Review Technology

libdatachannel, livestream, OBS, RTMP, simulcast, WHIP

Chad Hart · August 22, 2023

WebRTC cracks the WHIP on OBS

Open Broadcaster Software – Studio or *OBS Studio* is an extremely popular open-source program used for streaming to broadcast platforms and for local recording. WebRTC is the open-source real time video communications stack built into every modern browser and used by billions for their regular video communications needs. Somehow these two have not formally intersected – that is until recently.

OBS version 30 has official support for WebRTC via WHIP. WebRTC HTTP Ingestion Protocol (WHIP) is a new protocol designed for real time streaming applications. If you recall, WebRTC does not include a standard signaling mechanism so you can't connect a random client to a given service the same way you can with RTMP. WHIP fixes that, for narrow use cases like broadcasting at least.

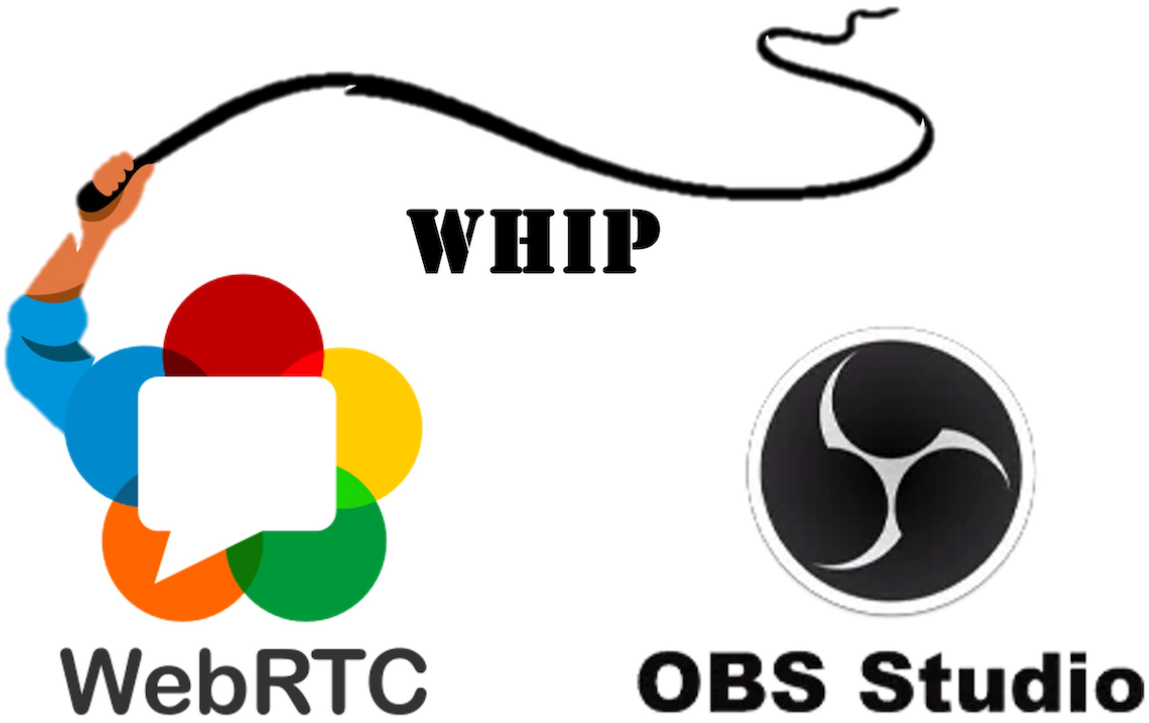
As of last week, the OBS 30 Beta package is available. Using WebRTC with OBS has been possible via forks for years, but now it is finally official.

In this post, we will:

1. Answer the question, "should I use WHIP in OBS?"

4. Examine the current OBS implementation of WebRTC
5. Look for some next-up items for obs-webrtc (AV1 & Simulcast)
6. Speculate on other features WebRTC could bring to OBS on a longer timeline

An enormous thanks to Sean DuBois for his review and suggestions in putting together this post. IETF WHIP author, Sergio Garcia-Murillio and OBS contributor Colin Edwards (ddrboxman) were also kind enough to share their insights and feedback.



Should I use WHIP in OBS?

If you are broadcasting to YouTube, Twitch, Facebook, and most of the other most popular streaming services, then the answer is NO today because these services don't support WHIP yet. However, this is likely to change over time as more providers add WHIP.

When your streaming provider supports WHIP, you should use it when you care about low-latency live

streaming. WHIP lets a broadcaster send their stream to a live audience in hundreds of milliseconds. This low latency allows the broadcaster to interact with the audience in real time. Today when a broadcaster asks a question, they typically need to wait half a minute to see any response show up in the chat. With WHIP, that response could be there in less than a second.

Will anyone support WHIP?

There are only a few services that support WHIP today. Without popular broadcast software that supports WHIP, streaming services have not had much incentive to support WHIP. OBS support for WHIP solves an important chicken-before-the-egg problem.

WHIP is a relatively easy add for existing WebRTC services to make. WebRTC services that want to support live streaming typically need to implement a RTMP gateway (or use one via an API) to handle media conversion. WHIP is just a standard signaling wrapper around WebRTC, so WebRTC services can potentially just add a WHIP signaling gateway and reuse their existing media infrastructure.

There are WHIP supporting projects, products, and services today

Today it is mostly open-source projects and communications platform providers that support WHIP. Here are some projects and products I could find that had public documentation showing WHIP support:

Commercial

- Cloudflare – streaming platform (Gustavo and Fippo did an analysis of this here)
- Dolby.io – streaming platform
- FlowCaster – desktop streaming software for editors
- Larix Broadcaster – iOS and Android app
- Nanocosmos – streaming platform
- Nimble Streamer – media server
- Osprey Video – had a WHIP enabled hardware encoder
- Red5Pro – media server

Free and/or Open Source

- Broadcast-box – another open source project now maintained by Sean DuBois

- Eyevinn WHIP – WHIP server, WHIP client, and WHEP client
- ffmpeg forks – there are 5 or more ffmpeg forks that support WHIP, but no official merge (yet)
- gstreamer – one of the most popular streaming libraries
- LiveKit – open source video platform
- OvenMediaEngine – open source streaming server
- simple-whip-server – open source companion to Janus
- SRS – realtime video server
- tinywhip – WHIP server
- TenCent – streaming platform
- Twitch – this is not advertised, but Sean added a Beta WHIP endpoint here (note this didn't connect for me when I tried today)
- webrtc-rs/whip – WHIP implementation in RUST
- whip-go – open source project lead by Gustavo Garcia
- whip-web-client – open source project from a consulting firm

There are several other smaller or less active projects I didn't mention. The barriers to adding WHIP to existing WebRTC projects is relatively low, so this list will certainly grow.

