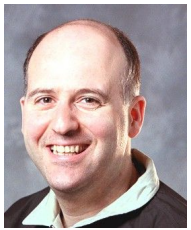
WebRTC, Mobility, Cloud, IOT and More...

IIT REAL-TIME COMMUNICATIONS

Conference & Expo Oct 10-13, 2022 Chicago

WebCodecs, WebTransport and the Next Generation of Web Media APIs

October 11, 2022



Bernard Aboba
Principal Architect, Microsoft

Jan-Ivar Bruaroey
Staff Software Engineer, Mozilla

The Pandemic Challenge

- A new wave of technology reached the mass market during the pandemic:
  - Podcasting
  - Video conferencing
  - Video streaming services (live, video-on-demand)
  - Game streaming
  - IoT devices (doorbells/security, exercise equipment, robots, smart speakers)
  - Large scale webinars, classes, “town hall” meetings (100K+ viewers)
  - Online events (auctions, conferences, sporting events, concerts)
  - Interactive entertainment (co-watching, “together mode”, etc.)
- Compiled in WebRTC-NV Use Cases, WebTransport Use Cases
- The new use cases blur the lines between “streaming” and “realtime communications” and create a new challenge:
  - Can we develop web APIs (and protocols) useful for both “streaming” and RTC applications (and combinations of both?)

Next generation Web media APIs

Overcome the “tyranny of OR”. Multi-threaded applications can deliver ***both*** low-latency ***and*** large scale, through low-level access to building blocks:

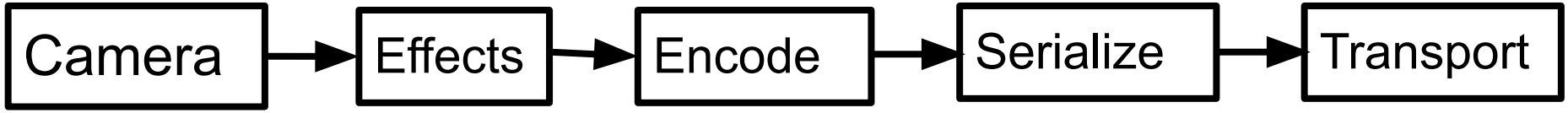
- Capture
- A/V processing (e.g. machine learning)
- Encode/Decode
- Transport (with support for caching)
- Rendering

Next generation Web media APIs

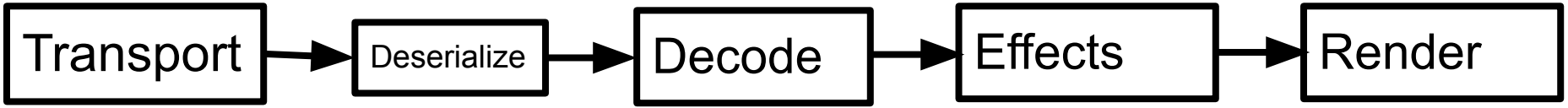
- Capture
  - Media Capture and Streams Extensions
  - Mediacapture-transform
- Encode/decode
  - WebCodecs
  - MSEv2
- Transport
  - WebTransport (HTTP/3 over QUIC)
  - WebRTC data channel in Workers (SCTP/DTLS/UDP)
- Framework
  - WHATWG Streams
  - Web Assembly

The “Pipeline” Model (WHATWG Streams)

- Send



- Receive



Last Year… and This Year

Last year at the **IIT RTC 2021 Conference**, we introduced the WebCodecs API.

This year's presentation will introduce the WebTransport API, and will demonstrate how to combine WebCodecs with WebTransport to build multi-threaded applications.

What is WebTransport?

WebTransport is a **transport protocol** (standardized in IETF WEBTRANS WG) and an **API** (standardized in W3C WebTransport WG), that enables **clients** operating under the **Web security model** to communicate with a remote **server** using a **secure, multiplexed transport**.

WebTransport provides:

- Unidirectional and bidirectional streams of reliable and ordered data.
- Support for receiving and sending datagrams
- Operation over HTTP/3, with potential fallback to HTTP/2

What is exciting about WebTransport?

- Protocol & API potentially usable in a wide range of use cases:
  - Video conferencing & telephony applications
  - Gaming
  - Low latency & live media delivery
- Looks like HTTP/3 to firewalls, proxies, network switches etc. which can greatly facilitate its reach and robustness.
- Browser support gives you billions of addressable clients (in addition to native OS support).
- Datagram access in JavaScript ☺
- When combined with WebCodecs and WebAssembly, closes the gap between native and browser RTC applications.

