W3C WebRTC WG Meeting

Stockholm, Sweden
June 20, 2018

Chairs: Stefan Hakansson

Bernard Aboba

Harald Alvestrand

W3C WG IPR Policy

- This group abides by the W3C Patent Policy https://www.w3.org/Consortium/Patent-Policy/
- Only people and companies listed at https://www.w3.org/2004/01/pp-impl/47318/status are allowed to make substantive contributions to the WebRTC specs

Welcome!

- Welcome to the face-to-face meeting of the W3C WebRTC WG!
- During this meeting, we hope to:
  - Make progress on WebRTC NV
  - Discuss issues relating to WebRTC 1.0 such as testing, identity and statistics.

About this Meeting

Information on the meeting:

- Meeting info:
  - https://www.w3.org/2011/04/webrtc/wiki/June_19-20_2018
- Link to latest drafts:
  - https://w3c.github.io/mediacapture-main/
  - https://w3c.github.io/webrtc-pc/
  - https://w3c.github.io/mediacapture-screen-share/
  - https://w3c.github.io/webrtc-stats/
- Link to Slides has been published on WG wiki
- Scribe? IRC http://irc.w3.org/ Channel: #webrtc
- The meeting is being recorded.
- WebEx info here

Proposed Agenda for June 20

9:00 AM - 9:30AM: Transports (Peter Thatcher)

a. Potential data channel transports: SCTP, QUIC, RTP
b. Potential media transports: RTP, QUIC

9:30 AM - 10 AM Use Case Reprise - includes API level discussion (Peter Thatcher)

10:00 AM - 11 AM QUIC (Peter Thatcher)
Reference: https://w3c.github.io/webrtc-quic/

11 AM - noon: SCTP, Data Channel and Streams (Lennart Grahl)
Noon - 12:30 PM: Scalable Video Coding (Sergio and Bernard)

12:30 PM - 1 PM: E2E Security Drill-down (~~Goran~~, Youenn & Dr. Alex)

1PM - 1:30 PM: Lunch break
1:30 PM - 2 PM: Protocol dependencies (Stefan)

2 PM - 2:30 PM: WebRTC-Stats (Varun)
2:30 PM - 3:00 PM: WebRTC NV Repriese (Bernard)
3:00 PM - 3:30 PM: Worker reprise (Peter Thatcher)
3:30 PM - 4:00 PM: RtpTransport (Bernard Aboba)
4 PM - 5 PM Wrapup and next steps

Proposed update to Agenda June 20

On request from some...

9:00 AM - 9:30 AM: E2E Security Drill-down (~~Goran~~, Youenn & Dr. Alex)

9:30 AM - 10:00AM: NV Transports (Peter Thatcher)

a. Potential data channel transports: SCTP, QUIC, RTP
b. Potential media transports: RTP, QUIC

10 AM - 10:30 AM Use Case Reprise - includes API level discussion (Peter Thatcher)

10:30 AM - 11:30 AM QUIC (Peter Thatcher)
Reference: https://w3c.github.io/webrtc-quic/
11:30 - noon AM: Scalable Video Coding (Sergio and Bernard)

noon - 1pm: SCTP, Data Channel and Streams (Lennart Grahl)

1PM - 1:30 PM: Lunch break
1:30 PM - 2 PM: Protocol dependencies (Stefan)

2 PM - 2:30 PM: WebRTC-Stats (Varun)

2:30 PM - 3:00 PM: Worker reprise (Peter Thatcher)
3:00 PM - 3:30 PM: ORTC Object Model Overview (Bernard and Peter)
4 PM - 5 PM Wrapup and next steps

Private exchange with WebRTC

Context

- User might want to use a service but keep control of his/her data
- Application level
    - MediaStream/Text content
- Network level
    - Direct connection
    - Intermediated connection (SFUs)

Trust model

- General case: Web App is untrusted
  - Delivery provider is not trusted
- Intermediate case: Web App is partially trusted
  - Embedded « untrusted » iframe in a « trusted » iframe for instance
- Limited case: Web App is trusted
  - Delivery provider is not trusted

Web App is trusted

- Web App can get access to user content
- Web App can have access to encryption/decryption keys
- Need for web app to pass keys to WebRTC
- Implemented as part of Symphony solution

Web App is partially trusted

- Doctor website embeds a WebRTC provider iframe
- User content should be protected for the iframe
  - Isolated streams/encrypted text
  - Doctor website to express this constraint on the WebRTC provider
- WebRTC provider iframe cannot get access to keys
  - Key identifiers might be fine

Web App is untrusted

- Audio/video content should be isolated
    - Isolated text exchange?
    - No funny hat anymore
- Web App cannot get access to keys
    - key identifiers are fine
- key identifiers <-> key values mapping out of the web app control
    - EME model / IDP like model
- Need for web app to get/pass key identifiers to WebRTC
    - Web app cannot do its own encryption

Conclusion

- Potentially limited API surface at WebRTC level
  - This is not the difficult part
- Might be able to cover some cases with partial trust
- If no trust at all, probably need to piggy back on IDP
- Delegating crypto to web app is not always possible

WebRTC NV Transports

6/2018 f2f

Do apps want to send audio, video, or data?

Do apps want to send audio video or data?

Yes

Do apps want to send over RTP, SCTP, or QUIC?

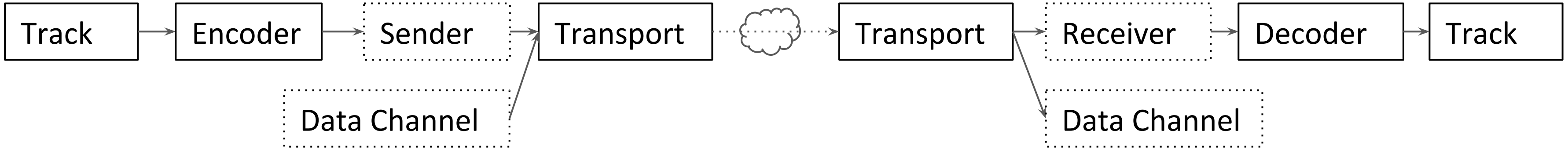
Do apps want to send over RTP, SCTP, or QUIC?

Yes

What WebRTC has right now

	RTP	SCTP	QUIC
Audio	Yes		
Video	Yes		
Data		Yes	Yes (ORTC)

Unified audio/video/data



Unified audio/video data

If you're doing audio, video, and data (games, VR, conference share, remote control):

- you want them all in the same congestion control context
- with the same protocol,
- with a protocol is easy to terminate on a server, hardware device, mobile app, etc

We could do all of them

	RTP	SCTP	QUIC
Audio	Done	Add Encoders	Add Encoders + QuicTransport
Video	Done		
Data	Add RtpTransport	Done	Add QuicTransport

Apps can choose which protocols are easiest/best for its use case

DtlsTransport

Just like in ORTC:

- Uses IceTransport
- **Provides crypto** for SctpTransport
- Provides crypto handshake for RtpTransport
- Takes a local crypto cert and remote cert fingerprint

SctpTransport

Just like in ORTC:

- Uses DtlsTransport for crypto
- **Provides congestion control, reliability**, message framing for data channels
- Implements WebRTC 1.0 data channel in-band signaling on top of SCTP
- If encoders/decoders are separate, apps can send media over SCTP as well.
- Could use some improvements, especially around backpressure. Lennart will be covering that.

QuicTransport

Just like in ORTC and extension spec:

- **Provides crypto and congestion control and reliability.**
- Thin layer on top of QUIC protocol:
  - QuicTranport == QUIC Connection
  - QuicStream == QUIC stream
- Data channels and media senders/receivers can be built on top (more on that later)
- More info about QuicTransport later

RtpTransport

Not like ORTC! ORTC doesn't provide RTP data channels, nor low-level control (e2ee, etc)

- **Provides crypto and congestion control (and reliability?)**
  - Crypto via SRTP
  - Congestion control via transport-cc?
  - Reliability via RTX + transport-cc?
- Apps send/receive RtpPackets, fully controlling PT, SSRC, seqnum, timestamp, payload, header extensions
- Which means it can send data by making a "data payload"
- And can add e2ee by double-encrypting the payload
- And can decide how to packetize h264 (so many ways....)

