RtpTransport Breakout Session September 25, 2024

Erik Språng, Peter Thatcher, Harald Alvestrand, Bernard Aboba

TPAC 2024
Anaheim CA, USA
Hybrid meeting

23–27 SEPTEMBER 2024

W3C Code of Conduct

- This meeting operates under W3C Code of Ethics and Professional Conduct
- We're all passionate about improving WebRTC and the Web, but let's all keep the conversations cordial and professional

Safety Reminders

While attending TPAC, follow the health rules:

- Masks and daily testing are left to individual choice

Please be aware of and respect the personal boundaries of your fellow participants

For information about mask and test availability, what to do if you are not feeling well, and what to do if you test positive for COVID, see:

https://www.w3.org/2024/09/TPAC/health.html

Virtual Meeting Tips (Zoom)

- Both local and remote participants need to be on irc.w3.org channel #rtp-transport.
- Use “+q” in irc to get into the speaker queue and “-q” to get out of the speaker queue.
- Please use headphones when speaking to avoid echo.
- Please wait for microphone access to be granted before speaking.
- Please state your full name before speaking.

For Discussion Today

- Why RtpTransport?
- What Can I do with RtpTransport?
- Potential Footguns
- Demo
- Feedback

Why RtpTransport?

- WebRTC provides a lot of benefits
  - Compatibility with a large variety of endpoints
  - Highly optimized for low-latency use cases
  - Ease of use* - the monolith provides reasonable behavior out of the box

- At the same time, it is somewhat of a black box. Frames go in, packets come out, and only limited knobs are available to the user.

Why RtpTransport? (cont.)

- RtpTransport allows behaviours to be “unbundled” from WebRTC, should you wish to implement custom:
  - Codecs
  - Packetization/depacketization
  - Bandwidth Estimation
  - Bitrate Allocation
  - Resource Adaptation
  - Robustness/Recovery Mechanisms
  - Scalability Modes
- Default behavior still available where customization isn’t needed
  - Can customize behavior for specific media or streams, use default behavior for everything else

- WebRTC provides peer-to-peer support, low latency transport, encryption and compatibility with existing endpoints - providing significant benefits over WebTransport for RTC use cases.

Codec Flexibility

Use any codec:

- Built-in codecs
  - WebCodecs
- Custom codecs
  - WASM …
- Custom reference structures
- Custom rate control
- Custom adaptation due to CPU/Bandwidth/Quality constraints
- Customize all the things!

As long as you de-packetize what you packetize, you can use a codec with other WebRTC endpoints by sending/receiving over RtpTransport.

Examples

- Audio
  - Prototyping of new audio codecs (Opus 1.5, Lyra, Satin, etc.)
  - Transport of immersive audio containers (e.g. IAMF)
  - Codecs supported by WebCodecs but not WebRTC (AAC, FLAC, etc.)
- Video
  - Custom packetization (e.g. more equal packet sizes)
  - Custom RTP header extensions
  - WebCodecs-based support for:
    - Custom keyframe generation
    - Custom hardware/software failover
    - Custom rate adaptation (e.g. per-frame QP)
    - Long-term references (LTRs)
    - Spatial scalability with Layer Refresh (LRR)
- Text
  - Support for Realtime Text (RTT, RFC 4103)

Custom Robustness/Recovery Mechanisms

With custom RTCP, the application can make moment-by-moment decisions how it wants to deal with packet loss, delay and bandwidth by employing any combination of:

- NACK
- FIR/PLI
- FEC
- Long-Term Reference (LTR)
  - Avoids having to send a keyframe
  - Requires new WebCodecs encoder API
- Layer Refresh Request (LRR)
  - Avoids having to send a keyframe to recover from spatial layer loss
  - Requires new WebCodecs encoder API
- Frame drops
- [Insert your novel coding technique here]

Sample: Custom Codecs

```
const [pc, rtpSender] = await customPeerConnectionWithRtpSender();
const source = new CustomSource();
const encoder = new CustomEncoder();
const packetizer = new CustomPacketizer();
const rtpSendStream = await rtpSender.replaceSendStreams()[0];
for await (const rawFame of source.frames()) {
    encoder.setTargetBitrate(rtpSendStream.allocatedBandwidth);
    const encodedFrame = encoder.encode(rawFrame);
    const rtpPackets = packetizer.packetize(encodedFrame);
    for (const rtpPacket of rtpPackets) {
        rtpSendStream.sendRtp(rtpPackets);
    }
}
```

Custom Bandwidth Estimation

WebRTC is based on the GoogCC bandwidth estimator, which provides a reasonable behavior for video conferencing.