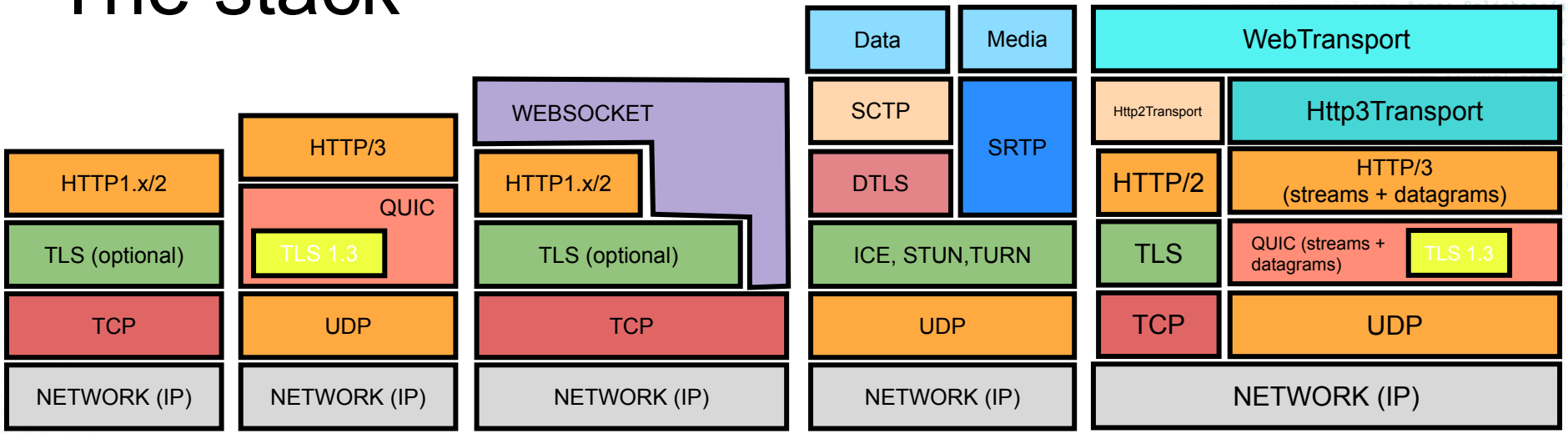
Use Cases

https://github.com/w3c/webtransport/blob/main/use-cases.md

1. Machine learning (client/server)
2. Multiplayer Gaming - web and consoles
3. Low-latency live streaming
4. Cloud Game Streaming
5. Server-based video conferencing
6. Remote desktop
7. Time Synchronized Multimedia Web communications
8. IOT sensor and analytics data transfer
9. PubSub Models - avoid long-polling

The Protocol

The stack



HTTP1.x/2 **HTTP/3** **WEBSOCKET** **WebRTC** **WEBTRANSPORT**

Bidirectional Communication on the Web

<table>
<tr><th></th><th>Client-Server</th><th>Peer-to-peer</th></tr>
<tr><td>Reliable and ordered</td><td>WebSocket
(also WebTransport)</td><td rowspan="3">RTCDataChannel</td></tr>
<tr><td>Reliable but unordered</td><td rowspan="2">WebTransport</td></tr>
<tr><td>Unreliable and unordered</td></tr>
</table>

