

WebRTC take 2, a decade later

2026 GStreamer conference

Matthew Waters

10 October 2026

Who Am I?

GStreamer developer and maintainer for over a decade

WebRTC, Vulkan, OpenGL

The beginning

```
commit 1894293d6378c69548d974d2965e9decc1527654
Author: Matthew Waters <matthew@centricular.com>
Date: Tue Jan 31 20:56:59 2017 +1100
```

```
webrtcbin: an element that handles the transport aspects of webrtc connections
```

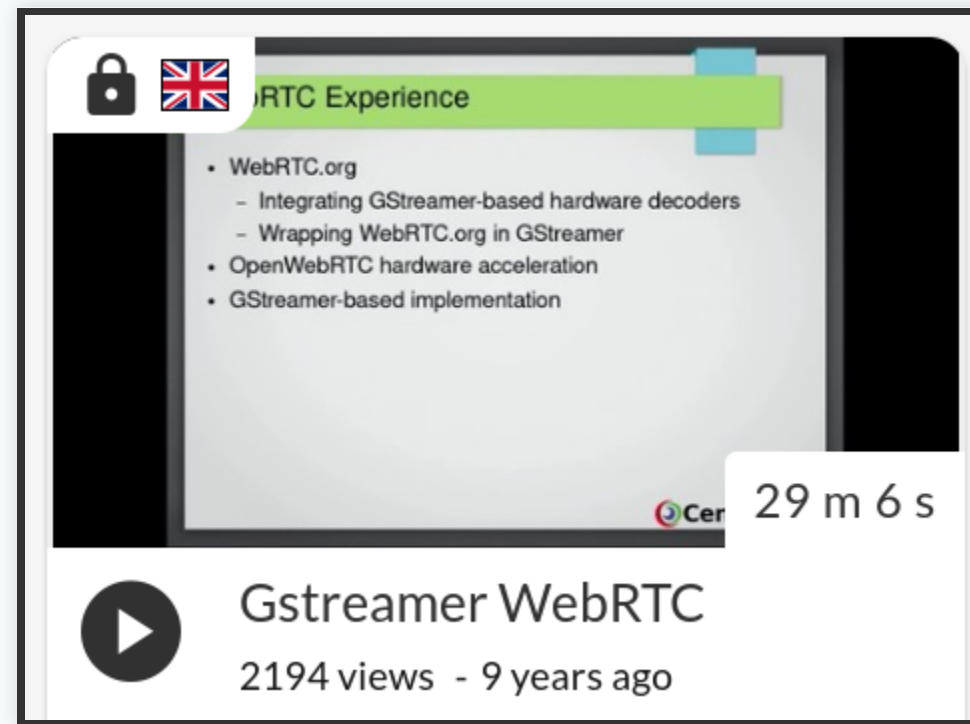
```
SDP's are generated and consumed according to the W3C PeerConnection API
available from https://www.w3.org/TR/webrtc/
```

```
...
```

Work had been underway throughout the previous year.

The beginning

GStreamer WebRTC was first presented at the [GStreamer conference 2017](#) in Prague.



Later

`webrtcsink` and `webrtcsrc` elements were introduced using `webrtcbin` with automatic encoding and decoding of audio and video and support for handling signalling with different providers such as:

- Amazon's KVS
- LiveKit
- WHIP/WHEP
- Janus' video room
- Unreal Engine's Pixel Streaming
- Custom signaller

The good

The good

1. Matching the call flow of the SDP API has increased compatibility and ease of use for applications with a web component.

The good

1. Matching the call flow of the SDP API has increased compatibility and ease of use for applications with a web component.
2. Supported features have grown to support a large subset of the available Web APIs.

The bad

The bad

1. Threads, too many with multiple `webrtcbin` elements.
Problematic for SFU use cases with many `webrtcbin` elements.

The bad

1. Threads, too many with multiple `webrtcbin` elements.
Problematic for SFU use cases with many `webrtcbin` elements.
2. Receiving data that is eventually sent out the same WebRTC connection (SFU/MCU/etc) requires wormhole elements.

The bad

1. Threads, too many with multiple `webrtcbin` elements.
Problematic for SFU use cases with many `webrtcbin` elements.
2. Receiving data that is eventually sent out the same WebRTC connection (SFU/MCU/etc) requires wormhole elements.
3. SDP media matching is suboptimal and is simple intersection.

The bad

1. Threads, too many with multiple `webrtcbin` elements.
Problematic for SFU use cases with many `webrtcbin` elements.
2. Receiving data that is eventually sent out the same WebRTC connection (SFU/MCU/etc) requires wormhole elements.
3. SDP media matching is suboptimal and is simple intersection.
4. Automated testing against real situations is suboptimal.

Counting threads with `webRTCbin`

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.
2. `libnice` (IO) thread with a dedicated `GMainLoop`.

