W3C WebRTC WG Meeting

October 14, 2021
8 AM Pacific Time

Chairs: Bernard Aboba
Harald Alvestrand
Jan-Ivar Bruaroey

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 October 2021 interim meeting of the W3C WebRTC WG, at which we will cover:
  - WHATWG Streams Issues
  - Mediacapture-transform API

About this Virtual Meeting

- Meeting info:
  - https://www.w3.org/2011/04/webrtc/wiki/October_14_2021
- Link to latest drafts:
  - https://w3c.github.io/mediacapture-main/
  - https://w3c.github.io/mediacapture-extensions/
  - https://w3c.github.io/mediacapture-image/
  - https://w3c.github.io/mediacapture-output/
  - https://w3c.github.io/mediacapture-screen-share/
  - https://w3c.github.io/mediacapture-record/
  - https://w3c.github.io/webrtc-pc/
  - https://w3c.github.io/webrtc-extensions/
  - https://w3c.github.io/webrtc-stats/
  - https://w3c.github.io/mst-content-hint/
  - https://w3c.github.io/webrtc-priority/
  - https://w3c.github.io/webrtc-nv-use-cases/
  - https://github.com/w3c/webrtc-encoded-transform
  - https://github.com/w3c/mediacapture-transform
  - https://github.com/w3c/webrtc-svc
  - https://github.com/w3c/webrtc-ice
- Link to Slides has been published on WG wiki
- Scribe? IRC http://irc.w3.org/ Channel: #webrtc
- The meeting is being recorded. The recording will be public.
- Volunteers for note taking?

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

Virtual Interim Meeting Tips

This session is being recorded

- **Type +q and -q in the Google Meet chat to get into and 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.**
- **Poll mechanism may be used to gauge the “sense of the room”.**

Understanding Document Status

- Hosting within the W3C repo does ***not*** imply adoption by the WG.
  - WG adoption requires a Call for Adoption (CfA) on the mailing list.
- Editor's drafts do ***not*** represent WG consensus.
  - WG drafts ***do*** imply consensus, once they're confirmed by a Call for Consensus (CfC) on the mailing list.
  - Possible to merge PRs that may lack consensus, if a note is attached indicating controversy.

Issues for Discussion Today

- 08:10 - 08:45 AM Streams Pipeline Model (Youenn)
  - 08:10 - 08:30 Slides
  - 08:30 - 08:45 Discussion
- 08:45 - 09:30 AM Mediacapture-transform API Alternative Proposal (Youenn & Jan-Ivar)
  - Slides (08:45 - 09:15)
  - Discussion (09:15 - 09:30)
- 09:30 - 09:50 AM Mediacapture-transform API (Harald)
  - 09:30 - 09:40 Slides
  - 09:40 - 09:50 Discussion
- 09:50 AM - 10:00 AM Wrap-up and Next Steps

Time control:

- A warning will be given 2 minutes before time is up.
- Once time has elapsed we will move on to the next item.

The Streams Pipeline Model (Youennf)

End Time: 8:30 AM

Stream<VideoFrame> challenges (Youenn)

- **Goals**
  - List issues related to stream-based video pipelines
  - Identify blocking issues to adopt that design
  - Identify pros/cons vs. promise-based callback approach

- Identified issues
  - Issue 1063: Transferable streams: the double transfer problem
  - Issue 1124: Permitting transferable stream optimisation
  - Issue 1155: Is when pull() is called on an underlying source deterministic?
  - Issue 1156: Allow web authors to tee() with cloned chunks
  - Issue 1157: Allow web devs to synchronize branches with tee()?
  - Issue 1158: Piping to writable streams with HWM 0

Media pipeline using Stream<VideoFrame>

- Media pipeline using
  - ReadableStream/WritableStream as source/sinks
  - TransformStream as processing steps
  - pipeTo/pipeThrough to connect source, sinks & steps


```
const transport = new WebTransport("https://webtransport.org:8080/up");
const encoder = createTransformStreamFromWebCodec("H264");
const effect = new TransformStream({ transform: async (frame, controller) => {
  controller.enqueue(await applyEffect(frame));
});
videoStream.pipeThrough(effect)
           .pipeThrough(encoder)
           .pipeThrough(serializer)
           .pipeTo(transport.createUnidirectionalStream());
```

Where to run Stream<VideoFrame> pipeline?