RtpTransport

```
interface RtpTransport {
  sendRtpPacket(RtpPacket packet);
  sendRtcpPacket(RtcpPacket packet);
  attribute eventhandler onrtppacket;
  attribute eventhandler onrtcppacket;
}

dictionary RtpPacket {
 unsigned octet payloadType;
 unsigned int sequenceNumber;
 unsigned long ssrc;
 array<RtpHeaderExtension> header_extensions;
 array<unsigned long> csrcs;
 bytes payload;
}
```

Questions for the WG

- Do we want to support audio/video/data over rtp/sctp/quic?

WebRTC NV Use Cases Round 2

Needs separate encoders/transport:

- e2e encryption (maybe...)
- BYO jitter buffer, FEC, codec
- Media over QUIC/SCTP or Data over RTP

Needs unified audio/video/data:

- Realtime text + audio
- Browser<->devices
- Client <-> server games
- VR
- Remote control
- Live-ish video to/from server

Other stuff

- Make data channels be less aggressive against HTTP? For background data
- More “stats” :), especially RTTs and BWEs (from SCTP/QUIC?)

WebRTC NV Use Cases Round 2

If we don't split encoders from transports, how will we:

1. Do e2e encyption?
2. Do media over QUIC?
3. Do data over RTP?

Or do we just say no to these requirements?

Once possible course of action

Many of our doubts about a low-level API revolve around two things:

1. Libraries for doing parts of the stack won't exist
2. They won't be performant

So... what if someone goes and does a proof of concept combination of low-level browser API + library on top that proves it can be performant and provides the library on top, or finds that it really isn't feasible?

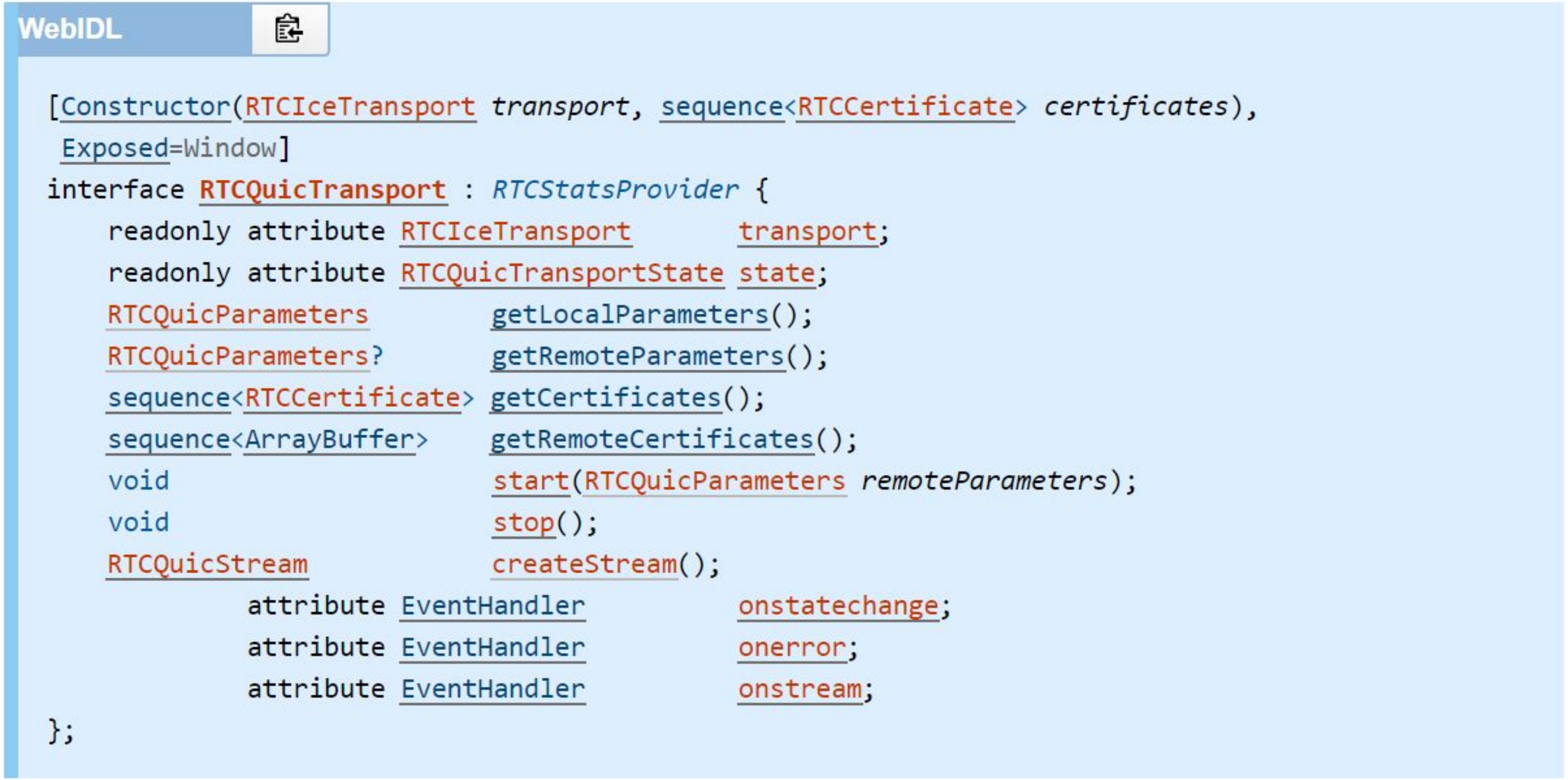
QUIC
(Peter Thatcher)

https://w3c.github.io/webrtc-quic/

Thin layer on top of QUIC protocol; App controls the rest

- QuicTransport == QUIC connection
- QuicStream == QUIC stream
- Reliable/ordered "channels" with a simple message framing in a single stream
- Unreliable/unordered "channels" with one stream per message
- Media with a serialization of encoded frames from an Encoder
- The tricky part: back pressure