Establishing a WebTransport Connection

https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3

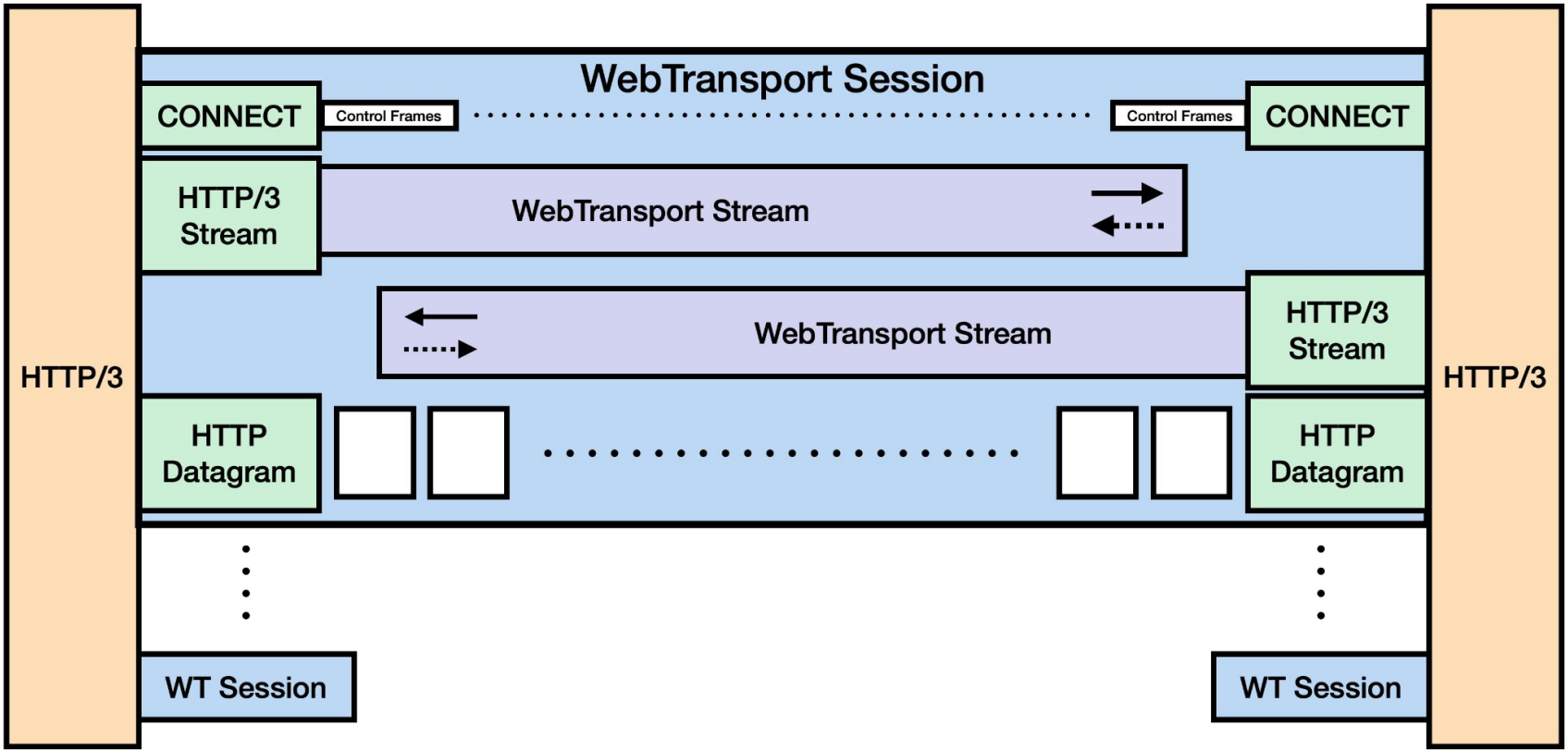
- After HTTP/3 connection is established, client and server send SETTINGS frames.
- In order to indicate support for WebTransport, both the client and the server MUST send a SETTINGS_ENABLE_WEBTRANSPORT value set to "1" in their SETTINGS frame.
- The client MUST NOT send a WebTransport request until it has received the setting indicating WebTransport support from the server.
- Similarly, the server MUST NOT process any incoming WebTransport requests until the client settings have been received, as the client may be using a version of WebTransport extension that is different from the one used by the server.

Establishing a WebTransport Connection (cont'd)

https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-http3

- Client then sends an extended CONNECT request [RFC8441].
  - If the server accepts the request, a WebTransport session is established.
  - The *CONNECT stream ID* identifies a WebTransport session within the connection, known as a *Session ID*.
  - A WebTransport session is terminated when the CONNECT stream that created it is closed.

HTTP/3


HTTP/3
WebTransport Session
CONNECT
Control Frames
Control Frames
CONNECT
HTTP/3 Stream
WebTransport Stream
WebTransport Stream
HTTP/3 Stream
HTTP/3
HTTP Datagram
HTTP Datagram
WT Session
WT Session
HTTP/3

Unidirectional Streams

4.1. Unidirectional streams

Once established, both endpoints can open unidirectional streams. The HTTP/3 unidirectional stream type SHALL be 0x54. The body of the stream SHALL be the stream type, followed by the session ID, encoded as a variable-length integer, followed by the user-specified stream data (Figure 1).

