LiveKit

55 Followers About

An Introduction to WebRTC Simulcast

Everything you wanted to know about simulcast, but were afraid to ask

David Zhao Aug 26 · 8 min read

Simulcast is one of the coolest features of WebRTC, allowing WebRTC conferences to scale despite participants with unpredictable network connectivity. In this post, we'll dive into simulcast and explore how it works with SFUs like LiveKit, to make WebRTC work for larger conferences.

Challenge of the "slow" user

Selective Forwarding Units (SFU) have become the predominate architecture for WebRTC-based conferencing. Their role is to forward data from one user to others in the room; thus, significantly reducing the amount of data each user has to send. SFUs have gained quite a bit of popularity in recent years because they are relatively simple to scale. Since they do not need to encode/decode media, forwarding data typically requires little CPU overhead.

While it's straightforward to simply forward a stream as it arrives, real world network conditions pose some challenges. Specifically, not everyone has an internet connection fast enough to receive the streams that others have published. From the moment when a "slow" user joins a conference, this issue becomes very visible.

User C would publish and receive choppy streams.

Since the slow user has insufficient bandwidth to fetch high quality streams others are sending, they'll have high packet loss on the receiving side, causing choppy picture quality or black/blank frames. As downstream bandwidth becomes saturated, congestion also impacts the user's ability to upload their own video.

To make matters worse, the slow user's WebRTC client will continually send PLI (picture loss indication) messages to other publishing participants (via SFU), since it's not able to receive sufficient packets to render frames. When other WebRTC clients see a PLI packet, they respond by generating new keyframes, which need to be sent to everyone, increasing bandwidth requirements for the entire session. The increased bandwidth requirements can trigger a cascading effect when they exceed the bandwidth limits of other participants, too.

In a conference with large number of people, the likelihood of someone in the room having a weak connection increases. Every conference eventually experiences this slow user problem. So, to ensure smooth and high quality delivery, we have three options:

- Use a scalable codec such as VP9 or AV1
- Lower the bitrate of everyone's streams so it doesn't overwhelm the slow user (i.e. the lowest common denominator)
- Send separate streams to participants, catering to each user's available bandwidth

Scalable video codecs

It's worth mentioning some of the newer video codecs and their trade-offs. Both VP9 and AV1 have built-in scalability: they encode in a way such that a single stream can adapt to multiple target bitrates. However, they do come with some drawbacks (as of Aug 2021):

- VP9 is not supported in Safari, and is more CPU-intensive
- AV1 is only supported in Chrome

While the future looks bright, today, neither codec is a practical solution for dynamic bitrate.

Enter simulcast

Simulcast allows WebRTC clients to publish multiple versions of the same source track, with different encodings (i.e. spatial layers). In LiveKit, participants publish high, medium, and low-res versions of the same video, encoded at different bitrates.

Simulcast is designed to work with an SFU, where the SFU receives all three layers for the track and makes a decision on which layer to forward to each subscriber.

SFU chooses to send User C low-res streams

From the subscriber perspective, simulcast is invisible. WebRTC tracks are designed to support adjusting resolution, dynamically. For a subscriber, this enables the SFU to switch to a different layer as their bandwidth changes while maintaining a continuous video stream, sans interruption.

Does this mean a publisher sends more data? Yes. However, the lower resolution layers consume far less bandwidth than the high-quality version. For example:

- original/high: 1280 × 720 @ 2.5mbps
- medium: 640 × 360 @ 400kbps
- low: 320 × 180 @ 125kbps