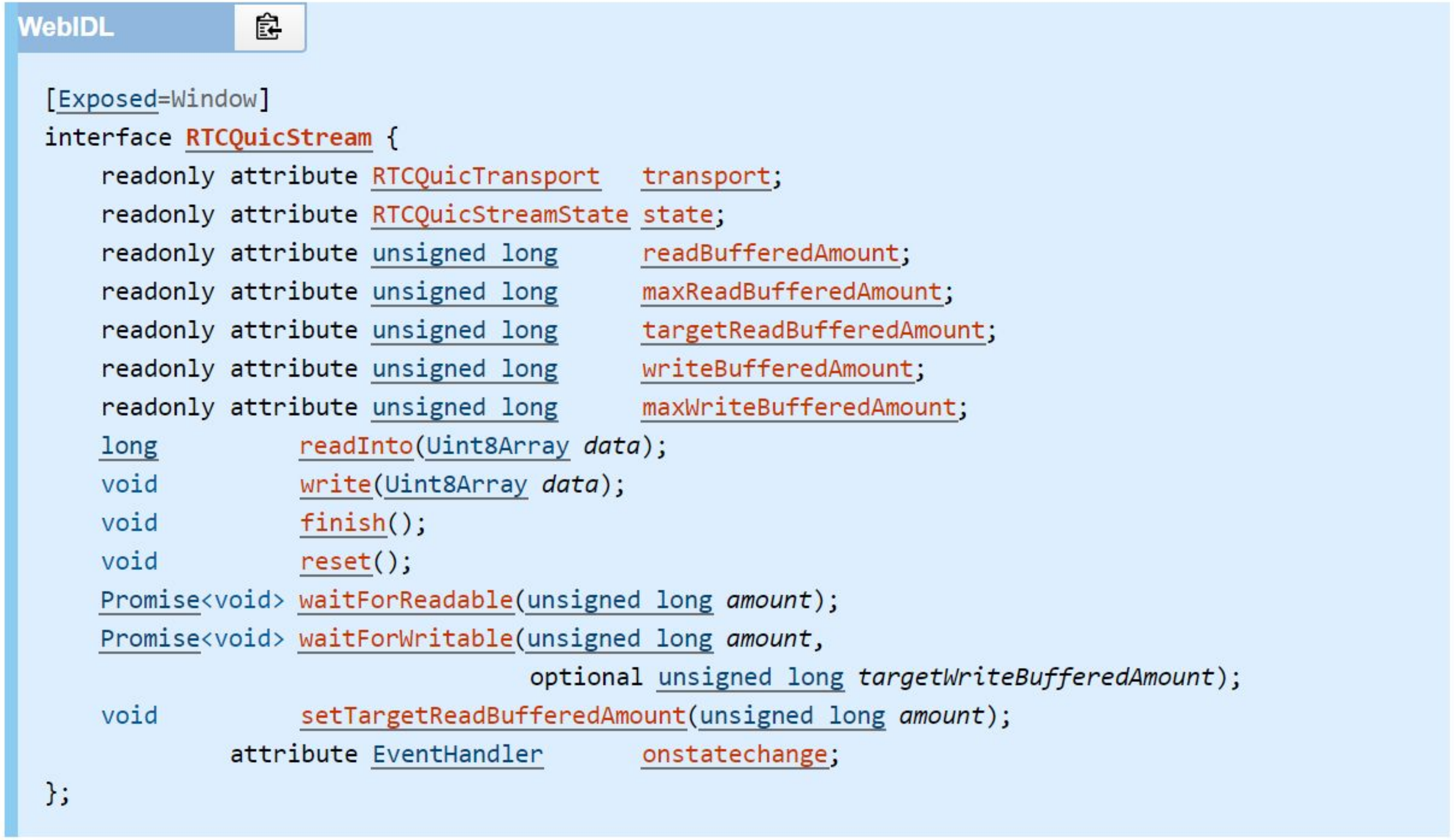
RTCQuicTransport Interface


WebIDL

```
[Constructor(RTCIceTransport transport, sequence<RTCCertificate> certificates),
 Exposed=Window]
interface RTCQuicTransport : RTCStatsProvider {
    readonly attribute RTCIceTransport        transport;
    readonly attribute RTCQuicTransportState state;
    RTCQuicParameters          getLocalParameters();
    RTCQuicParameters?         getRemoteParameters();
    sequence<RTCCertificate> getCertificates();
    sequence<ArrayBuffer>    getRemoteCertificates();
    void                       start(RTCQuicParameters remoteParameters);
    void                       stop();
    RTCQuicStream              createStream();
             attribute EventHandler          onstatechange;
             attribute EventHandler          onerror;
             attribute EventHandler          onstream;
};
```

Source: https://w3c.github.io/webrtc-quic/#quic-transport*

RTCQuicStream Interface


WebIDL

```
[Exposed=Window]
interface RTCQuicStream {
    readonly attribute RTCQuicTransport    transport;
    readonly attribute RTCQuicStreamState state;
    readonly attribute unsigned long       readBufferedAmount;
    readonly attribute unsigned long       maxReadBufferedAmount;
    readonly attribute unsigned long       targetReadBufferedAmount;
    readonly attribute unsigned long       writeBufferedAmount;
    readonly attribute unsigned long       maxWriteBufferedAmount;
    long          readInto(Uint8Array data);
    void          write(Uint8Array data);
    void          finish();
    void          reset();
    Promise<void> waitForReadable(unsigned long amount);
    Promise<void> waitForWritable(unsigned long amount,
                                  optional unsigned long targetWriteBufferedAmount);
    void          setTargetReadBufferedAmount(unsigned long amount);
             attribute EventHandler        onstatechange;
};
```

Source: https://w3c.github.io/webrtc-quic/#quicstream*

Questions for the WG

- Should the WG work on QuicTransport for reliable use cases?
- Are we OK with the low-level approach?
- How shall we deal with back pressure and buffering? (see more slides)

What about worklets?

As you will see, there's a lot of complexity around buffering and backpressure on the receive side, all because JS can pause and not keep up with incoming data.

But what if we used a worklet on the received side?

```
// Called synchronously off main thread when data comes in.
// If it takes along time or doesn't call consume(), back pressure is applied.
QuicStream.setReceiver(worklet);
QuicStream.consume(count);  // Called by receiver to consume or not
```

Similarly on the send side?

```
// Called synchronously off main thread when more can be written
QuicStream.setSender(worklet);
```

`Could be a lot more simple and performant!`

Alternative

1. **Define DataChannel on top of QUIC in browser as well**
    a. Bernard and I did the work to define this and it's surprisingly hard to get all of the edge cases right. We decided it wasn't worth it.
    b. Plus, the lower-level API is better in many ways (buffering, overhead)
2. Define MediaSender on top of QUIC in browser as well
    a. Have to define what parts of an EncodedFrame are sent, how it is serialized, how it is mapped to QUIC streams, when those streams should be cancelled, how stats and key frame requests are sent. Etc, etc.
    b. And now this is well in the direction of the IETF people interested in defining media over QUIC, and it will take a while to specify it
    c. And once we do, we'd be stuck with that format and couldn't experiment with other formats

I think we're much more future proof and get benefits sooner by letting apps/libraries do this.

Buffering and back-pressure

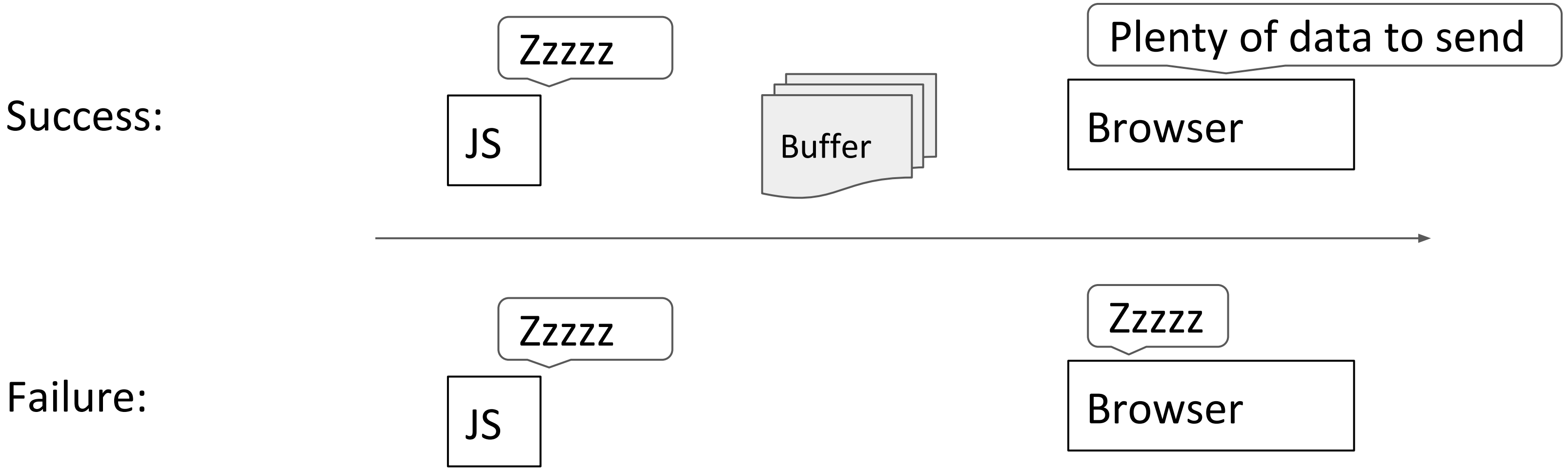
Buffering and back-pressure aren't simple. Because Javascript. Can freeze up.

We don't want throughput to suffer.

But we don't want to buffer a lot.