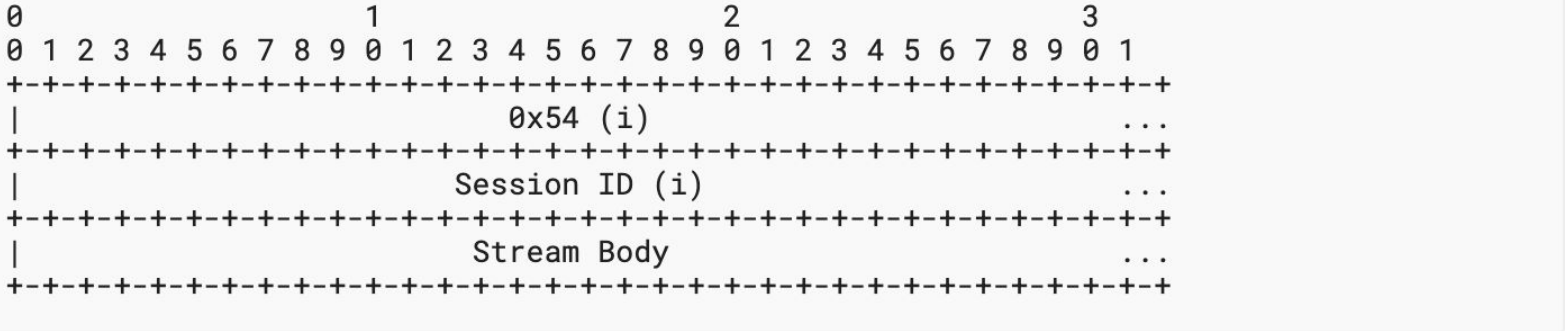


Figure 1: Unidirectional WebTransport stream format

Bidirectional Streams

4.2. Bidirectional Streams

WebTransport endpoints can initiate bidirectional streams by opening an HTTP/3 bidirectional stream and sending an HTTP/3 frame with type `WEBTRANSPORT_STREAM` (type=0x41). The format of the frame SHALL be the frame type, followed by the session ID, encoded as a variable-length integer, followed by the user-specified stream data (Figure 2). The frame SHALL last until the end of the stream.

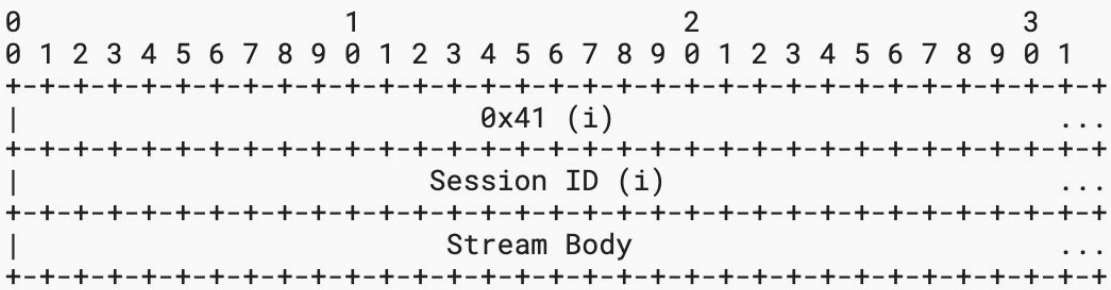


Figure 2: WEBTRANSPORT_STREAM frame format

Datagrams

RFC 9221
https://datatracker.ietf.org/doc/html/draft-ietf-masque-h3-datagram

```
QUIC Datagram {
     Type (i) = 0x30, 0x31
     [Length (i)],  //present only when Len bit is set (0x31)
     Quarter Stream ID (i),
     HTTP Datagram Payload (..),
   }
```

Quarter Stream ID: A variable-length integer that contains the value of the Stream ID that this datagram is associated with, divided by four. The division by four stems from the fact that HTTP requests are sent on client-initiated bidirectional streams, and those have stream IDs that are divisible by four.