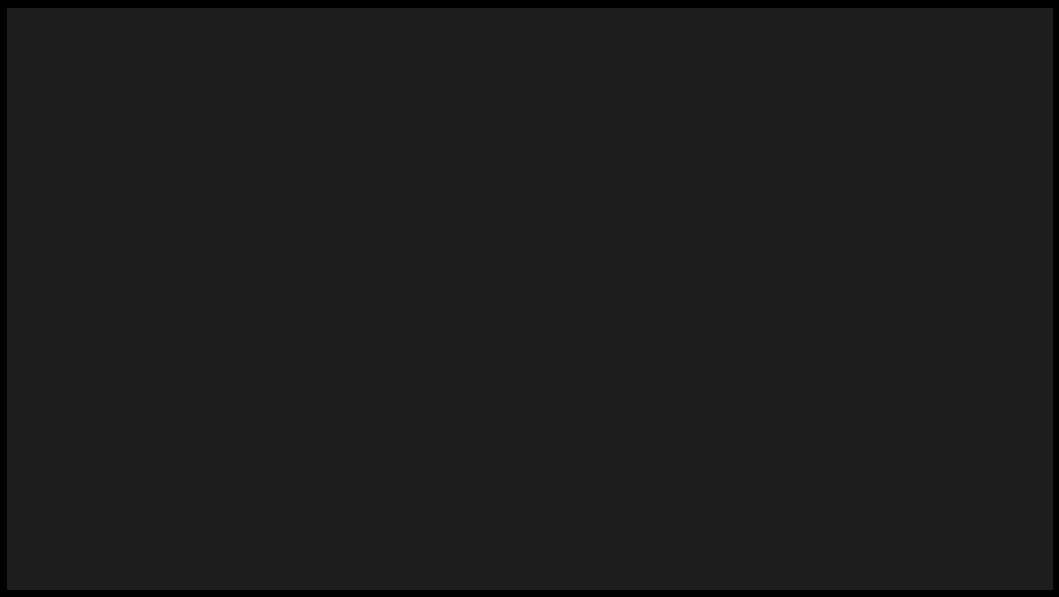
With simulcast, a publisher uses just 17% more bandwidth to publish all three layers.

With more advanced feedback mechanisms, it's also possible to use significantly less bandwidth. If one's video is strictly rendered as a thumbnail on the receiving end, why send a 720p layer? Each subscriber could inform the SFU of the layers it actually needs.

Take a look at Zoom's network stats and you'll see they manage this incredibly well. When a participant is not the active speaker and displayed as a thumbnail to others, Zoom's client only sends a 320×180 layer.

Spatial vs. temporal scalability

We covered spatial layers above, but what about temporal scalability? In essence, temporal scalability is the ability to lower a stream's bitrate by dynamically reducing the track's frame rate.



Frames in a non-scalable video stream, each frame references the previous

While the idea is simple, the complexity comes from how videos are encoded — streams contain mostly delta frames which depend on previous frames. If the decoder needs to apply a delta to a frame that was dropped, it can't render subsequent frames. For temporal scalability to work, the encoder needs to plan ahead for frame dependencies.

Frames in a scalable stream with two temporal layers

VP8 supports temporal scalability, and browsers enable it automatically when simulcast is used. In the above diagram, we see a temporal-enabled stream with two temporal layers. We use TID to denote the temporal layers: a base layer with TID 0 and an upper layer with TID 1. Frames on the base layer only reference other base layer frames. For a subscriber with limited bandwidth, we can skip sending all frames with TID 1; only TID 0 frames will be forwarded. This effectively reduces bandwidth requirements by ~50% while maintaining the original stream's quality. Temporal scalability is preferred for content such as screen share, where sharpness trumps FPS.

. . .

What goes on in the SFU

To support simulcast, the SFU needs to perform a few extra steps to make the magic happen. Let's take a look.

Rewrite sequence numbers and timestamps

Every RTP packet contains a sequence number indicating its order in the stream, and timestamp indicating when the frame should be played back. WebRTC clients rely on sequence numbers to detect packet loss, and if it should re-request (NACK) the packet. When a client receives sequence numbers that have gaps, it assumes packets have been lost in transit.