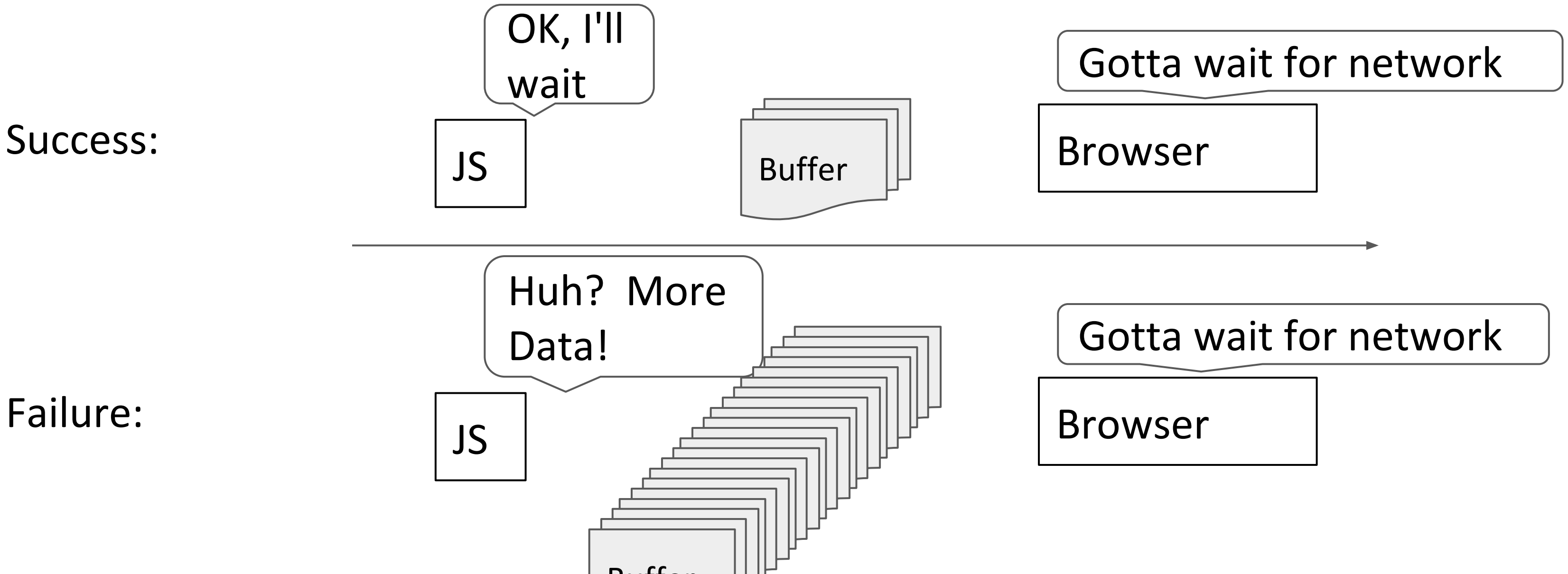
Properties of an ideal buffering/back-pressure solution

1. If the sending JS freezes up, the browser keeps sending what is buffered and has enough to keep throughput high. Scenario to avoid: the send buffer dries up and nothing is sent for a while, throughput suffering.



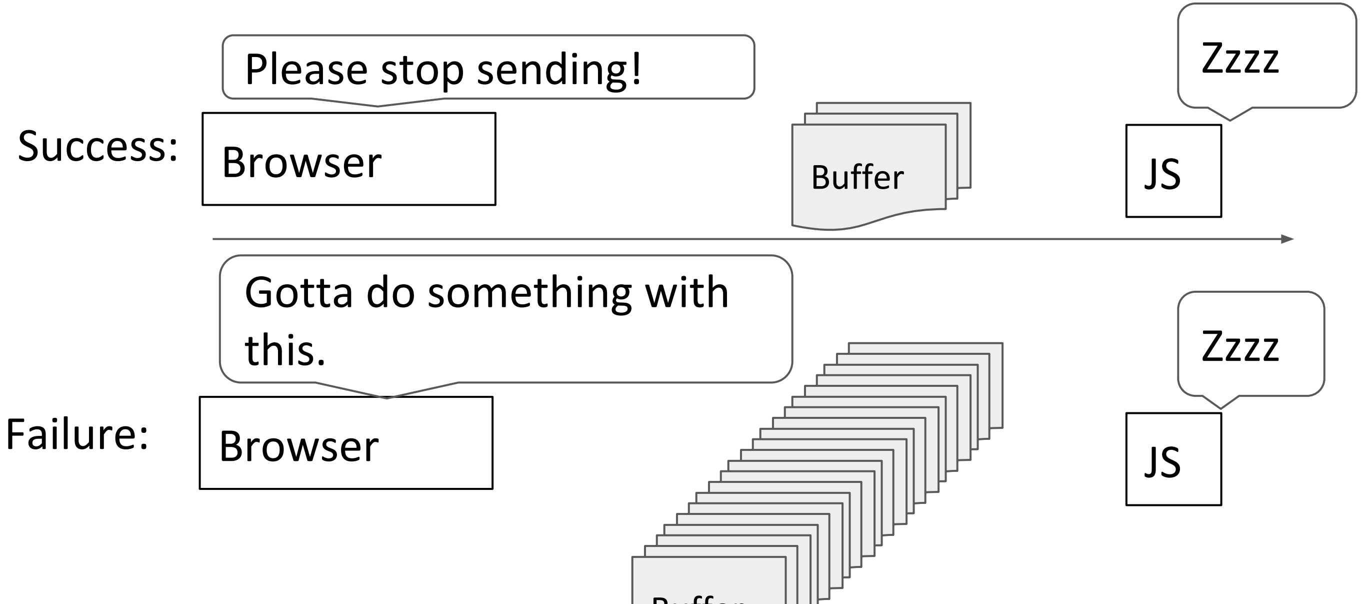
Properties of an ideal buffering/back-pressure solution

2. If the sending JS has a very large amount, the browser applies back pressure to the JS and doesn't buffer huge amounts. Scenario to avoid: a huge amount gets buffered.



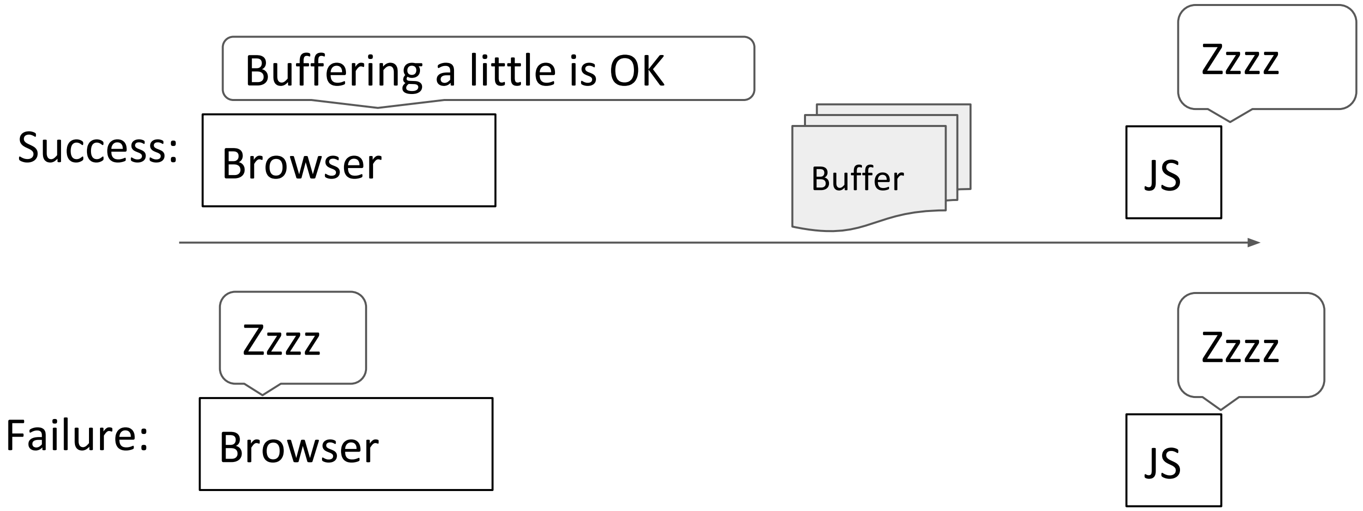
Properties of an ideal buffering/back-pressure solution

3. If the receiving JS can't process data as quickly as it comes across the network, the browser applies back-pressure across the network and the send side stops sending. Scenario to avoid: the receive buffer grows unbounded.



Properties of an ideal buffering/back-pressure solution

4. If the receiving JS freezes up, the browser keeps receiving and buffers what is received enough to keep throughput high. Scenario to avoid: the receive buffer is full or non-existent, and nothing is received for a while, throughput suffering.



Current solution (alternatives can be considered)

As an app, you:

1. Set the amount you're willing to buffer
2. Wait for space in the buffer before reading/writing

And you get:

- Back pressure automatically
- Throughput despite pauses (if buffer is big enough)

Current solution (alternatives can be considered)

Or if you're sending lots of small, regular messages where reliability doesn't matter and you don't want buffering, all you have to do is:

- Create a stream.
- Call write()
- Call finish()

That's it. If it succeeds, great. If not, oh well.