Buffering (Section 4.5)

4.5. Buffering Incoming Streams and Datagrams

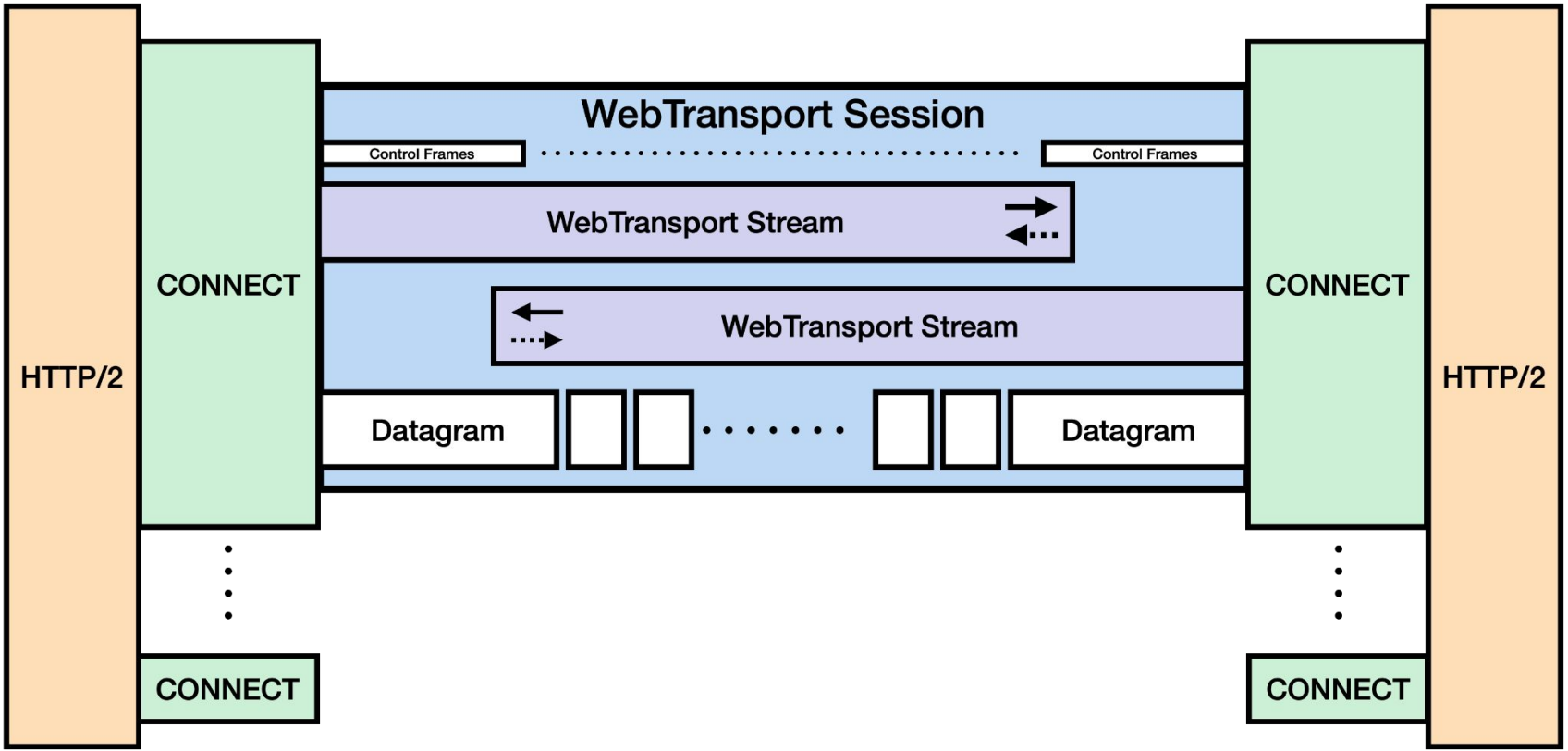
In WebTransport over HTTP/3, the client MAY send its SETTINGS frame, as well as multiple WebTransport CONNECT requests, WebTransport data streams and WebTransport datagrams, all within a single flight. As those can arrive out of order, a WebTransport server could be put into a situation where it receives a stream or a datagram without a corresponding session. Similarly, a client may receive a server-initiated stream or a datagram before receiving the CONNECT response headers from the server.

To handle this case, WebTransport endpoints SHOULD buffer streams and datagrams until those can be associated with an established session. To avoid resource exhaustion, the endpoints MUST limit the number of buffered streams and datagrams. When the number of buffered streams is exceeded, a stream SHALL be closed by sending a RESET_STREAM and/or STOP_SENDING with the `H3_WEBTRANSPORT_BUFFERED_STREAM_REJECTED` error code. When the number of buffered datagrams is exceeded, a datagram SHALL be dropped. It is up to an implementation to choose what stream or datagram to discard.