- Best running a realtime pipeline off the main thread
- No guarantee where video frames will actually flow
  - If thread (where pipeline is set up) is blocked, what happens?
- Safest assumption
  - Video frames flow where the pipeline is set up

Example

```
// Example 1: camera with native transform displayed to HTMLVideoElement (no JS
in the pipe)
source.pipeThrough(new NativeBackgroundBlurTransform())
      .pipeTo(writableStreamToDisplayInHTMLMediaElement);

// Example 2: camera with JS transform displayed to HTMLVideoElement (some JS in
the pipe)
source.pipeThrough(new MySpecialWebGLBackgroundBlurTransform())
      .pipeTo(writableStreamToDisplayInHTMLMediaElement);

// Example 3: camera with native transform, teed to display and transmit (no JS
in the pipe)
const transformed = source.pipeTo(new NativeBackgroundBlurTransform());
const [displayed, transmitted] = transformed.tee();
displayed.pipeTo(writableStreamToDisplayInHTMLMediaElement);
transmitted.pipeThrough(new MyNativeEncoder())
           .pipeTo(new MyWebTransportStream(...));
```

Stream<VideoFrame> transfer to the rescue?

- Chrome optimized stream-transfer
  - Specifically for native video MediaStreamTracks
    - Avoids main thread if stream transferred to worker
    - But not standard (issue 1124) & difficult to make optimization transparent to web developers

Example

```
const worker = new Worker('MyVideoFrameProcessingWorker.js');

// Example 1: send camera stream directly.
worker.postMessage({action:'blur+transmit', source:cameraReadable }, [cameraReadable]);

// Example 2: send camera stream after native transform.
const transformed = cameraReadable.pipeTo(new NativeBackgroundBlurTransform());
worker.postMessage({action:'transmit', source:transformed }, [transformed]);

// Example 3: tee then transfer
const [toAnalyze, toTransmit] = cameraReadable.tee();
worker.postMessage({action:transmit', source:toTransmit }, [toTransmit]);

// Example 4: non-camera native streams
worker.postMessage({action:'blur', source:peerConnectionTrackReadable },
[peerConnectionTrackReadable]);
worker.postMessage({action:'crop', source:getDisplayMediaReadable },
[getDisplayMediaReadable]);
worker.postMessage({action:'transmit', source:transferredGetViewPortTrackReadable },
[transferredGetViewPortTrackReadable]);
```