WHIP benefits

What is the benefit of using WHIP in OBS? This of course assumes your preferred streaming destinations make it available. In *obs-webrtc*'s current, minimal viable state, there are a couple of advantages:

- **Ultra-low latency** – as mentioned, you can't get lower latency with any other mechanism
- **Better audio codec** – WebRTC doesn't preclude the use of any codec but does require Opus support for audio. Opus outperforms AAC on audio quality, the audio codec most often used with RTMP in OBS today. Opus's built-in resiliency features and narrow to fullband range help it perform well in tougher environments.
- **NAT traversal** – if you are stuck in a restrictive network environment, WebRTC's Interactive Connectivity Establishment (ICE) protocol is a proven mechanism to giving the best connectivity option available (assuming the WHIP provider is setup for that). The same mechanisms also allow more seamless switching between networks, like what is more common on mobile.

Beyond that, WebRTC brings potential for other great things that could come in the future:

1. Encapsulating multiple audio and video streams in the same broadcast – WebRTC allows for sending several tracks at the same time – like sending multiple camera feeds as part of the same stream
2. Simulcast & Scalable Video Codec (SVC) for better handling viewers with mixed render size and bandwidth capabilities with the added bonus of offloading server-side encoding for non-real-time playback
3. Server-less broadcasting – WebRTC is a peer-to-peer protocol, so having a WebRTC stack in OBS opens up possibilities for smaller broadcasts to few people or hosting group chats that can be broadcast without requiring a server

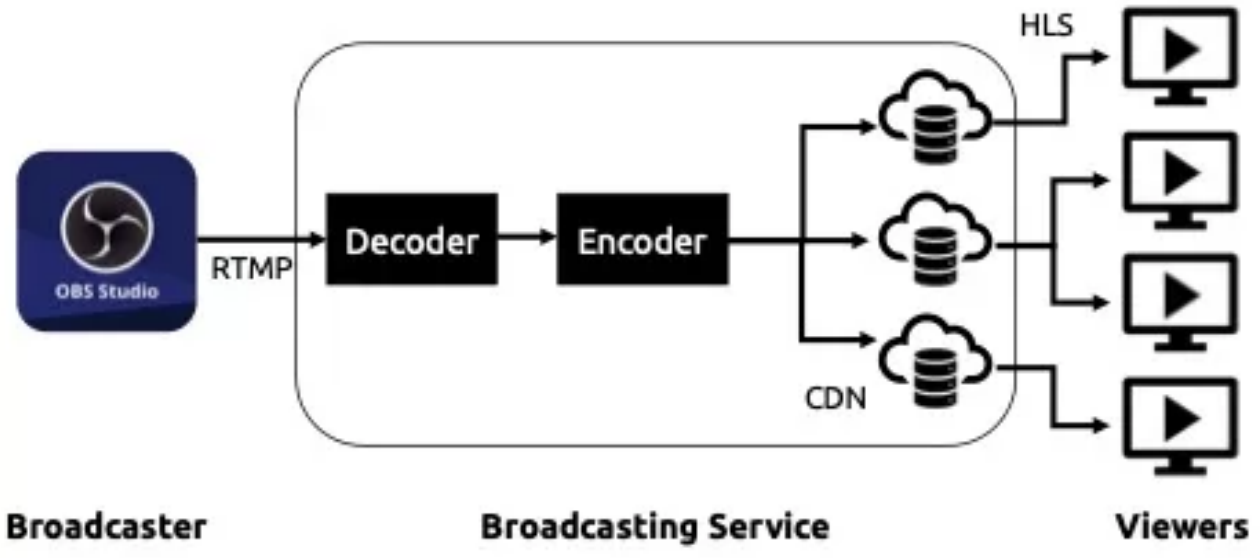
Some of these are in the works already. We will discuss these in more detail below. Let's start by comparing WHIP vs. RTMP.

What's wrong with RTMP?

Most comparisons I have seen of WebRTC vs. RTMP end up comparing the wrong things. These protocols are part of larger streaming systems, and those systems are what needs to be compared.