Pooling

- Pooling allows multiple WebTransport sessions to share a connection.
  - Use cases: Multiple tabs of the same application sharing a connection.
  - WebTransport load balancer multiplexing sessions to a server, over a single connection.
  - How to allocate bw/priority to the sessions?
  - No implementations, off by default in the API.
- In HTTP/3, streams and datagrams are linked to the CONNECT streamID.
- In HTTP/2, each session has one connect stream.

HTTP/2


HTTP/2
CONNECT
WebTransport Session
Control Frames
Control Frames
WebTransport Stream
WebTransport Stream
Datagram
Datagram
CONNECT
HTTP/2
CONNECT
CONNECT

The API

WebTransport status as of Oct 2022

- API in Working Draft Status at https://www.w3.org/TR/webtransport/
- Github project milestones now aligned with W3C release process - see Candidate recommendation milestone
- Browser support
  - Chrome shipped WebTransport with M97 in January 2022
    - Protocol stable as of M99.
  - Firefox anticipated having WebTransport in Nightly build by end of year.
- An echo server for Web Platform Tests is available.
- Timetable for this year and next:
  - **Sept 30 :** Candidate for Recommendation - requires stability in API
  - **Dec 30** : Proposed Recommendation - requires two independent implementations per our charter.
  - **Feb 2023:** Call for Review of a Proposed Recommendation
  - **April 2023** - Publication by W3C as a Recommendation after AC review.

WebTransport Session Functionality

capability	definition
send a datagram	[WEB-TRANSPORT-HTTP3] section 4.4
receive a datagram	[WEB-TRANSPORT-HTTP3] section 4.4
create an outgoing unidirectional stream	[WEB-TRANSPORT-HTTP3] section 4.1
create a bidirectional stream	[WEB-TRANSPORT-HTTP3] section 4.2
receive an incoming unidirectional stream	[WEB-TRANSPORT-HTTP3] section 4.1
receive a bidirectional stream	[WEB-TRANSPORT-HTTP3] section 4.2

Capabilities by Stream Type

capability	definition	incoming unidirectional	outgoing unidirectional	bidirectional
send bytes (potentially with FIN)	[QUIC] section 2.2	No	Yes	Yes
receive bytes (potentially with FIN)	[QUIC] section 2.2	Yes	No	Yes
send STOP_SENDING	[QUIC] section 3.5	Yes	No	Yes
reset a WebTransport stream	[QUIC] section 19.4	No	Yes	Yes

Webtransport Stream Signals

event	definition	incoming unidirectional	outgoing unidirectional	bidirectional
STOP_SENDING	[QUIC] section 3.5	No	Yes	Yes
reset	[QUIC] section 19.4	Yes	No	Yes

Protocol Mappings (Section 10)

API Method	QUIC Protocol Action
writable.abort(errorCode)	sends RESET_STREAM with errorCode
writable.close()	sends STREAM_FINAL
writable.getWriter().write()	sends STREAM
writable.getWriter().close()	sends STREAM_FINAL
writable.getWriter().abort(errorCode)	sends RESET_STREAM with errorCode
readable.cancel(errorCode)	sends STOP_SENDING with errorCode
readable.getReader().cancel(errorCode)	sends STOP_SENDING with errorCode
wt.close(closeInfo)	terminates session with closeInfo

QUIC Protocol -> API Effect (Section 10)

QUIC Protocol Action	API Effect
received STOP_SENDING with errorCode	errors `writable` with `streamErrorCode`
received STREAM	(await `readable`.getReader().`read`()).value
received STREAM_FINAL	(await `readable`.getReader().`read`()).done
received RESET_STREAM with errorCode	errors `readable` with `streamErrorCode`
Session cleanly terminated with closeInfo	(await wt.`closed`).closeInfo, and errors open streams
Network error	(await wt.`closed`) rejects, and errors open streams

WebTransport > WebSocket

WebSocket's message-framed semantics

```
socket.send( “Hello World!” );
socket.send( “How are you?” );
```

Receiver:

“Hello World!”

“How are you?”

(framed + ordered = blocking)

WebTransport's long-lived stream semantics

```
writer.write(utf8.encode( “Hello World!” ));
writer.write(utf8.encode( “How are you?” ));
```

Receiver:

“Hello World!How are you?”