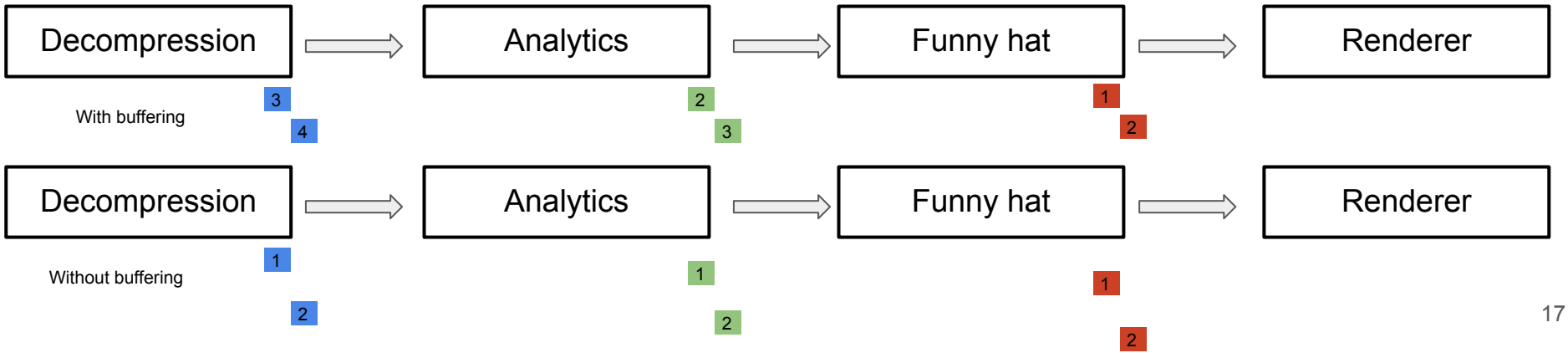
Stream<VideoFrame> transfer evaluation

- Transferring stream<VideoFrame> is a generic tool
  - Designed for flexibility, no performance guarantee
- MediaStreamTrack transfer is a dedicated tool
  - Designed to guarantee performance
    - Optimal path between frame source & destination

- Proposal
  - Build on MediaStreamTrack transfer
  - No hard requirement to resolve issue 1124
  - Stream under-the-hood optimization is a bonus, not a prerequisite

Stream<VideoFrame> pipeline buffering (1 / 3)

- Buffering happens at each transform step in the media pipeline
  - Even if transform does nothing
- Buffering is sometimes very good
  - Allows parallel processing, optimize latency on sink side
  - Great when you want to process all chunks in the pipe



Stream<VideoFrame> pipeline buffering (2 / 3)

- Pipeline buffering of VideoFrame(s) is a real issue
  - VideoFrame(s) are big and scarce resources
  - Sink is not processing the freshest VideoFrame
  - Buffering is mostly hidden from web developer
- We need a solution to turn off buffering!
  - Under active discussion (issue 1158)

```
new WritableStream((start => (controller) {
    controller.releaseBackpressure(); // To be called when needed.
}, { highWaterMark: 0 });
```

  - Chances are low to have no buffering by default

Stream<VideoFrame> pipeline buffering (3 / 3)

- Some buffering might be good, but not too much
  - Monitoring/updating the overall buffering in the media pipeline is desirable
- WhatWG streams do not make that easy
  - Streams queue state is mostly opaque to the application
    - By design
    - Some limited support in TransformStream (desiredSize)

- Unclear whether WhatWG streams fit the bill
  - Might be feasible, not easy

Why teeing ReadableStream<VideoFrame>?

- Tee is WhatWG streams usual way to allow multiple consumers

```
const transport = new WebTransport("https://webtransport.org:8080/up");
const encoder = createTransformStreamFromWebCodec("H264");
const renderer = createRendererAsWritableStream();
const effect = new TransformStream({ transform: async (frame, controller) => {
  controller.enqueue(await applyEffect(frame));
});

// Add a funny hat
videoStream.pipeTo(effect.writable);

const [branchToTransmit, branchToRender] = effect.readable.tee();
// Transmit to the other party
branchToTransmit
  .pipeThrough(encoder)
  .pipeThrough(serializer)
  .pipeTo(transport.createUnidirectionalStream());
// Render in local view
branchToRender.pipeTo(renderer);
```

But tee is broken

- Tee is exposing the same VideoFrame object in both branches
  - If branch1 closes a frame F, branch2 cannot make use of F

- We need a solution!
  - Under active discussion (issue 1156)
  - Will probably not be the default behavior
    - But should be straightforward to adopt

```
const [branch1, branch2 ] = effect.readable.tee({structuredClone: true});
```

- Are we good with tee now?

We need more changes to tee

- Tee creates hidden buffering
  - If branch1 & branch2 are not consumed at the same pace
    - This is the typical case: we want to drop frames if needed
- We need a solution!
  - Under discussion (issue 1157)
  - Unclear yet how it might work
    - And how easy it will be for web developers
  - Should be as good as JS promise-based callbacks
    - Keep backpressure/no buffering/independent frames

VideoFrame lifetime management (1 / 2)

- Expected design pattern
  - VideoFrame.close() should be called as early as possible
    - Do not rely on GC
- Consequence 1
  - Need a clear API contract on who should close frames
  - MSTG WritableStream is closing frames
    - To match MSTP not doing it
- Potential footgun
  - No built-in contract, streams are actually relying on GC
  - Not all streams may follow MSTP/MSTG pattern

VideoFrame lifetime management (2 / 2)

- Expected design pattern
  - VideoFrame.close() should be called as early as possible
    - Do not rely on GC
- Consequence 2
  - Aborting/cancelling a stream is relying on GC
    - Readable/WritableStream queues own video frames
      - Especially with tee+structure cloning
- Media pipeline will want to abort/cancel streams
  - In case of errors or in case of updating the pipeline
- We need a solution!