Current solution (in detail)

1. App specifies a certain amount to buffer *on top of* the send/receive windows inside of QUIC
2. Browser can push back on send JS by making write() fail if there's no room in the buffer (Avoiding problem of DataChannel.send buffering indefinitely).
3. If send JS doesn't call write() because it's frozen, browser sends from the buffer (Avoiding the problem of reduced throughput from JS pausing)
4. Receive JS can push back on browser by not calling read() (Avoiding problem of DataChannel.ondata firing even if he JS can't keep up).
5. If receive JS doesn't call read() because it's frozen, browser receives into the buffer. (Avoiding the problem of reduced throughput from JS pausing)
6. JS can know when to call read() or write() without polling by waiting for a certain amount to be available in the buffer or QUIC send/recv window (so things keep flowing even if buffer size is 0)

Alternative solution

Each QUIC stream has:

- A WritableStream of bytes
- A ReadableStream of bytes

(see https://github.com/lgrahl/ortc-streams-demo)

So waiting and buffering is (cross your fingers) taken care of by WritableStream and ReadableStream.

Note that one could implement these streams *on top* of the current solution.
And the apparatus of WritableStream might be a too much overhead for many small streams.

TBD (more questions)

1. What should the default buffering be?
   - Infinite? Then we aren't applying back pressure to the sender by default
   - 0? Then throughput might not be as good by default.
   - N? How do we pick N?
2. Can we make QuicTransport work without IceTransport?
   - Easier for server endpoints
   - Like a websocket, but with QUIC and more control
   - QUIC handshake and PING frames act as consent checks.
   - Any reason not to?

SCTP, Data Channel, Streams

Evolutionary Steps

(Lennart Grahl)

Tackling Use Case Requirements

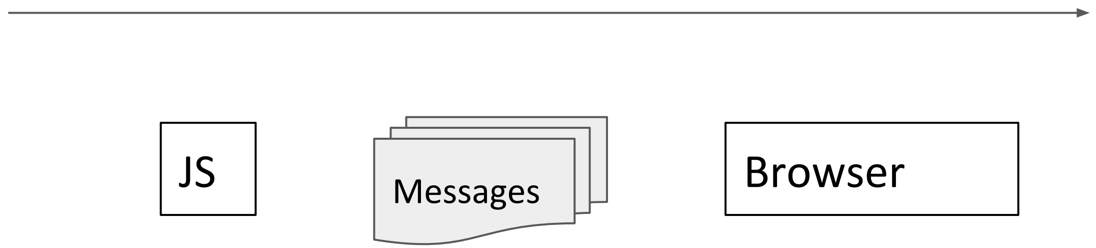
- Backpressure on both sides (flow control)
- Control over retransmissions
- Less noise on the wire
- Congestion control that works well with audio/video
- (Reduce initial RTTs)

Status Quo: Sender Side Buffering/Backpressure

1. Set `bufferedAmountLowThreshold` to *some carefully chosen low water mark*.
2. Send messages until it reaches *some carefully chosen high water mark*, then pause.
3. Continue sending (2.) once the `bufferedamountlow` event fires.

It works… but only under certain circumstances.

Success!

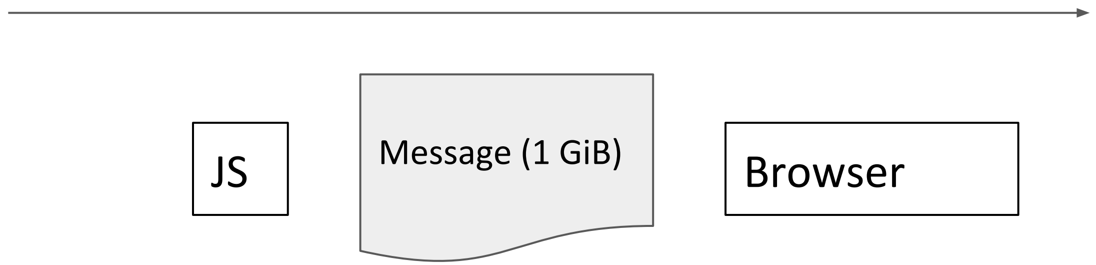


Status Quo: Sender Side Buffering/Backpressure

1. Set `bufferedAmountLowThreshold` to *some carefully chosen low water mark*.
2. Send messages until it reaches *some carefully chosen high water mark*, then pause.
3. Continue sending (2.) once the `bufferedamountlow` event fires.

Large messages still blow up the buffer.

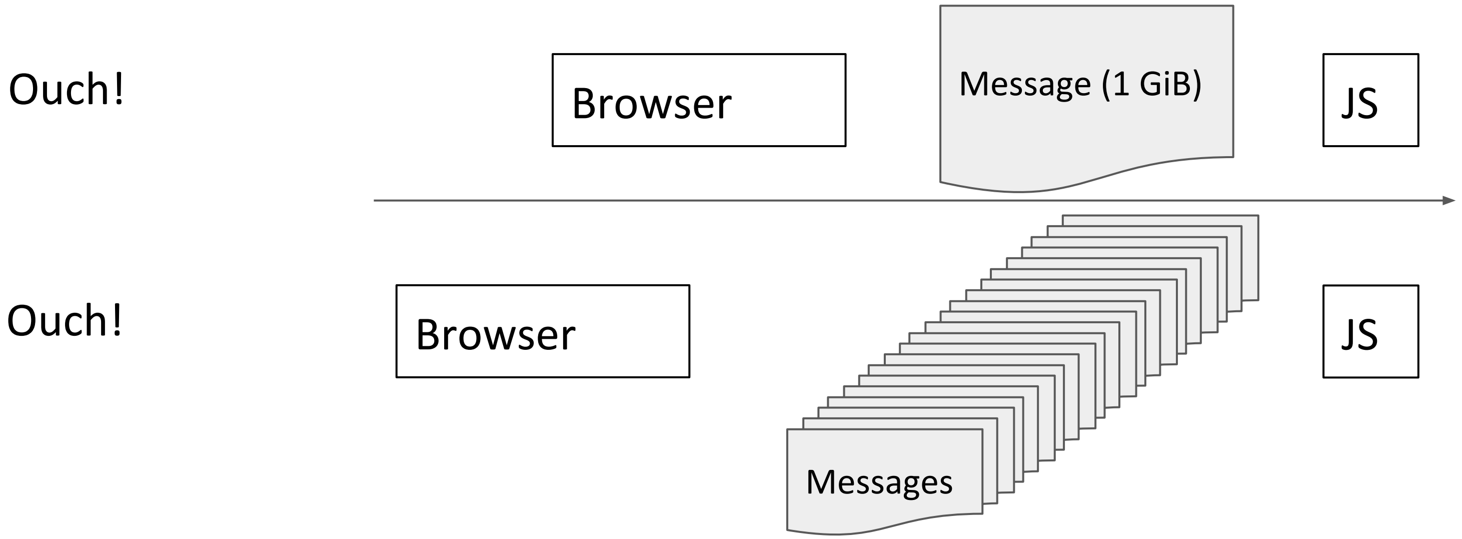
Ouch!



Status Quo: Receiver Side Buffering/Backpressure

1. Just hope you've installed enough RAM.

The receiver can't say *no*.



Fixing Buffering/Backpressure: What Does SCTP Offer?

SCTP is message-oriented:

- **Don’t:** Break up a large file into small messages and add a protocol on top to mark the end of a file.
- **Do:** Break up a large file into small chunks and leverage the message-oriented principle to mark the end of a file.

How:

- Sending: Explicit EOR mode to send a large message in small chunks.
- Receiving: Pause receiving on the SCTP association.*

Fixing Buffering/Backpressure: Quick & Dirty

Sending:

- Send data by calling `send({ data: chunk, eor: true|false });`

Receiving:

- Add a new `binaryType`: `‘arraybuffer-chunked’.`
- Hand out partial messages in the `message` event and explicitly mark the *end of record* with an additional attribute.
- Pause receiving is tricky.

IMHO a brittle API.

Fixing Buffering/Backpressure: Streams

Sending:

- New method `createWritableStream(): WritableStream`
- Add chunks by calling `write(chunk)` on the associated writer.
- `close()` the writer to mark the end of the message.

Receiving:

- Add a new `binaryType`: `‘stream’.`
- Hand out a `ReadableStream` instance as part of the `message` event with the first chunk received.
- Pause receiving is still tricky but could be added later without API changes.

More details in webrtc-pc#1732.