RtpTransport enables the user to implement custom bandwidth estimation by controlling packet timing and processing of feedback signals.

This allows the user to make their own trade-off between bandwidth usage, latency and potential packet loss.

Sample: Custom Bandwidth Estimation

```
const [pc, rtpTransport] = setupPeerConnectionWithRtpTransport();  // Custom
const estimator = createBandwidthEstimator();  // Custom
rtpTransport.onsentrtp = () => {
    for (const sentRtp of rtpTransport.readSentRtp(100)) {
        if (sentRtp.ackId) {
            estimator.rememberSentRtp(sentRtp);
        }
    }
}

rtpTransport.onreceivedrtpacks = () => {
    for (const rtpAcks in rtpTransport.readReceivedRtpAcks(100)) {
        for (const rtpAck in rtpAcks.acks) {
            const bwe = estimator.processReceivedAcks(rtpAck);
            rtpTransport.customMaxBandwidth = bwe;
        }
    }
}
```

Custom Bitrate Allocation

Based on feedback from the bandwidth estimator, it is possible to distribute the total available bandwidth to the various media feeds being sent. One use case is moment-by-moment reprioritization of camera vs screenshare bitrate usage.

Sample: Custom Bitrate Allocation

```
const [pc, rtpTransport] = await setupPeerConnectionWithRtpSender();
setInterval(() => {
    for (const [rtpSender, bitrate] of
          allocateBitrates(rtpTransport.bandwidthEstimate)) {  // Custom
        const parameters = rtpSender.getParameters();
        parameters.encodings[0].maxBitrate = bitrate;
        rtpSender.setParameters(parameters);
    }
}, 1000);
```

Potential Foot-Guns

The browser should aim to help preserve the user's toes. For instance by:

- Preventing over-allocation, which could allow DoS attacks
- Providing BWE circuit breakers (RFC 8083)
- Enforcing correct RTP headers
- …

Open Issues

- Worker Support (#13, #36, #70, #71)
- Performance (#20)
- Queue depth control (#54)
- Custom RTP header extensions (#12, #29)
- Custom RTCP messages (#11)
- Interaction with SDP (#10)

For More Information

- Chromium Status
  - Intent to Protype Notice
  - Demo
- RtpTransport GitHub Repo
  - Explainer
    - Use case 1: Custom Packetization
    - Use case 2: Custom Congestion Control
    - Use Case 3: Custom RTX/FEC (not ready yet)
  - API outline
  - Specification (very early)

Demo

- Link: https://tonyherre.github.io/rtptransport-samples/basic_bwe/
- Requires a Chromium build >= 129.0.6658.0 started with the flag `--enable-blink-features=RTCRtpTransport`

Feedback

- What are your potential use cases?
  - If the API could do anything, what would you want it to do?
- What features would motivate you to use RtpTransport?
- What issues would prevent you from using RtpTransport?
- Would you join an RtpTransport Community Group?

Nitty Gritty Topics

- NACK/RTX/RTCP
  - What kind of custom NACK/RTXing would you want to do?
  - What kind of custom RTCP would you want to do?
- Workers
  - Is it a problem to require using Workers?
  - Is it a problem to only support Dedicated Workers?
- Simulcast
  - Should an “RTP send stream” API point be per-simulcast-layer (RID) or across-simulcast-layers (MID)?
- SDP
  - Is it a problem to be constrained by the negotiated SDP?
  - Would you like to be able to construct an RtpTransport without any SDP offer/answer?

Thank you for attending!