With simulcasted tracks, each layer of the track is sent as a distinct track coupled with its own set of sequence numbers. When the SFU switches between layers, there's a gap in sequence numbers. This also applies to temporal layers. When a portion of frames are dropped, there are skips in the outgoing sequence numbers.

To compensate for this, the SFU has to keep track of sequences for each subscriber and rewrite the sequence number for each packet. This mapping needs to be maintained even with retransmissions of lost packets.

Bandwidth estimation / congestion detection

So how do we know which spatial and temporal layer to send each subscriber? Ideally, it would be the highest quality stream their connection could handle. But how is that determined?

Turns out, there are some neat feedback mechanisms built into WebRTC via RTCP packets:

- Receiver Reports: periodically sent by the receiver to indicate packet loss rate and jitter, among other things
- Transport-Wide Congestion Control (TWCC): allows the sender to detect congestion on the receiver end
- Receiver Estimated Maximum Bitrate (REMB): periodically sent by the receiver indicating estimated available bandwidth. (Note: TWCC is preferred over REMB today)

Keep track of subscriber layers

For each subscriber and track combination, the SFU needs to keep track of:

- Which spatial layer does the subscriber want? For example, if the client only needs a 320×180 stream, then we shouldn't send anything higher.
- Which spatial and temporal layer(s) does the subscriber's bandwidth allow?

It then makes a decision on the desired layer for each subscriber, governed by: `min(desired, allowed)`

. . .

Handling publisher constraints

Thus far, we've covered how to deal with a subscriber's bandwidth limitations. What if bandwidth constraints exist on a publisher's side? If a publisher has limited upstream bandwidth, they can't successfully publish all three spatial layers.

Web browsers handle this scenario fairly well. When bandwidth is limited and simulcast is enabled, WebRTC will disable the high-res layer until it can send the stream without significant packet loss. You can observe this via WebRTC stats, where `qualityLimitationReason` is set to `bandwidth.` Similarly, if the publisher's processor isn't fast enough to encode three separate layers, `qualityLimitationReason` is set to `cpu` .

frameWidth	960
frameHeight	540
framesPerSecond	29
framesSent	3231
[framesSent/s]	25.986691614725963
hugeFramesSent	8
totalPacketSendDelay	153.304
[totalPacketSendDelay/packetsSent_in_ms]	3.22429906542055
qualityLimitationReason	bandwidth
qualityLimitationResolutionChanges	11
encoderImplementation	SimulcastEncoderAdapter (libvpx, libvpx, libvpx)
firCount	0
pliCount	3