Stream<VideoFrame> challenges - conclusion (1 / 3)

- We need to solve those issues
  - Buffering, tee & lifetime management

- Scope is any video pipeline, not only camera pipeline
  - Audio processing has its own challenges
    - Not discussed here as we have WebAudio

- It is unclear whether and/or how these issues will be solved
  - Progress is being made but we need more

Stream<VideoFrame> challenges - conclusion (2 / 3)

- If we do not solve those issues
  - Stream-based design is inferior to promise-based callbacks
    - And vulnerable to easy-to-make but hard-to-debug bugs
  - MediaStreamTrack + Callback API as a default
    - Streams API as opt-in by developer

- If we solve those issues
  - Stream-based design is superior to promise-based callbacks
    - E.g. ability for more User Agent optimizations
  - Streams API = default, Callback API = opt-in

Stream<VideoFrame> challenges - conclusion (3 / 3)

- We need to solve those issues before selecting the API model
  - Where we want developers and APIs to go

- If we select streams as the foundation for media pipeline, we should extend streams integration with existing & new APIs
  - Current APIs are all based on callbacks
    - VideoDecoder, VideoEncoder
    - BarcodeDetector, FaceDetector

Discussion (End Time: 8:45 AM)

-

Alternative Mediacapture-transform API (Jan-Ivar)

End Time: 9:15 AM

Realtime Media pipeline is off main thread today (1)

MST is purely a *control surface* on main thread

Media flows “in parallel” (e.g. on a dedicated media thread)

```
const stream = await navigator.mediaDevices.getUserMedia({video: true});
const [track] = stream.getVideoTracks();
const pc = new RTCPeerConnection();
pc.addTrack(track, stream);
await track.applyConstraints({video: {height: {max: 200}}}); // main done
```

Media thread: camera frames → downscale → rtp sender ⚡

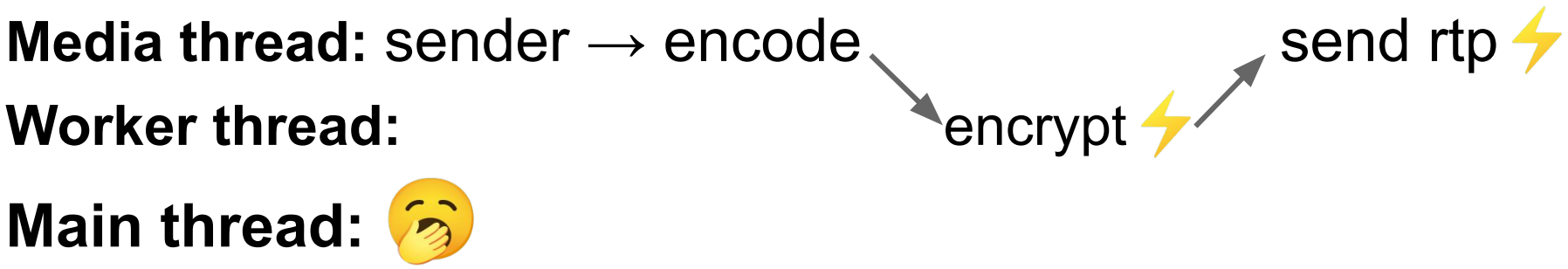
Main thread: 🤭

Media delivery never ever blocks on the main thread

Realtime Media pipeline is off main thread today (2)

True even in [webrtc-encoded-transform]:

```
const sender = pc.addTrack(track, stream);
sender.transform = new RTCRtpScriptTransform(worker, "myEncrypter");
sender.transform.port.postMessage("go");
sender.transform.onmessage = e => console.log("done"); // main done
```



And that's for *encoded* video not even raw video!