(stream of unframed ordered data)

WebTransport's stream-per-message semantics

```
for (const msg of [ “Hello World!” , “How are you?” ]) {
  const writable = await wt.createUnidirectionalStream();
  const encoder = new TextEncoderStream("utf-8");
  const writer = writable.pipeThrough(encoder).getWriter();
  await writer.write(msg);
  await writer.close();
}
```

Receiver:

“How are you?”

“Hello World!”

(in-parallel streams arrive unordered)

WebTransport Interface

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransport {
  constructor(USVString url, optional WebTransportOptions options = {});

  Promise<WebTransportStats> getStats();
  readonly attribute Promise<undefined> ready;
  readonly attribute WebTransportReliabilityMode reliability;
  readonly attribute Promise<WebTransportCloseInfo> closed;
  undefined close(optional WebTransportCloseInfo closeInfo = {});

  readonly attribute WebTransportDatagramDuplexStream datagrams;
```

WebTransport Interface (cont'd)

```
  Promise<WebTransportBidirectionalStream> createBidirectionalStream();
  /* a ReadableStream of WebTransportBidirectionalStream objects */
  readonly attribute ReadableStream incomingBidirectionalStreams;

  Promise<WebTransportSendStream> createUnidirectionalStream();
  /* a ReadableStream of WebTransportReceiveStream objects */
  readonly attribute ReadableStream incomingUnidirectionalStreams;
};

enum WebTransportReliabilityMode {
  "pending",
  "reliable-only",
  "supports-unreliable",
};
```

WebTransport Interface (cont'd)

```
dictionary WebTransportHash {
  DOMString algorithm;
  BufferSource value;
};

dictionary WebTransportOptions {
  boolean allowPooling = false;
  boolean requireUnreliable = false;
  sequence<WebTransportHash> serverCertificateHashes;
  WebTransportCongestionControl congestionControl = "default";
};

enum WebTransportCongestionControl {
  "default",
  "throughput",
  "low-latency",
};
```

Example

```
EXAMPLE 4
async function sendData(url, data) {
  const wt = new WebTransport(url);
  const writable = await wt.createUnidirectionalStream();
  const writer = writable.getWriter();
  await writer.write(data);
  await writer.close();
}
```

Observations

- `await wt.ready` not required before `await wt.createUnidirectionalStream()` or `wt.createBidirectionalStream()`
  - Promises won't resolve before connection is established.
  - For sending datagrams, await writer.ready to avoid dropping due to exceeding `outgoingMaxAge`.
- `allowPooling` is false by default. Pooling not implemented, so…
- `getStats()` not yet implemented
  - Event for estimated bandwidth under discussion.
- HTTP/2 fallback (`wt.reliability`) not yet implemented.

Datagram Interface

```
[Exposed=(Window,Worker), SecureContext]
interface WebTransportDatagramDuplexStream {
  readonly attribute ReadableStream readable;
  readonly attribute WritableStream writable;

  readonly attribute unsigned long maxDatagramSize;
  attribute double? incomingMaxAge;
  attribute double? outgoingMaxAge;
  attribute long incomingHighWaterMark;
  attribute long outgoingHighWaterMark;
};
```

Example

```
EXAMPLE 1
async function sendDatagrams(url, datagrams) {
  const wt = new WebTransport(url);
  const writer = wt.datagrams.writable.getWriter();
  for (const datagram of datagrams) {
    await writer.ready;
    writer.write(datagram).catch(() => {});
  }
}
```

Connection Statistics
https://w3c.github.io/webtransport/

```
dictionary WebTransportStats {
  DOMHighResTimeStamp timestamp;
  unsigned long long bytesSent;
  unsigned long long packetsSent;
  unsigned long long packetsLost;
  unsigned long numOutgoingStreamsCreated;
  unsigned long numIncomingStreamsCreated;
  unsigned long long bytesReceived;
  unsigned long long packetsReceived;
  DOMHighResTimeStamp smoothedRtt;
  DOMHighResTimeStamp rttVariation;
  DOMHighResTimeStamp minRtt;
  WebTransportDatagramStats datagrams;
};
```

```
dictionary WebTransportDatagramStats {
  DOMHighResTimeStamp timestamp;
  unsigned long long expiredOutgoing;
  unsigned long long droppedIncoming;
  unsigned long long lostOutgoing;
};
```

- Missing CC info
  - ecn, ACK info
  - latest_rtt
  - pkt_departure/pkt_arrival

Stream Stats

```
dictionary WebTransportSendStreamStats {
  DOMHighResTimeStamp timestamp;
  unsigned long long bytesWritten;
  unsigned long long bytesSent;
  unsigned long long bytesAcknowledged;
};
```