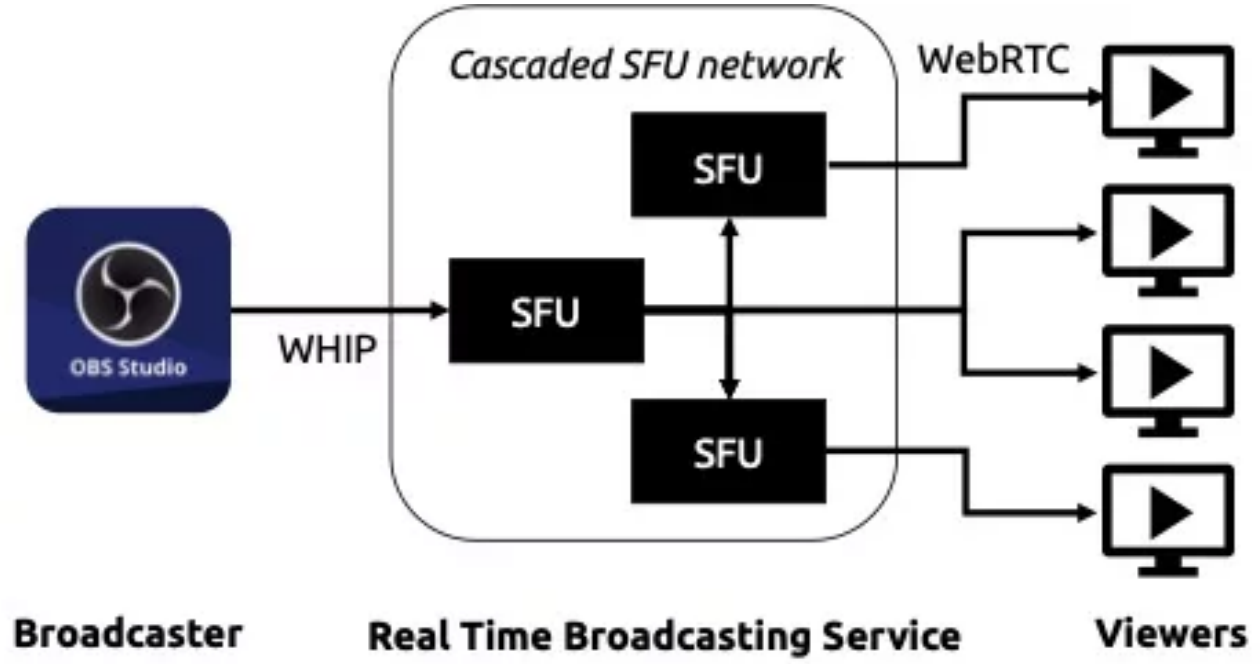
Real-Time Messaging Protocol (RTMP) dates to 1996. It was the only way to send an audio/video stream to a browser for a long time. RTMP has many variants with many capabilities that can in theory match much of what basic WebRTC offers. However, Flash was removed from browsers and its development has been near frozen for a decade. Because of this you can't natively broadcast or receive a RTMP stream in any major browser. This stands in stark contrast to WebRTC, which is native to every major browser.

In OBS today, RTMP is typically used to broadcast a single RTMP stream to a service. That service more-often-than-not converts that stream to HTTP Live Streaming (HLS). The HLS viewer grabs the HLS files for the resolution appropriate to the user's display size and bandwidth. Instead of HLS, some services may use MPEG Dynamic Adaptive Streaming (DASH) or use Low-Latency HLS (LL-HLS) with smaller chunks to reduce latencies, but the conversion mechanism and fundamentally the same, just with smaller delays.



High-level diagram of a typical broadcast service using RTMP and HLS. Decoding, encoding, and file distribution add inherent network transmission delays

The real goal of WHIP / WebRTC is not to just replace RTMP – it is to replace the whole "decode and convert this media chunk and make it available for download" that inevitably adds seconds of latency of more between the broadcaster and the viewer. Decoding, encoding, and file distribution along with the necessary buffering between these steps takes time. WebRTC systems avoid this latency by just relaying the stream to the user. WebRTC Selective Forwarding Units (SFU) only need repacketize the stream with minimal buffering, avoiding these slower steps.



High-level diagram of a real time broadcasting service using a cascaded SFU network. SFUs quickly duplicate and relay media with minimal latency

Latency vs. Media Quality

RTMP is designed to be cached and is subject to head-of-line issues when the packets don't get through in nice sequential order with consistent timing. Unless you have a large buffer that adds a lot of latency, this causes stutters and freezes at the player. However, the server eventually gets all those packets, so the quality is fine for later playback.

WebRTC optimizes for real time streaming. It has Bandwidth Estimation (BWE) mechanisms that auto adapt the bitrate – and thus the video quality – to the available bandwidth. WebRTC's has resiliency mechanisms like Forward Error Correction (FEC), NACK, and others that help to recover missing media and try very hard to ensure the stream is always flowing. To make this work, WebRTC drops information that doesn't get there in time. More importantly, it tells the sender to send less information to help reduce congestion.

This does mean if you are recording at the server, the quality will fluctuate since the information is either lost or never transmitted in the first place. If your goal is to have the highest quality stream recording as possible, then the RTMP→HLS maybe better. However, if you primarily care about the user's real time experience, then WebRTC/WHIP is the only choice. Reality is often more complicated – especially if your encoder is getting overrun and can't keep up. If you care about reliability in the sense that your live viewers will see something even if the network or your computer is overloaded, then WebRTC's many built-in forms of adaptivity give it the advantage.

What about recording quality?

Most broadcasts are recorded for playback later and no one likes a low-quality recording. The choice between real time quality vs. playback quality is not always an easy one to make. However, I would argue that if you really care about high-quality recordings, you are better off recording locally at a high bitrate and then upload that later when you are not worried about bandwidth availability for the live stream. OBS provides separate encoding options for the stream vs. the recording to help enable this.

Or you can get into even more complicated systems like we discussed in the WebCodecs and WebTransport discussion where you have more control over what happens to your media frames.

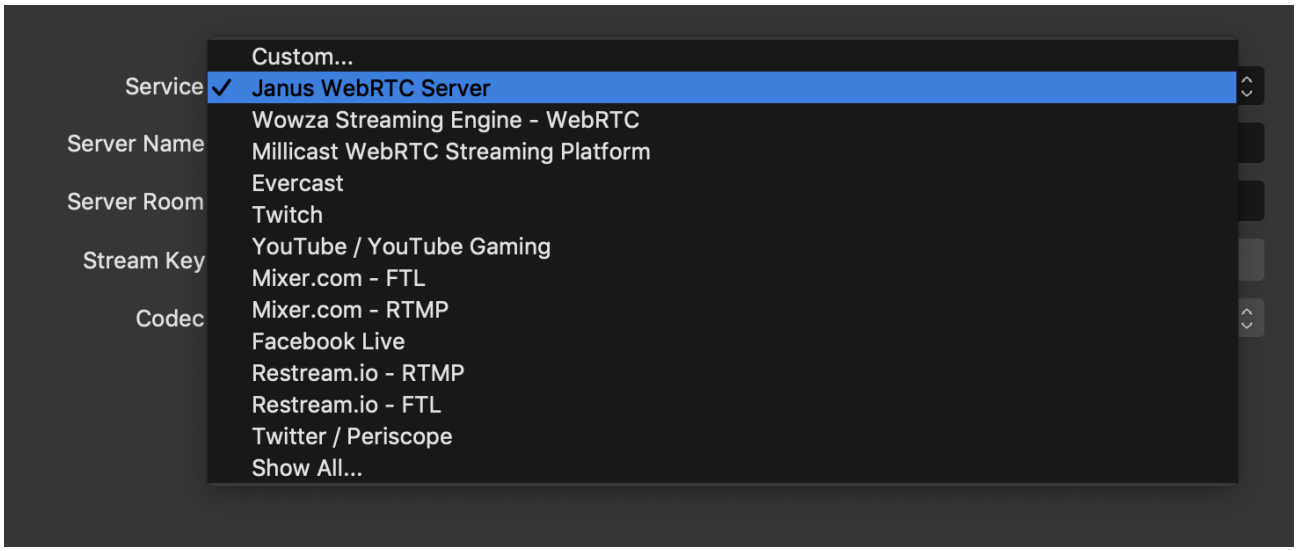
The long courtship that brought WebRTC to OBS