Fixing Buffering/Backpressure: Streams

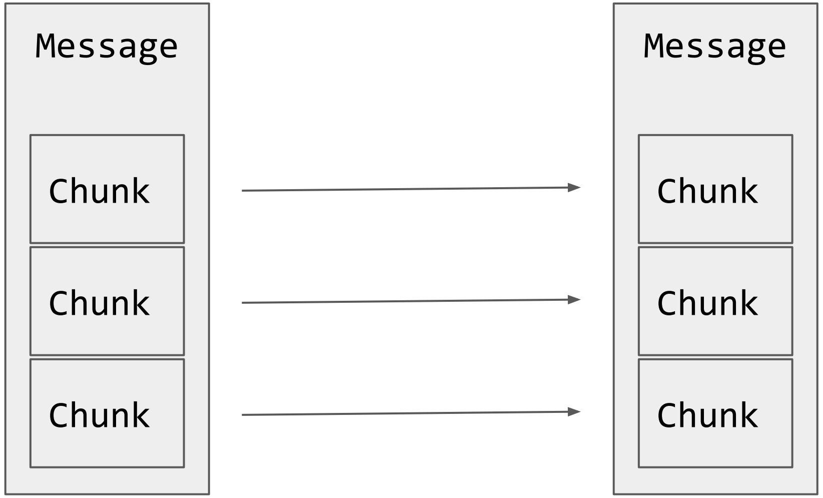
```
createWritableStream():
Writable Stream

   writer.write(chunk)

   writer.write(chunk)

   writer.write(chunk)

   writer.close()
```



```
message (event):
Readable Stream

reader.read(view)

reader.read(view)

reader.read(view)

reader.closed
```

```
httpResponseStream.pipeTo(dataChannel.createWritableStream())
```

Fixing Buffering/Backpressure: Streams

Advantages:

- Platform consistency
- Low level and high level
- `TransformStream` (unzip, double crypto, ...)
- `pipeTo` optimisation, BYOB, ...

Questions for the WG:

- Yay or nay?
- If yay: Should it be in NV or should we also backport it to 1.0?
- If yay, can we say *goodbye* to `maxMessageSize`?

More Control Knobs

Expose more ways to control data channel behaviour:

- Set the retransmission timeout values (initial/min/max)
- Allow to turn heartbeat on/off or set the interval
- Set the maximum amount of retransmissions until the SCTP association aborts

Ideally not controlled by `RTCDataChannelParameters` because it’s a dictionary that fails silently.

```
new RTCDataChannel(sctpTransport,
{
  maxRetransmits: 0,
  label: ‘call me ordered UDP’,
});
```

$\longrightarrow$

```
sctpTransport.newChannel()
  .withMaxRetransmits(0)
  .withLabel(‘call me ordered
UDP’)
  .create();
```

Improving SCTP For Our Requirements

- Message interleaving on channels: Specified (RFC8260) but still requires some work in usrsctp. Should we make this mandatory?
- Signal backpressure/flow control on a per-channel basis: No ongoing work so far.
- Drop one initial RTT: Paper is done, we need to explicitly raise our interest to get this specified.
- More A/V friendly congestion control: BBR can be added, at least to usrsctp. (Has been simulated, expect a paper.)

WebRTC NV & SCTP

- Copy the ORTC model
- A few API tweaks here and there
- No additional SCTP send/receive API on a lower level required
- Do we want to keep the `RTCDataTransport` abstraction?

Questions for the WG

- Thoughts on more control knobs?
- Opinions on SCTP extensions?
- NV plan sounds good?
- (Off-topic: Is the builder pattern a good replacement for controlling explicit behaviour which was formerly driven by dictionaries?)

Scalable Video Coding (Sergio and Bernard)

SVC Use cases

Simulcast and SVC use cases are very similar (if not identical): **content adaptation without transcoding**

Adaptation will happen as a combination and/or trade off of the following items:

- Bitrate
- Image size
- FPS
- Quality
- Decoding complexity

Typically you would always love to have the best quality, with biggest image size and most fps, but most of the times you have will be restricted to do that and have to adapt video stream to the match certain limits.

Also, another interesting use case would be for **increasing reliability against packet losses** by applying different protection levels to each layer and maximizing the probability that at least the base layer is received.

What do Application developers need?

- Ability to enable/disable temporal/spatial scalability (or simulcast/temporal scalability).
- Ability to set maximum framerate or bitrate.
- Set desired targets on image size
- Set desired granularity for adaptation (number of layers/steps or and relative difference between them in terms of bitrate/fps/size).
- Set priority between encodings/layers to control how they prefer to degrade quality/movement and/or split bitrate between.
- Little interest in dependencies between layers, it is a codec thing.

What do SFU developers need?

- Well understood and consistent browser algorithm for dropping simulcast or temporal/spatial layers.
    - In WebRTC 1.0, not documented how (or if) the algorithm is affected by degradationPreference, priority or DSCP markings
- Way to know when the browser stops sending a layer
    - Without this, SFU cannot initially distinguish a layer drop from burst loss.

How should we deliver SVC support?

Given the big differences on the SVC codecs it is not easy to come up with a common parameter/apis that will match all the functionalities provided by all present and future codecs in an easy way.

The proposed way forward for enabling SVC support is:

- WebRTC 1.0 : Basic functionality based on encoding parameters.
- WebRTC NV : Full API based on Encoders objects.

WebRTC 1.0

Background info:

- ORTC discussion on Simulcast/SVC capabilities https://github.com/w3c/ortc/issues/837
- Chrome's simulcast mode for screenshare https://groups.google.com/forum/#!msg/discuss-webrtc/hckBc35579w/as-1EDKIAwAJ

We need a way to signal the SVC/Simulcast support on the browser and a basic on/off switch. Some alternatives:

- Use getCapabilities to signal Simulcast/SVC capabilities and temporal/spatial hard limits?
- Use addTransceiver/setParameters for enabling SVC and setting maximum number of temporal/spatial layers?
- Use a "conference mode " flag in SDP as Chrome's `a=x-google-flag:conference` and let the browser decide best settings?

Do we need to allow configuring image sizes, bitrates, fps and degradation of each layer or is it too complex?

WebRTC NV

Seems that we will split the RTPSender into different components so we will have a VideoEncoder interface (yay!).

We can just follow the same approach for WebRTC 1.0 and have a generic method(s) for setting the svc settings for all codecs:

```
dictionary SVCEncodingParamers {
      // We know that defining a generic SVC parameters dictionary for all codecs is complex!
}
interface SVCVideoEncoder: VideoEncoder {
  setSVCEcodingParameters(SVCEncodingParamers)
};
```

But, we don’t have any requirement to provide a single interface for all codecs, so we could specialice each SVC encoder and provide specific APIs for each codec that matches exactly its features:

```
interface AV1Encoder: SVCVideoEncoder {
  //Specific AV1 SVC control API
};
```

Protocol Dependencies (Bernard)
- IETF dependencies?

1. E2E Security
    - Potential dependency on IETF PERC documents (only if implemented in browser)
    - Potential dependency on frame-marking
    - Potential dependency on FEC/RTX implementation
2. Changes to ICE to enable requirements not covered by existing specifications
    - Separate slide
3. QUIC things
    - Separate slide
4. ~~SDP support for QUIC (if we want QUIC support in WebRTC 1.0)~~
5. SCTP improvements - backpressure per channel, drop one initial RTT
    - Separate slide
6. Anything needed for SVC?

ICE

Changes needed to protocol if we go with FlexICE

- Freezing optional
- Candidate renomination

QUIC

- As-is
  - QUIC documents in progress (core, transport)
- New stuff
  - QUIC WG
    - Unreliability
  - On top of QUIC
    - data channel protocol: maybe leave to app
    - Media: maybe leave to app

SCTP

- Remove one initial RTT
- Per channel backpressure/flow control
- More A/V friendly congestion control (BBR?)

webrtc-stats

Varun Singh*
Harald Alvestrand

Spec Status

- Since TPAC 17, asked for CR
- Close to being processed as CR.
  - Dom has updates :)

- Are we done? → Maybe, perhaps not.

