varying bit-rates and keyframe intervals.
3. Each encoded frame is forwarded to an array of forwarding media servers [1 thru "n"] . The number and geographic location of these servers can vary, but ideally should be located as closely as possible to the content subscriber to minimize latency.
4. Clients open a WebRTC connection to the nearest forwarding media server and receive each frame as it arrives. This is where RTC streaming blows away the latency of HTTP streaming. Instead of spending 10-30 seconds at this step, we're only limited by client network latency - about 100 milliseconds.
5. Client and server-side code analyzes the network traffic to determine whether congestion can be alleviated using forward error correction (FEC), temporal/spatial scalability (SVC), or packet retransmission (RTX), or whether a lower/higher-profile stream should be consumed.



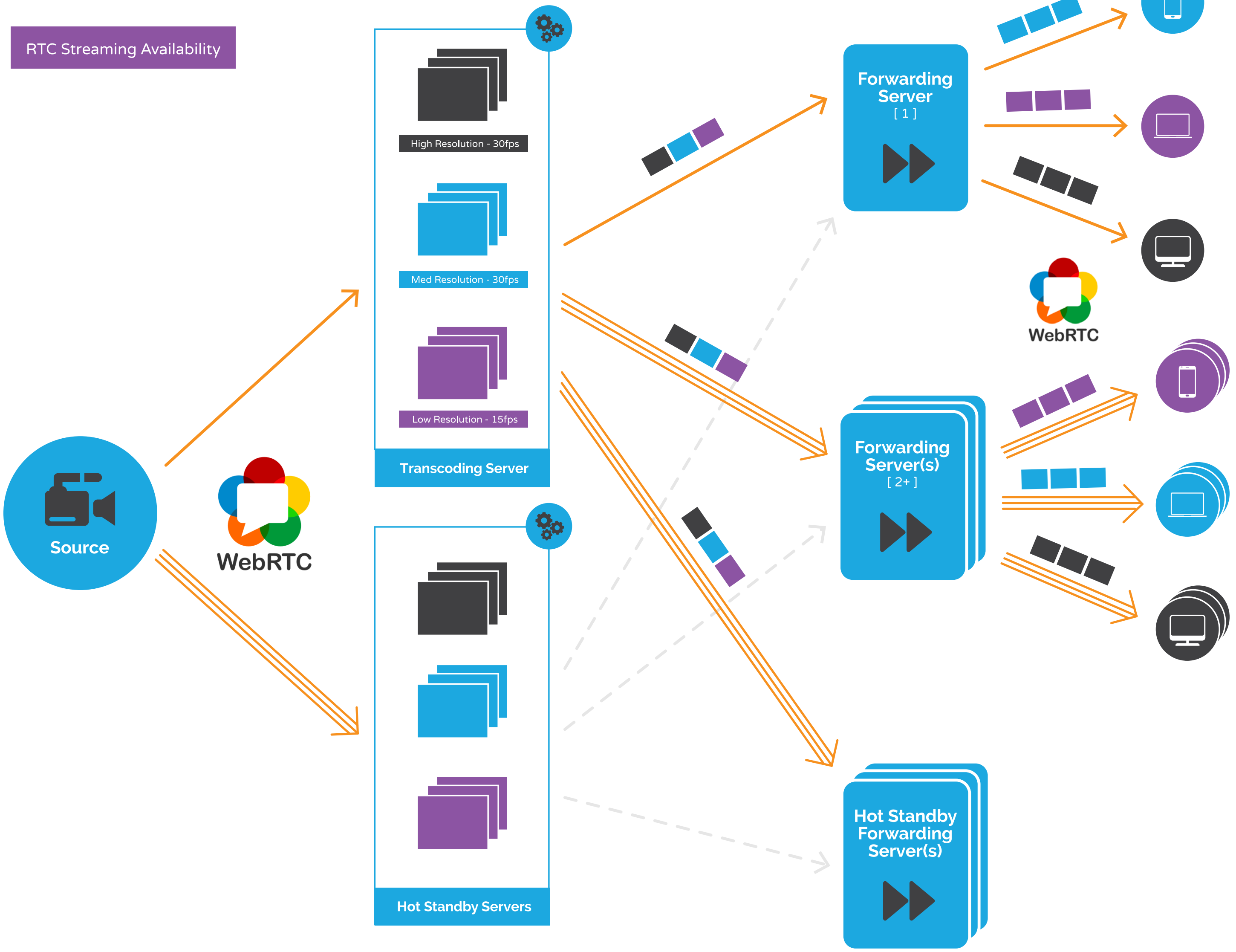
Additional tiers of forwarding media servers can be deployed to this tree structure to increase backend scalability.

If we know how many streams a given server can handle, we can infer how many servers (or tiers of servers) are needed to support a given client load. For example, assuming each server can deliver a video feed to 1,000 concurrent clients, then a large-scale broadcast setup with 1 upstream server and 1,000 downstream servers would be able to broadcast to 1,000,000 concurrent clients. We can achieve massive scale (with an associated cost) by using an array of servers whose sole task is to broadcast data within the internal server network.


RTC Streaming Scaling
Tier 1 Forwarding (1000 Servers)
Tier 2 Forwarding (1000 Servers)
Tier "3+" Forwarding (1000 Servers)
Internal Network #1 (1 Data Center)
Internal Network(s) #2+ (1 Data Center)
Transcoding Server
[1]
[2+]
[1]
[2+]
[1]
[2+]
[1]
[2+]
[1]
[2+]
[1]
[2+]
[1]
[2+]
[1]
[2+]
= 1 billion clients

Availability

Making a WebRTC stream highly available requires at least one additional "upload" media server to provide redundancy for the inbound stream. The uploader is responsible for sending the content to both servers, or at minimum reacting quickly to network problems by failing over to the other server. Internally, additional forwarding media servers are required for redundancy at each tier except the last one, where connection failures are addressed with simple reconnection/rehydration logic in the client application.



Challenges

The primary challenge for RTC streaming is its technical complexity. Persistent connections don’t load balance as easily as stateless HTTP requests, and so failover has to be built into the software itself. This final tier of servers is responsible for negotiating capabilities with the clients (like forward error correction), and determining how to get the highest-possible-quality content to the client as reliably as possible, which means applying codec-specific scalability patterns, estimating FEC effectiveness, and eventually switching the client to a new stream. Media server software has to be incredibly careful about protecting the upstream servers from data generated by the downstream servers. Even a tiny bit of information becomes a stampeding herd in highly concurrent use cases. Failover and migration of sessions from one server to another is especially important for use cases that scale quickly, predicting the load and adjusting the internal server communication infrastructure to optimize traffic.

Wrap-Up

Live broadcasts are always better with WebRTC. The user experience is better, new doors are opened for participant interactivity, and it can be very cost-effective. While some people are still attempting to squeeze small bits of performance out of HTTP streaming technologies, it makes a whole lot more sense to use a proven technology with real-time in its name.

If you would like to learn more about how Frozen Mountain‘s LiveSwitch products use WebRTC-based broadcasting to help you reduce latency in your live broadcasts, don’t hesitate to contact us at sales@frozenmountain.com.