This is not the first attempt to put WebRTC in OBS. Let's take a quick look at some past attempts, some of which are still active.

OBS-studio-webrtc

The first real attempt at adding WebRTC support was with CoSMoSoft's OBS-studio-webrtc fork of OBS back in 2018. This initial version streamed to a Janus with other platforms / services added later, notably Millicast, the service CoSMoSoft evolved into and was acquired by Dolby.io in 2022. Dolby still maintains that here for use with Dolby's live streaming service. That implementation is based on libwebrtc.

There was an attempt by SCG82 to merge Cosmo's OBS-studio-webrtc do into the main OBS project in August of 2019. The OBS maintainers never accepted that merge, so you had to download one of these "unofficial" forks if you wanted WebRTC in OBS.



2019 Attempt to add WebRTC to OBS using CoSMOSoft's OBS-studio-webrtc project. Source: https://github.com/obsproject/obs-studio/pull/2009

Rust

Colin Edwards (ddrboxman) attempted to create another OBS WebRTC plugin a year ago in the Summer of 2022. This differs from Cosmo's fork mainly in that it uses webrtc.rs – a Rust implementation of WebRTC. libwebrtc's enormous size and build difficulty wasn't going to work for OBS's simple build requirement and webrtc.rs was an easier to work-with alternative. Why webrtc.rs of all available WebRTC stack options. As Colin put it:

> *I was writing up the initial RFC for WebRTC support in OBS. I built a prototype with the Amazon Kinesis Video Streams C WebRTC SDK since libobs is mostly C. I was itching to play around with Rust though and ended up using webrtc-rs for the proof of concept that got published in the first PR for WebRTC support.*

This implementation also exclusively used the WebRTC-HTTP Ingestion Protocol (WHIP) for signaling – the standard signaling interface designed for broadcasting authored by CoSMo/Dolby's Sergio Murillo (and webrtcHacks author). WHIP is critical here because it allows a standard interface into OBS. Without it every service that wanted to interface with OBS would need to write its own signaling plug-in.

Colin was a long-time OBS contributor, so this attempt seemed promising. However, this attempt ultimately did not work. This was largely because of integration of Rust and CMake didn't end up being that easy resulting in long compile times. There were packaging issues that made building on Linux difficult. In the end, it just wasn't worth it, as Colin remarked:

> *all of the solutions we could come up with at the time for properly integrating rust into our workflow just weren't up to our standards. Hopefully we can get to a place in the future where Rust does have a home in OBS though.*