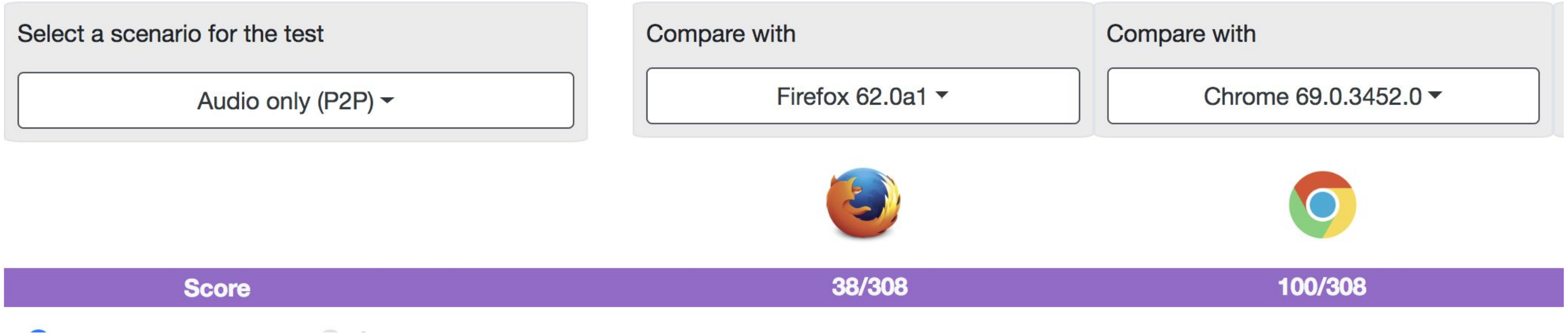
Experimental stats

- Propose a new stats by filing a new issue on github. Why is it useful for the application to be able to measure this? How is it measured?
- Create a PR that makes the change to the spec.
  - Ping @hta, @vr000m, they will review or assign someone will follow up with feedback and comments.
  - Fix review comments until LGTM.
- If it is an experimental metric, submit them to https://github.com/henbos/webrtc-provisional-stats
  - Thanks Henrik for kicking this off.

Implementation Status

A lot of implementation work is needed here

Audio-only



Select a scenario for the test

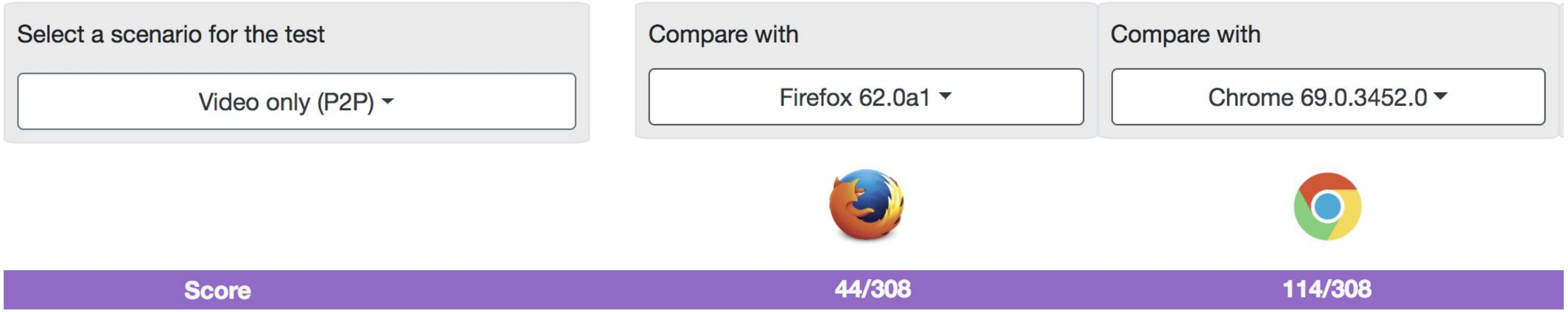
Audio only (P2P)

Compare with	Compare with
Firefox 62.0a1	Chrome 69.0.3452.0

Score	38/308	100/308

A lot of implementation
work is needed here

Video-only



Select a scenario for the test

Video only (P2P)

Compare with

Firefox 62.0a1

Compare with

Chrome 69.0.3452.0

Score	44/308	114/308

A lot of implementation work is needed here

Verify the implementations

- We built https://webrtc-stats.callstats.io/verify
- uses KITE and is Work In Progress
  - Currently only runs Audio-only and video-only Peer-to-peer cases
  - Will be extended to:
    - Simulcast
    - More codecs
    - More browsers
    - … new stuff

Adding Validations

- Use an experimental network, mock streams, etc
- Are the implementations reporting correctly
    - Example: 30 frames per second, is the API reporting 30 frames per second

- This is currently in works
    - Will have some initial results at TPAC’18 in October

webrtc-stats lessons learnt

- Implementations are incomplete
    a. Chrome does not report remote stats
    b. Most other browsers do not report much --- so cannot say much

- From a developer’s perspective, this is frustrating
    a. Makes adoption of some browsers difficult
    b. Developers prefer a certain browser because it has more telemetry

Some suggestions for NV

- Tie the stats with each component/objects in some way,
- **Thesis:** The metrics are developed and implemented simultaneously
-

Webrtc-stats lessons learnt

- For our use-case, callstats.io, getting access to stats from a peer-connection is complex.
    a. Explicitly ask developers to give us the peerconnection (our model)
    b. Override peerconnection (done by some others)

- Would like to explore better alternatives.

Alternatives

- Stats per Object
    - Awesome, but hard to correlate stats across objects
    - Discussion: some objects do not exist in pc, for example ice candidate paid.
- Stats Gatherer
    - Getters on objects which can be called by passing all the objects to a getStats…
        - There might be performance implications

Experimental Stats

How do we make it happen?

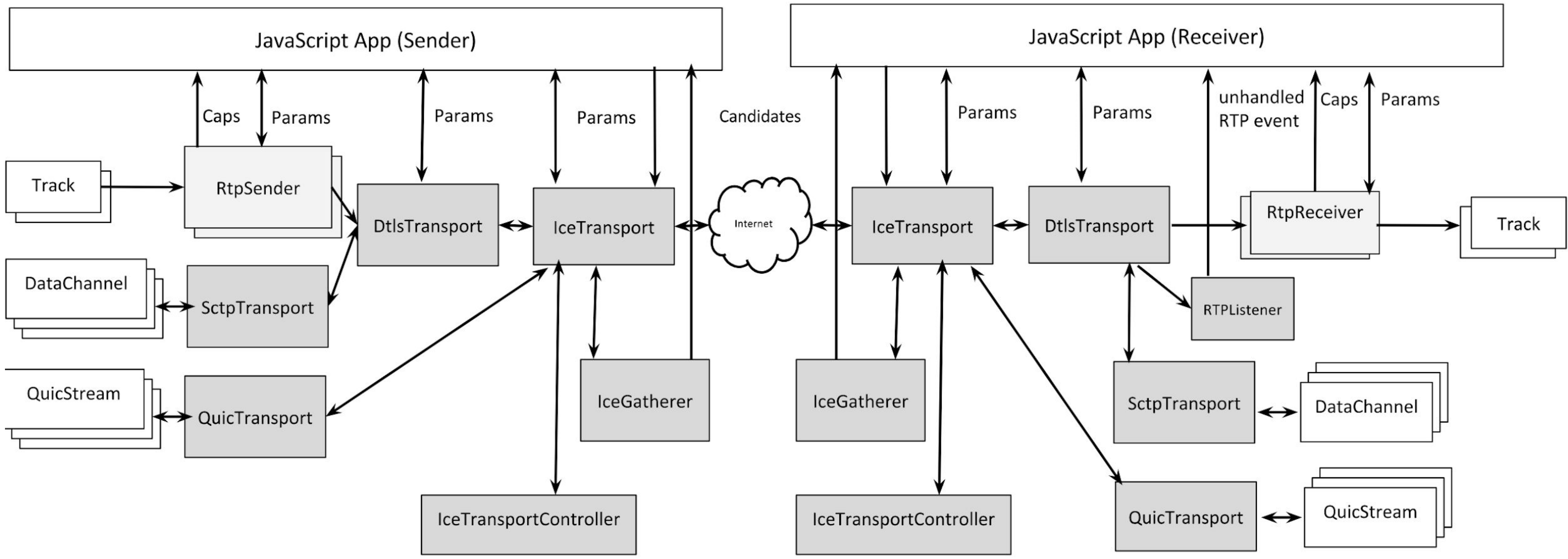
1. We have something to measure, but do not know how to measure it
2. We know this well defined metric, but we do not know how useful it is.

How do we enable these?

WebRTC NV: What We’ve Talked About

- In this meeting, we have talked about the following aspects:
  - Overall API level (B or perhaps C)
  - E2E security (discussion ongoing)
  - QUIC support (no consensus)
  - RtpTransport (no consensus)
  - Encoder/Transport separation (no consensus)
  - Data Channel objects: RTCDataChannel, SctpTransport (existing WebRTC 1.0 Model)
  - ICE (flexice extensions)... for use with QUIC?
  - Scalable video coding (SVC) support for WebRTC 1.0.