Why: The main thread is overworked & underpaid —

[Chrome Dev Summit 2019 (video)]

“We are setting ourselves up for failure here. We have no control over the environment our app will run in. ***The main thread is completely unpredictable****. What takes 2 ms on a modern flagship phone, might take 20 ms on the next low-end phone. How can we escape this unpredictability? … through Web Workers”*

60 fps = 16 ms deadline
90 fps = 11 ms deadline
120 fps = 8 ms deadline
144 fps = 6.9 ms deadline

WebCodecs decision to expose in Window

*"There is consensus and we agree that media processing in general should happen in a Worker context. Not all [WebCodecs] use cases require this, though, e.g., **non-realtime** transcoding. ...*

*A decision to expose WebCodecs in Window context should **not be interpreted as precedent for allowing the proposed MediaStreamTrackProcessor in Window**. That proposal should be discussed and considered on its own merits."*

"Best Practices for Authors Using WebCodecs: authors working with realtime media ... are encouraged to ensure their media pipelines operate in worker contexts entirely independent of the main thread ..."

What's up with mediacapture-transform? 🪟🐘

"NOT AN ADOPTED WORKING GROUP DOCUMENT."
Non-standard = no consensus. Google has shipped it in M94. 😢

- **Blocker: Exposes realtime pipeline on main thread by default**
  Tracks stay on main thread, while streams *may* be xfered or not
- **Blocker: Fails to encourage use in workers (users & UAs)**
  Requires extra optional work by web devs to get off main thread
  Relies on non-standard UA "optimizations" to fully get off main
  APIs are thread-coupled to main thread; unfriendly to workers
- **We need to standardize an API without these problems & reclaim URL https://w3c.github.io/mediacapture-transform/**
- MediaStreamTrack is now transferable (CfC: 👍) **Rethink API**

MediaStreamTrack is now transferable

Core principles from a worker point of view:

Q: A worker encounters a track, how does it access its data?

❌ postMessage it to main thread to ask it to create MSTP and postMessage mstp.readable back

✅ `track.readable`

The former makes no sense. Shouldn’t have to go back to main thread to get access. (It also doesn’t get us fully off main:)

Core principles from a worker point of view (2)

Q: A worker has video frames. How does it create a track from it?

❌ postMessage to ask main thread to create MSTG and postMessage its mstg.writable & mstg.clone() back

✅ `const {writable, track} = new VideoTrackSource()`

The former makes no sense. The worker should be able to create a track directly without having to ask main thread to do it.

MSTP/MSTG warps code organization to be main-thread oriented.

Core principles from a worker point of view (3)

Q: How does a worker send processed video frames  over WebTransport without dropping resolution or FPS in self-view?

❌ `const [r1, r2] = readable.tee();` #whatwg/1157

❌ Clone original track and process video frames twice

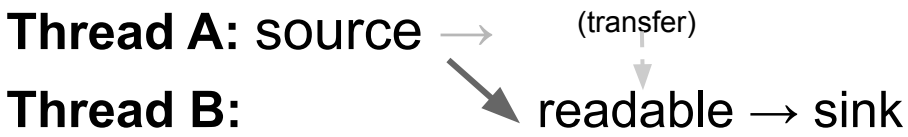
❌ postMessage *constraints* to main thread to create MSTG, applyConstraints to mstg.clone(), create MSTP from clone, postMessage mstg.writable + mstp.readable back

✅
```
const sendTrack = processedSource.track.clone();
await sendTrack.applyConstraints(constraints);
await sendTrack.readable.pipeTo(...);
```

MSTP/MSTG violate WHATWG Transferable Streams

> *"...and postMessage mstp.readable / mstg.writable back"*

WHATWG transferable streams have **tunnel semantics**: "*a special kind of identity transform which has the writable side in one realm and the readable side in another realm … to implement ... **cross-realm transforms**".* Meaning:



- They only transfer one side of the stream (consumer or producer).
- They exist to **create tunnels** between threads on purpose, ***NOT*** to solve creating them on the wrong thread in the first place
- Without **spec-breaking UA optimizations** the MSTP.readable remains anchored to main thread (on one side) after transfer.
- MSTP is a broken API built on broken assumptions.