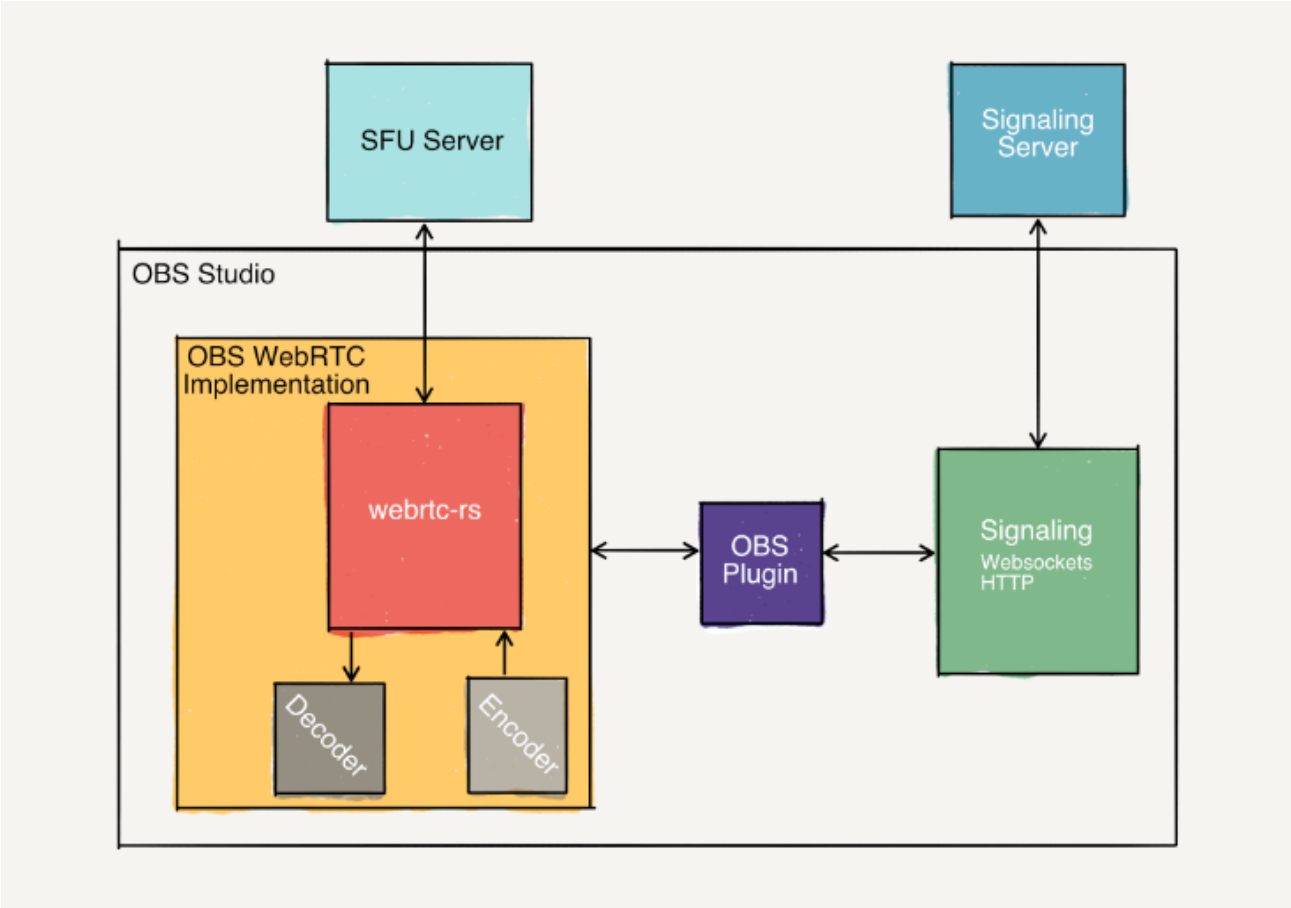


Diagram from the Colin Edwards (ddrboxman) OBS merge attempt using webrtc-rs. Source: https://github.com/obsproject/obs-studio/pull/7192

Third try is a Charm

While Colin's project never made it to master, it did seem to ignite a lot more interest in officially adding WebRTC support in OBS. Sean DuBois of pion and Colin and other regular OBS contributors John Bradley (kc5nra), tt2468, and pkviet submitted a brand new Pull Request (PR). This PR was also used WHIP as the signaling interface. Unlike previous attempts, this pull used Paul-Louis Ageneau's libdatachannel. Despite the project's name, *libdatachannel* does support media setup in addition to data channels. Importantly, that stack is based on C++, aligning with the existing OBS build system.

The development effort took many months. Sean volunteered to help when the Rust projects stalled thinking it would be relatively quick to resolve remaining issues. Instead, it has involved many rewrites, hoping for a favorable review from the core OBS team. After all the struggles, this effort finally produced the desired result – Colin merged the PR on June 9, 2023.

Since then there have been regular updates and improvements to the *obs-webrtc* plugin – mostly

minor bug fixes and build updates.

The OBS 30 Beta 1 package was released on Thursday, August 17 with Beta 2 days later. Keep an eye on for that and newer releases here.

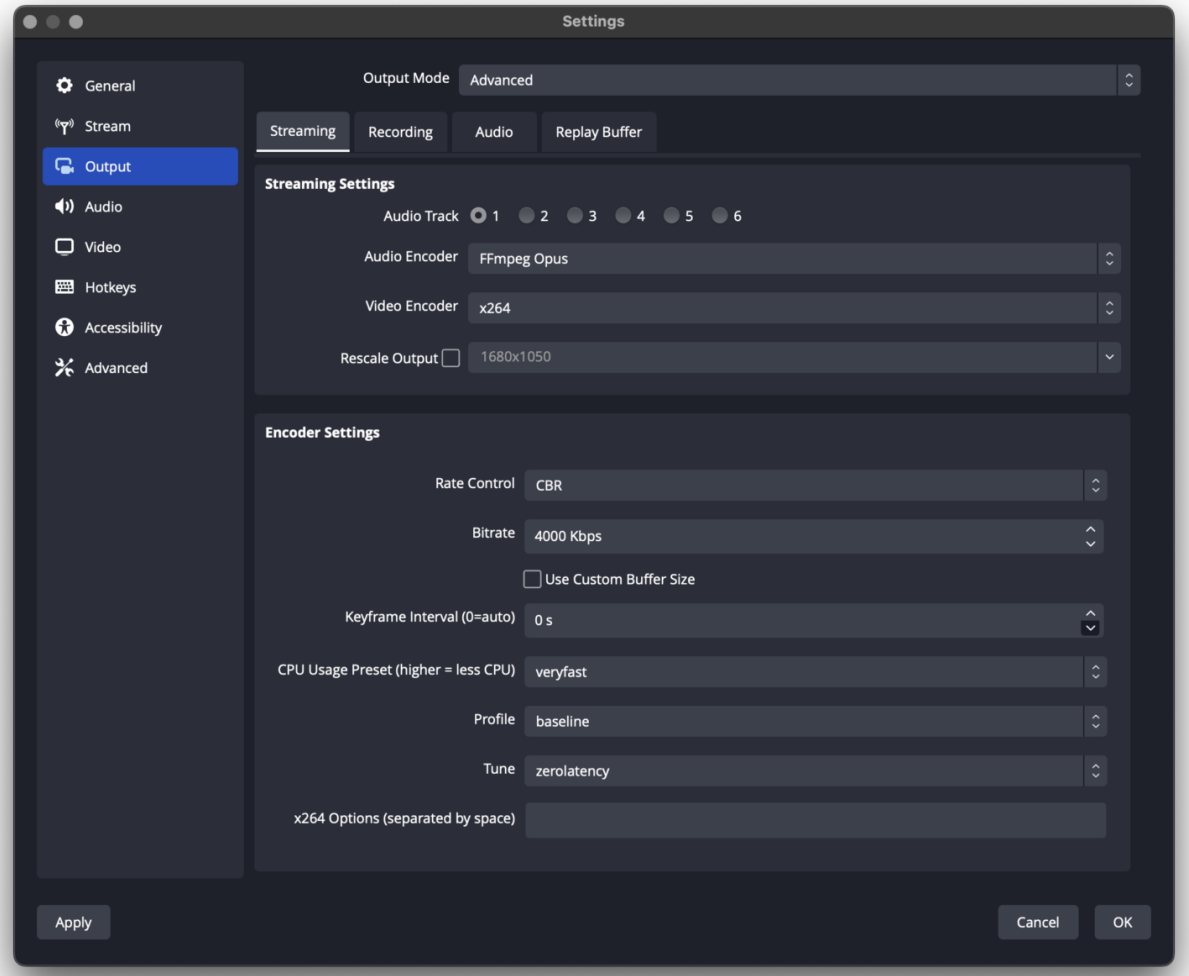
Implementation review

The initial implementation is a basic one:

Feature	Support	Notes
WebRTC Stack	libdatachannel	
Signaling	WHIP	
ICE support	Based on WHIP provider	
Audio codecs	Opus	Includes inband FEC
Video codecs	H.264	I could only get baseline profile to work
DataChannels	Not used	Not part of the WHIP spec
BWE	None	
Simulcast	No	Follow this
SVC	No	
RED	No	

Recommended Settings

I ran a test using Broadcast Box and got it working on my Mac without problems. I was able to broadcast a 1920x1080x30fps@4000kbps stream locally without any major issues using the following settings:



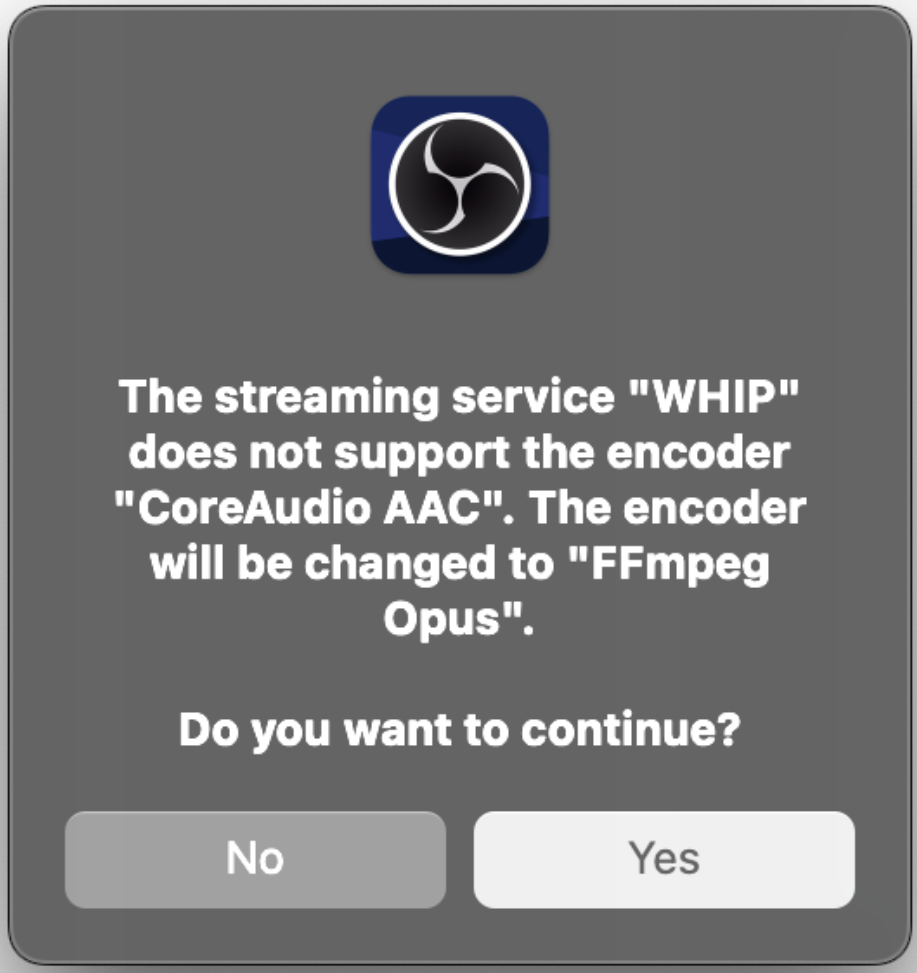
The first time I tried this with a self-built OBS, the webrtc-internals `total interframe delay` showed around 275 ms, but my glass-to-glass latency was about 750ms. When I tried it again with OBS 30 Beta 2, my glass-to-glass latency was only 42 ms, with `total interframe delay` around to 33 ms. Maybe I did something wrong in the build or I ran into a bug when I first tried? No matter, Beta 2 worked better than expected.

Settings experiments

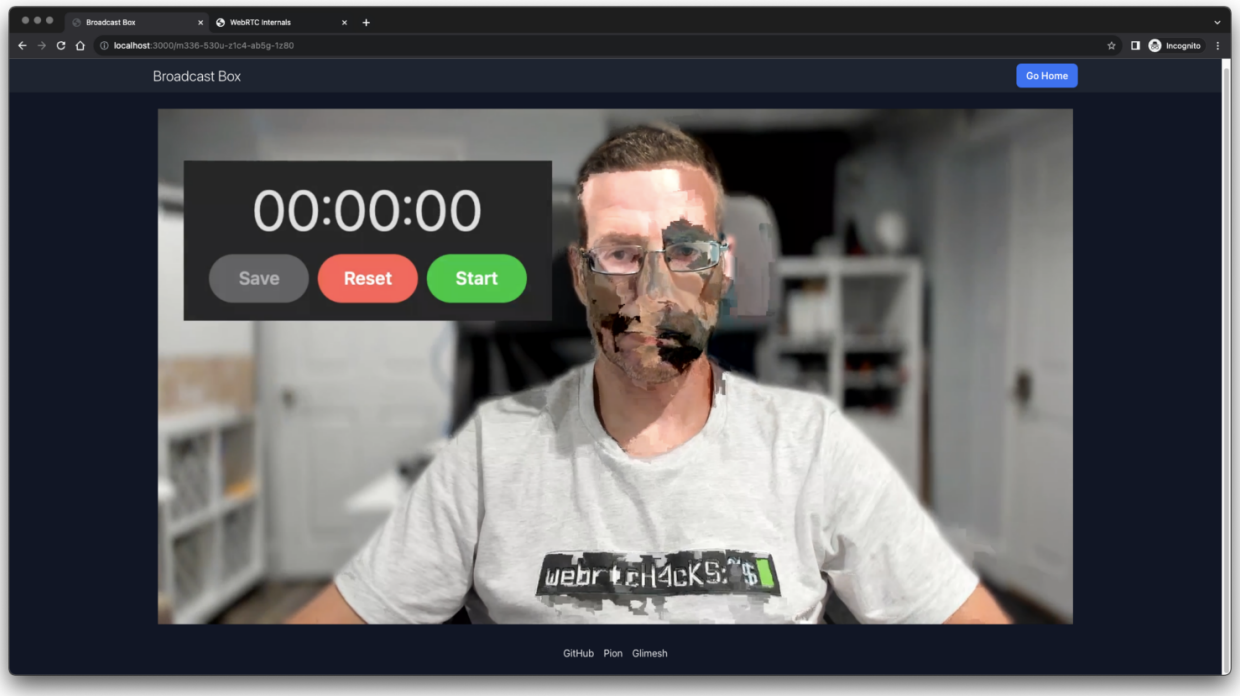
I only have the H264 encoders available on my Mac. I was able to use the Apple VT H264 Hardware Encoder, but it caused a lot of stuttering in the video. The Apple VT H264 Software Encoder made my OBS crash. The x264 gave me no issues. I tried the various encoding settings and they all seemed to work, though the encoding profile always showed `42001f` on the browser-side, indicating only Base Profile 3.1 is supported.