ORTC Object Model



Source: http://draft.ortc.org/

WebRTC NV: Missing Pieces

- The following objects have not been discussed:
  - Mandatory-to-implement in ORTC
    - RtpSender
    - RtpReceiver
    - DtlsTransport
    - IceTransport
    - RTCDataChannel
    - SctpTransport
  - Optional
    - RtpListener (RTP/RTCP packet routing)
    - IceTransportController (freezing)

WebRTC NV: Goals or Non-Goals

- Through our discussion so far, we have learned what we ***don't*** have consensus about.
- Question: Is the group interested in moving forward on aspects of WebRTC NV not discussed so far?
  - Yes!
  - Not now, but soon.
  - Never - remove WebRTC NV from the Charter!

Do we move forward? If so, how?

- Assuming there is interest, should the WebRTC WG move forward with:
  - Individual specs for each object
  - A single specification, based on ORTC, with all objects of interest

Wrapup and Next Steps (Chairs)

- 1.0 + family:
  - Create extension specs
    - Identity: Ekr+MT+Cullen
      - CR immediately
    - Simulcast/SVC: Harald, Sergio, Bernard
      - Can’t go CR immediately if we add SVC
    - Streams and extra knobs on datachannel: Lennart, Randell, Jan-Ivar
  - -stats: add ICE & datachannel stats: Varun - extension spec
  - Jan-Ivar + callstats volunteering to edit -screen-share

Wrapup and Next Steps (Chairs)

- NV:
    - Consensus to continue object clean up a la ORTC
    - Continue FlexICE: Peter
    - Big interest in Work* (workers, worklets)
        - Workers: Youenn + Tim (+ Harald)
        - Worklets: Youenn, Randell, Harald, Goran to investigate
            - (raw media goes here)
    - E2E security: Youenn to write up, Sergio and DrAlex to help out
    - Peter + Varun (+DrAlex) to continue push for QUIC...

Wrapup and Next Steps (Chairs)

- <content goes here>

Worklets Galore

Lots of high perf stuff is already using worklets or JS callbacks:
WebAudio (https://webaudio.github.io/web-audio-api/#AudioWorklet)

Animation worklets (https://github.com/WICG/animation-worklet)

CSS paint worklets (https://drafts.css-houdini.org/css-paint-api/#dom-css-paintworklet)

Layout worklet(https://drafts.css-houdini.org/css-layout-api/#dom-css-layoutworklet)

requestAnimationFrame

So... why not in the media pipeline? It might help with a few use cases:

1. e2ee
2. RTP data channel (for unified audio/video/data)
3. Media over QUIC (for unified audio/video/data)

Worklets Galore?

```
// RawFrame -> list<EncodedFrame> (simulcast/SVC)
RtpSender.setEncoder(worklet);
RtpSender.encode(rawFrame);  // -> list<EncodedFrame>, to get default behavior
// EncodedFrame -> list<RtpPacket>
RtpSender.setPacketizer(worklet);
RtpSender.packetize(encodedFrame);  // -> list<RtpPacket>, to get default behavior

// RtpPacket -> list<EncodedFrame> (RED; FEC)
RtpReceiver.setDepacketizer(worklet);
RtpReceiver.depacketize(rtpPacket);  // -> list<EncodedFrame>, to get default behavior
// EncodedFrame -> list<RawFrame>  (queued dependent frames)
RtpReceiver.setDecoder(worklet);
RtpReceiver.decode(encodedFrame);  // -> list<RawFrame>, to get default behavior

// This would allow sending and receiving over custom transports (SCTP or QUIC) and SFU in JS, but on the main thread.
RtpReceiver.injectEncodedFrame(EncodedFrame);
```

Can WebCrypto be synchronous in a worklet?

Worklets + ReadableStream Galore?

1. Define RtcWorklet, a worklet stream transform: `worklet + ReadableStream => ReadableStream` (but efficient and on special thread)
2. Add the following `ReadableStreams:`
   a. `MediaStreamTrack.frames`
   b. `RtpSender.encodedFrames`
   c. `RtpSender.rtpPackets`
   d. `RtpReceiver.rtpPackets`
   e. `RtpReceiver.encodedFrames`
3. Add the following methods:
   a. `RtpSender.setEncodedFrameSource(ReadableStream)`
   b. `RtpSender.setRtpPacketSource(ReadableStream)`
   c. `RtpReceiver.setRtpPacketSource(ReadableStream)`
   d. `RtpSender.setEncodedFrameSource(ReadableStream)`
   e. `QuicTransport.setStreamSource(ReadableStream)  // One stream per messages`
   f. `QuicTransport.getStreamSource  // One stream per message`

Worklets + ReadableStream Galore?

Custom encoder/decoder:

```
RtpSender.setEncodedFrameSource(RtcWorklet(MediaStreamTrack.frames, ...))
MediaStreamTrack.setFrameSource(RtcWorklet(RtpReceiver.encodedFrames, ...))
```

Double encryption per packet:

```
RtpSender.setRtpPacketSource(RtcWorklet(RtpSender.rtpPacket, ...))
RtpReceiver.setRtpPacketSource(RtcWorklet(RtpReceiver.rtpPackets, ...))
```

Custom transport for encoded frames

```
QuicTransport.setStreamSource(RtcWorklet(RtpSender.encodeFrames), ...)
RtpReceiver.setEncodedFrameSource(RtcWorklet(QuicTransport.getStreamSource()), ...)
```

RTP Data Channel

```
RtpSender.setRtpPacketSource(...);
```

RtpTransport

- **Provides crypto and congestion control (and reliability?)**
  - Crypto via SRTP
  - Congestion control via transport-cc?
  - Reliability via RTX + transport-cc?
- Apps send/receive RtpPackets, fully controlling PT, SSRC, seqnum, timestamp, payload, header extensions
- Which means it can send data by making a "data payload", which doesn't need to be standardized (but could be)

RtpTransport

```
interface RtpTransport {
  void sendRtpPacket(RtpPacket packet);
  void sendRtcpPacket(RtcpPacket packet);
  attribute eventhandler onrtppacket;
  attribute eventhandler onrtcppacket;
}
```

```
dictionary RtpPacket {
 unsigned octet payloadType;
 unsigned int sequenceNumber;
 unsigned long ssrc;
 array<RtpHeaderExtension> header_extensions;
 array<unsigned long> csrcs;
 bytes payload;
}
```

```
dictionary RtcpPacket {
  ...
}
```

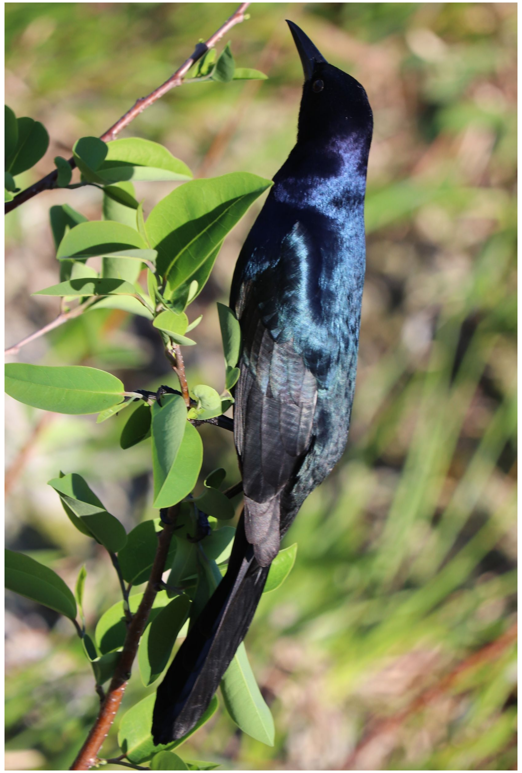
RtpTransport (ReadableStream version)

```
interface RtpTransport {
  void sendRtpPackets(ReadableStream);
  void sendRtcpPackets(ReadableStream);
  readonly attribute ReadableStream receivedRtpPackets;
  readonly attribute ReadableStream receivedRtcpPackets;
}
```

```
dictionary RtpPacket {
 unsigned octet payloadType;
 unsigned int sequenceNumber;
 unsigned long ssrc;
 array<RtpHeaderExtension> header_extensions;
 array<unsigned long> csrcs;
 bytes payload;
}
```

```
dictionary RtcpPacket {
  ...
}
```

For extra credit



Name that bird!

Thank you

<u>Special thanks to:</u>

W3C/MIT for WebEx

WG Participants, Editors & Chairs

The bird