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.
2. `libnice` (IO) thread with a dedicated `GMainLoop`.
3. DTLS timeout thread - mostly idle and only sporadically relevant when handling missed/dropped DTLS control messages.

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.
2. `libnice` (IO) thread with a dedicated `GMainLoop`.
3. DTLS timeout thread - mostly idle and only sporadically relevant when handling missed/dropped DTLS control messages.
4. RTCP timeout thread per WebRTC session (inside `rtpbin`).

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.
2. `libnice` (IO) thread with a dedicated `GMainLoop`.
3. DTLS timeout thread - mostly idle and only sporadically relevant when handling missed/dropped DTLS control messages.
4. RTCP timeout thread per WebRTC session (inside `rtpbin`).
5. RTP jitter buffer thread/s per media SSRC.

Counting threads with `webrtcbin`

1. Signalling thread - only used for call setup and mostly idle while data flow is in progress.
2. `libnice` (IO) thread with a dedicated `GMainLoop`.
3. DTLS timeout thread - mostly idle and only sporadically relevant when handling missed/dropped DTLS control messages.
4. RTCP timeout thread per WebRTC session (inside `rtpbin`).
5. RTP jitter buffer thread/s per media SSRC.
6. Streaming threads.

How to use less threads

Share them! Especially the threads that are mostly idle.

In Rust this is an async runtime.

Introducing webrtcsend/webrtcrecv

- Splitting send and receive halves into separate elements solves the wormhole element problem by having data flow from the receive element to the send element within the pipeline.
- Same design as used by rtpsend/rtprecv which you [can read about](#) or watch [this GStreamer conference talk from 2 years ago](#).

```
webrtcrecv id=some-unique-id ! processing... !  
webrtcsend id=some-unique-id
```

Less threads with `webrtcsend/` `webrtcrecv`

Less threads with `webrtcsend/` `webrtcrecv`

- Signalling, DTLS timeouts, and IO all use the same async runtime threads and are shared across multiple `webrtcsend/webrtcrecv`.

Less threads with `webrtcsend/` `webrtcrecv`

- Signalling, DTLS timeouts, and IO all use the same async runtime threads and are shared across multiple `webrtcsend/webrtcrecv`.
- RTCP timeout threads and RTP jitter buffer threads are also reduced by using `rtpsend/rtprecv`.

Less threads with `webrtc` send/ `webrtc` recv

- Signalling, DTLS timeouts, and IO all use the same async runtime threads and are shared across multiple `webrtc` send/`webrtc` recv.
- RTCP timeout threads and RTP jitter buffer threads are also reduced by using `rtp` send/`rtp` recv.
- SCTP (future) timeouts can use the same async runtime threads.

Less threads with `webrtcsend/` `webrtcrecv`

- Signalling, DTLS timeouts, and IO all use the same async runtime threads and are shared across multiple `webrtcsend/webrtcrecv`.
- RTCP timeout threads and RTP jitter buffer threads are also reduced by using `rtpsend/rtprecv`.
- SCTP (future) timeouts can use the same async runtime threads.
- Further improvements in streaming thread usage can be achieved by configuring the use of threadshare elements.

Components within webrtcsend/ webrtcrecv

Components within webrtcsend/ webrtcrecv

- ICE - [librice](#)

Components within webrtcsend/ webrtcrecv

- ICE - [librice](#)
- DTLS - uses `openssl` but could use [dimpl](#) in the future if the C dependency is a concern.

Components within webrtcsend/ webrtcrecv

- ICE - [librice](#)
- DTLS - uses `openssl` but could use [dimpl](#) in the future if the C dependency is a concern.
- SRTP - `srtppenc/srtppdec` elements but could be done manually internally.

Components within webrtcsend/ webrtcrecv

- ICE - [librice](#)
- DTLS - uses `openssl` but could use [dimpl](#) in the future if the C dependency is a concern.
- SRTP - `srtppenc/srtppdec` elements but could be done manually internally.
- SDP - [sdp-types](#) with some WebRTC specifics on top.

Current features

- Audio and Video streaming.
- BUNDLE required for multiple media.
- `rtcp-mux` is required.

Primary goal - feature parity with `webrtcbin`

- SCTP and Data Channels likely using `sctp-proto`
- Retransmissions (RTX) and (FEC) Forward Error Correction (requires support within `rtpsend/rtprecv`).
- SDP renegotiation
- Simulcast for video
- Statistics

Further goals

- Better SDP negotiation than `webrtcbin` using actual SDP offer/answer rules for each payload format.
- SDP rollback and partial answer support.

Thanks

A blog post version of this is [available here](#).

- ystreet00 on #gstreamer on OFTC
- <https://gitlab.freedesktop.org/ystreet>
- ystreet00@floss.social on mastodon
- matthew at centricular dot com