The HEVC (H.265) codecs and AOM AV1 codec options in OBS are not available with the WHIP option today.

When you select WHIP, the codec automatically changes to Opus. This came through as expected when inspecting in webrtc-internals.



I had problems if I used anything other than the `zerolatency` option.



It's easy to overload the OBS h264 encoder if you try ☺

Concerns

My main concerns are mostly around the maturity of the implementation.

Is libdatachannel mature enough?

As explained here, the *obs-webrtc* authors chose to use *libdatachannel* for their WebRTC stack because:

- They could build it in 9 seconds (vs. 7 minutes for *libwebrtc*)
- It is based on C++ with CMake, which aligns with the OBS build environment

Ok, libwebrtc is certainly the most mature, but it does come with a lot of baggage. Fortunately, there are many popular WebRTC stacks that have emerged, some of which have been referenced already like pion and webrtc.rs that are supported by large communities with many active projects. More usage generally means fewer problems when it comes to interoperability and feature support.

How does the *libdatachannel* project compare? *libdatachannel* was started by Paul-Louis Ageneau back in min 2019. When analyzed the WebRTC GitHub repos for FOSDEM earlier this year, *libdatachannel* came in 46th in terms of number of unique contributors and 52nd by my popularity

metric. Not too shabby, but not near some of the other projects mentioned.

Will this WebRTC stack cause issues when OBS's many users start using it? According to the webrtc-echoes project, *libdatachannel* was able to setup a `RTCPeerConnection` with all the other major WebRTC stacks. This is good, but the tests are not as comprehensive as you would get with a project like KITE.

Still, given the very restricted use case for WHIP in OBS at this point, the surface for something to go wrong is relatively small. The added attention from deployment inside OBS will give the project extra attention from the community too. Hopefully this initial OBS implementation kicks off an improvement cycle as streaming vendors add WHIP support and help do more testing.

Missing bandwidth estimation (BWE)

Another concern – for me at least – is the lack of Bandwidth Estimation in the OBS implementation. As disused in the Latency vs. Media Quality section above, the ability to regulate bandwidth usage for real time use case is one of the best parts of WebRTC. RTMP users are used to specifying their bandwidth, so perhaps this hasn't made it to the request list yet.

Unlike some of the other features discussed in the next section, I don't even see anyone explicitly working on this. Perhaps some of these features will come to *libdatachannel*, which would make adding these capabilities to OBS more manageable. Thread on that here

Roadmap

The team behind *obs-webrtc* – mostly Sean – has a couple of items that are in progress that will make it better:

- AV1
- Simulcast

AV1

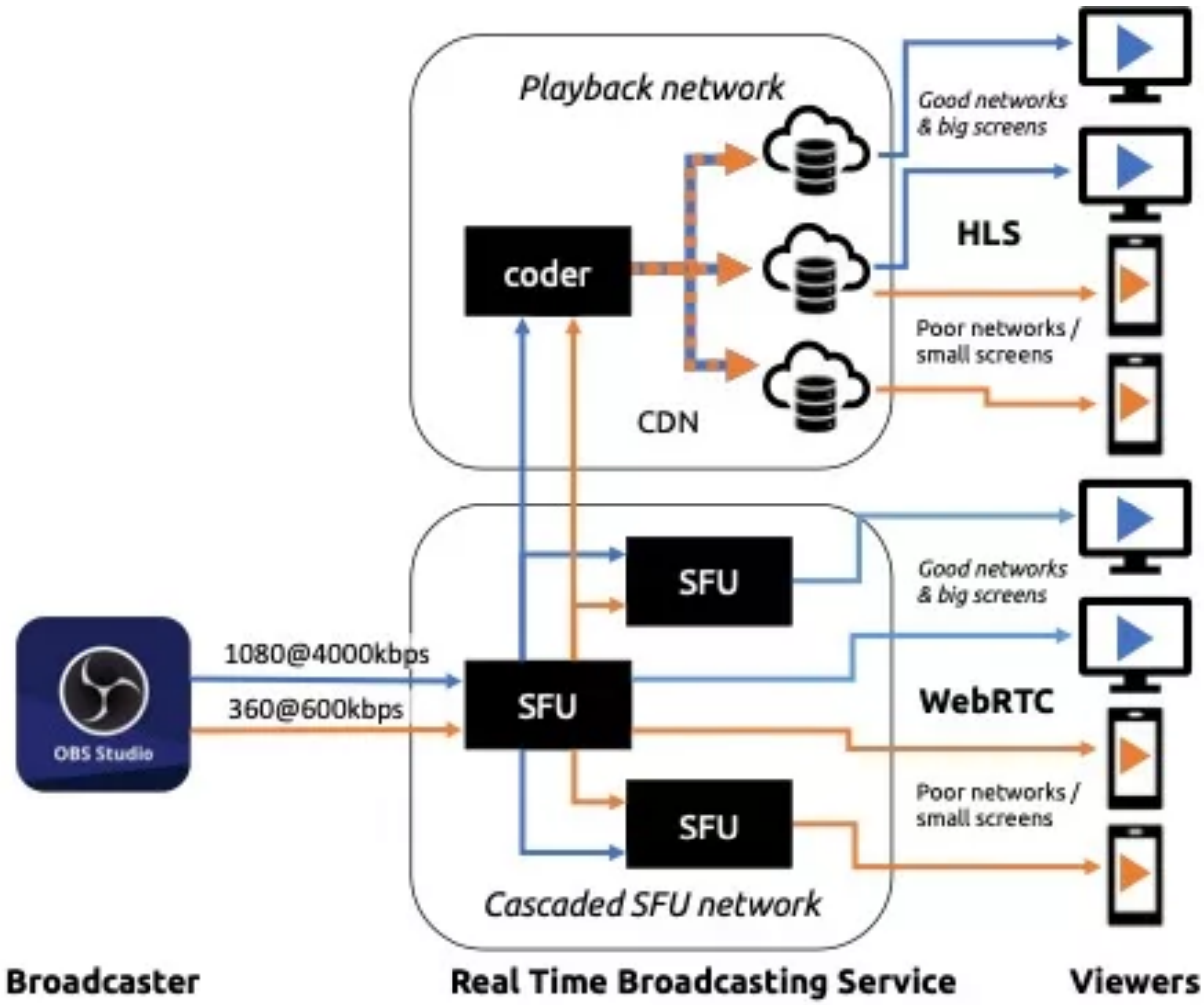
As noted above AV1 is already supported in OBS. AV1 will give you about 30% to 40% more pixels for the same bandwidth vs H264 (or 30-40% less bandwidth for the same input). AV1 is available with WebRTC inside Chrome (try on the Change Codecs sample).