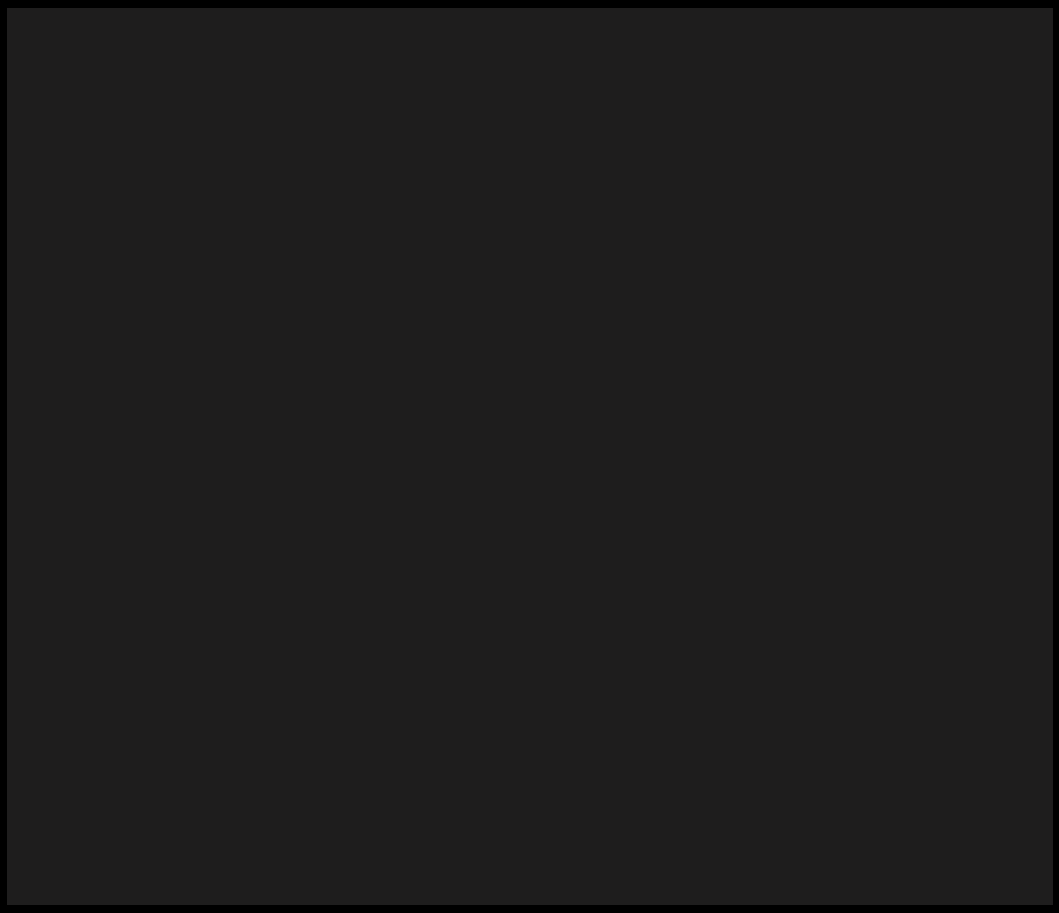
chrome://webrtc-internals

Unfortunately, advanced notice isn't given when a browser decides to stop sending a layer. Any subscribers on the terminated layer stop getting new packets, which causes viewers to see a frozen frame.

This is undesirable. It's a better experience to place a subscriber on an available, lower-res layer. As you might have guessed, this is another issue the SFU mitigates. The SFU needs to monitor incoming packets on each layer for interruptions. If detected, that layer will be marked unavailable, and subscribers moved to alternative layers. You can see how LiveKit handles this in StreamTracker.

Browser behavior

It's worth noting the specific behavior of the browser when bandwidth limitations are encountered. After disabling a layer, the browser will attempt to re-enable it just seconds later, only to disable it again upon discovering bandwidth has not improved. This vacillation is apparent in webrtc-internals graphs. You can simulate a low-bandwidth link with the Network Link Conditioner in MacOS.



framesEncoded/s and framesSent/s both exhibit see-sawing

Through testing, we've discovered that once the browser begins to seesaw, it remains in this state even after bandwidth has recovered. This is unexpected and may be due to bugs in bandwidth estimation logic, either in the SFU or Google's WebRTC implementation. More research is needed to pinpoint the root cause.

Client-side layer management

In order to ensure stream quality recovers after network conditions improve, we've introduced custom management of simulcast encodings on the client side. This leverages a couple knobs that WebRTC exposes:

- Getting a list of encodings for a particular track via RTCRtpSendParameters.encodings

- Controlling whether a layer is active by setting RTCRtpEncodingParameters.active. Browsers will stop sending that layer when set to false.

With these levers, clients can monitor current bandwidth. When limitations are detected, a client can pause the higher layer(s) entirely and after a longer backoff, attempt to re-enable and send. We've found this approach resolves the seesaw behavior, allowing for consistent quality recovery.

. . .

If you're still reading, I hope you found this post to be useful. While simulcast increases complexity of WebRTC implementations, it's necessary to provide high-quality, multi-user conferences.

If you're looking for a simulcast-capable WebRTC conferencing platform, give LiveKit a try! :)

*Thanks to Raja, Orlando, and Russ for reading a draft of this post