```
dictionary WebTransportReceiveStreamStats {
  DOMHighResTimeStamp timestamp;
  unsigned long long bytesReceived;
  unsigned long long bytesRead;
};
```

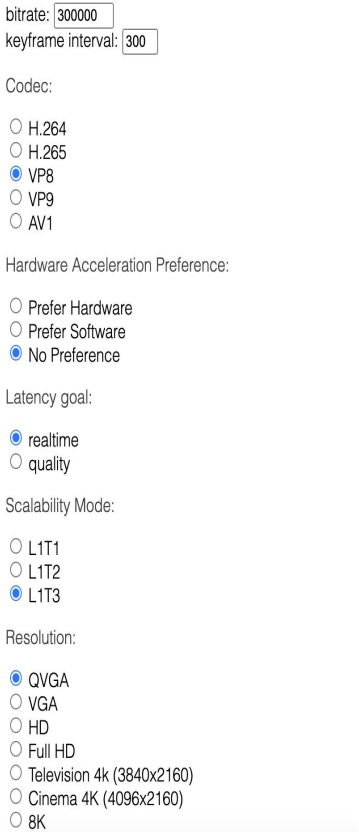
Demos

1. Basic echo https://webrtc.internaut.com/wt/ (also https://webtransport.day/)
2. File upload fiddle https://jsfiddle.net/jib1/z965juL7/
3. WARP - https://quic.video/demo/
    a. GitHub repo: https://github.com/kixelated/warp-demo

4: Integrating WebCodecs and WebTransport

1. This demo is an extension of a WebCodecs in Workers sample, which encodes and decodes video in a WHATWG Streams pipeline.
   a. Live site: https://webrtc.internaut.com/wc/wcWorker/
   b. Github repo: https://github.com/aboba/wc-demo/
2. This demo adds network transport to the sending and receiving pipelines, bouncing encoded frames off an echo server in the cloud.
   a. Live site: https://webrtc.internaut.com/wc/wtSender2/
   b. Live site with BYOB reads (Chrome Canary): https://webrtc.internaut.com/wc/wtSender4/
   c. GitHub repo: https://github.com/aboba/wt-demo

Parameters to Select



- Bitrate: “Average Target Bitrate” target provided to the encoder.
  - Actual bandwidth consumption is typically lower.
- Keyframe interval: number of frames between each keyframe.
- Codec: VP8, VP9, H.264 or AV1
  - Some oddities noted with VP9 (large frame size with “realtime”)
  - AV1 most solid on MacOS
  - H.265 not supported currently.
- Hardware Acceleration Preference: hw accelerated versus software encode/decode. Hw acceleration often not available.
- Latency goal: “quality” produces smaller frame sizes, but takes (marginally) longer than “realtime”.
- Scalability mode: how many temporal layers to use. Enables differential protection for the base layer.
- Resolution: reflected in getUserMedia constraints. If your camera doesn’t support the requested resolution, window will be blacked out.

High Level Observations

- Video quality
  - Quality dependent on device and camera.
  - Good quality possible with desktop/high quality notebook and appropriate settings.
  - Full-HD video (talking head) consumes < 500 kbps.
- CPU Utilization
  - Higher resolutions (e.g. full-HD) or complex codecs can result in high CPU utilization.
- Resilience
  - QUIC reliable transport + temporal scalability provides good resilience.
    - QUIC stream/frame transport provides retransmission.
    - Temporal scalability enables partial reliability.
      - Non-base layer frames can be considered discardable.

High Level Observations (cont'd)

- Latency
  - Observed glass-glass latency ***considerably*** higher than measured frame RTT.
  - P-frames are typically small (a few packets) and exhibit low frame RTT.
  - I-frames are ***much*** larger (10X or more) and exhibit frame RTT multiple times higher (though not always).
    - Effect most pronounced with high GoP sizes (only a few I-frames per experiment)
    - Effect seen even under conditions of low bandwidth utilization and low loss.
    - Potentially due to "app limited" congestion window?

Example

- AV1 @full-hd
  - Target average bitrate = 1 Mbps, GoP = 300
  - P-frame RTT ~ 100 ms with low jitter/no frame loss
  - For large frames, frame RTT multiple times higher
  - Glass-glass latency ~ 630 ms
  - Much lower glass-glass latency with “no network” sandbox

Local Video