Sean is working on AV1 support with *obs-webrtc* here. Notably, he had to migrate *libdatachannel* to C++ from C. C was used originally due to OBS build issues, but that was resolved here.

Simulcast

WebRTC can auto-optimize its bandwidth usage, but as noted above, *obs-webrtc* does not include any Bandwidth Estimate (BWE). Outside of this mechanism, most modern WebRTC services use WebRTC simulcast where the user sends multiple quality levels simultaneously. Simulcast does "cost" the user some extra processing overhead and bandwidth, but it is far more efficient for the service since it can simply forward the appropriate stream level. When optimizing for latency, there is no time to decode and re-encode the stream – the user needs to provide all the stream levels needed .

If the WHIP service is distributing the stream via WebRTC, there is nothing addition it needs to do. If the service is converting the streams to HLS, it can save some time and processing overhead by not needing to reencode a lower bitrate.



How broadcasting with Simulcast could look for both real-time, low-latency viewers and for playback network offloading. Diagram simplified to only show 2 layers.

Follow the simulcast RFC & PR

Sean made an RFC for this here with discussion here. He describes the benefits as:

- Generation Loss – Decoding and re-encoding videos causes generation loss. Simulcast means encodes come from the source video which will be higher quality.
- Higher Quality Encodes – Streamers with dedicated hardware can provide higher quality encodes. Streaming services at scale are optimizing for cost.
- Lower Latency – Removing the additional encoding/decoding allows video to be delivered to users faster.
- Reduce server complexity – Users find it difficult to setup RTMP->HLS with transcodes. With Simulcast setting up a streaming server becomes dramatically easier.

Conveniently, he also started a PR already here with active development on this fork. Much of the discussion is how to expose this functionality to the user. WebRTC services set their specific simulcast layers in both the client and SFU. The WHIP approach creates a generic WebRTC client that can talk to any WHIP interface. Fortunately, this can be negotiated in the SDP, so it looks like OBS could just include a checkmark for this. Still, OBS will need to choose the layer details or eventually expose that to users to select to match the service.

WebRTC Potential

Once WebRTC is in OBS, the potential will be there to use the stack for other purposes. Services like StreamYard, StreamLabs, Riverside, and many others create recording studios where guests can join for a group discussion. Conceivably an OBS user could create peer-to-peer connections with guests to replicate a similar experience in OBS without requiring third party software – perhaps even using WHEP, the **E**gress pair to WHIP's **I**ngress .

For example, Sean has a OBS2Browser project that runs a very basic WHIP server locally so you can send audio/video from your browser into OBS.

Thoughts for now

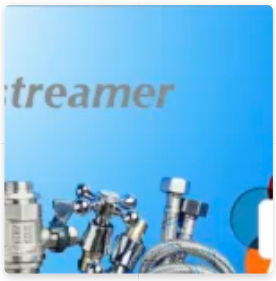
WebRTC via WHIP in OBS 30 is a key first step to seeing more of WebRTC for streaming use cases. This initial implementation is fairly bare bones, but it gets to the job done. The path to using WebRTC to replace RTMP as an ingest protocol for streaming services will be a very long one, but at least we have passed this first milestone.

{"author": "chad hart"}

Related Posts

WebRTC vs. MoQ by Use Case

2024 WebRTC in Open Source Review: A Quantitative Analysis



WebRTC Plumbing with GStreamer

How Cloudflare Glares at WebRTC with WHIP and WHEP

RSS FEED

Subscribe

Comments

Fact Checker says

August 22, 2023 at 10:42 am

Its called “Open Broadcaster Software – Studio”

Not “Open Broadcast Studio”

Reply

Chad Hart says
August 25, 2023 at 6:37 pm

corrected – thanks!

Reply

Liubomyr says
December 7, 2024 at 6:27 am

Which platforms besides Twitch are supporting streaming with WHIP? YouTube API Explorer shows that “webrtc” is one of the possible ingestion types, but I could not create live using this ingestion type in API Explorer. (https://developers.google.com/youtube/v3/live/docs/liveStreams/insert).

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

SUBSCRIBE

webrtcH4cKS:~$ guides and information for WebRTC developers

SITE

Post List

About

Contact

CATEGORIES

Guide

Other

Reverse-Engineering

Review

Standards

Technology

Uncategorized

TAGS

apple Blackbox Exploration Brief Chrome code computer vision DataChannel debug e2ee Edge gateway getUserMedia Google Meet ICE ims insertable streams ios ip leakage janus jitsi MCU MoQ NAT Opus ORTC Promo Q&A quic raspberry pi Safari SDP sfu simulcast standards TURN video vp8 w3c Walkthrough Web Audio webcodecs webrtc-internals webtransport WHIP wireshark

FOLLOW

Twitter

YouTube

GitHub

RSS