#59 Alternative MediaCapture transform API (jib)

Goals

1. Align API with transferable MST for simpler API surface
2. Remove aforementioned blockers to standardization:
    a. Expose realtime media pipeline to workers (not main thread)
    b. Encourage use of workers by making it simple, friendly to workers, and the default. Discourage using main thread
3. Start with video (discussing audio is separate)

track.readable

WebIDL

```
partial interface MediaStreamTrack {
  [Exposed=DedicatedWorker] readonly attribute ReadableStream readable;
};
```

```
// worker.js

onmessage = async ({data: {track}}) => {

  const wt = new WebTransport("https://webtransport.org:8080/up");

  await track.readable
    .pipeThrough(createMyEncodeVideoStream({
      codec: "vp8",
      width: 640,
      height: 480,
      bitrate: 1000000,
    }))
    .pipeThrough(new TransformStream({transform: mySerializer}));
    .pipeTo(wt.createUnidirectionalStream());
};
```

VideoTrackSource()

```
// worker.js - a crop transform (fiddle)

onmessage = async ({data: {track}}) => {

  const source = new VideoTrackSource();

  parent.postMessage({track: source.track}, [source.track]); // → video.srcObject

  await track.readable
    .pipeThrough(new TransformStream({transform: crop}))
    .pipeTo(source.writable);
};

const canvas = new OffscreenCanvas(640, 360);
const ctx = canvas.getContext("2d", {desynchronized: true});

function crop(frame, controller) {
  ctx.drawImage(frame, 320, 180, 640, 360, 0, 0, 640, 360);
  controller.enqueue(new VideoFrame(canvas));
  frame.close();
}
```

WebIDL

```
[Exposed=DedicatedWorker]
interface VideoTrackSource {
  constructor();
  readonly attribute WritableStream writable;
  attribute boolean muted;
  readonly attribute MediaStreamTrack track;
}
```

Processed video → applyConstraints (main thread)

```
// worker.js - a crop transform with high-fps self-view & low-fps send

onmessage = async ({data: {track}}) => {
  const source = new VideoTrackSource();
  parent.postMessage({track: source.track}, [source.track]);

  await track.readable
     .pipeThrough(new TransformStream({transform: crop}))
     .pipeTo(source.writable);
}

// main.js (skipping setup)

const {data: track} = await new Promise(r => worker.onmessage);
selfView.srcObject = new MediaStream([track.clone()]); // 60 fps 

await track.applyConstraints({width: 320, height: 200, frameRate: 30});
const pc = new RTCPeerConnection(config);
pc.addTrack(track); // 30 fps 
```

Processed video → applyConstraints (worker)

```
// worker.js - a crop transform with high-fps self-view & low-fps send (no tee!)

onmessage = async ({data: {track}}) => {
  const source = new VideoTrackSource();
  const sendTrack = source.track.clone();
  parent.postMessage({track: source.track}, [source.track]); // 60 fps

  await sendTrack.applyConstraints({width: 320, height: 200, frameRate: 30});

  const wt = new WebTransport("https://webtransport.org:8080/up");

  await track.readable
    .pipeThrough(new TransformStream({transform: crop}))
    .pipeThrough({writable: source.writable, readable: sendTrack.readable})
    .pipeThrough(new TransformStream({transform: mySerializer}))
    .pipeTo(wt.createUnidirectionalStream()) // 30 fps
}
```

Alternate MediaCapture transform API (Jan-Ivar)

Benefits

1. Simpler API taking advantage of transferable MST
2. Fewer new API objects to learn. Friendly to workers
3. Satisfies core principles from a worker POV, without main-thread entangled APIs (“worker see, worker do”)
4. Doesn’t block realtime media pipeline on main thread by default
5. Source stays in worker, separate from its track(s). Clean xfer
6. applyConstraints available in the worker
7. Parity+ with MSTP/MSTG features & brevity (e.g. 41 lines)
8. Doesn’t need transferable streams UA “optimizations”
9. Source.muted attribute

Appendix A: “But we want promise callbacks, not streams”

No problem!

```
// worker.js

onmessage = async ({data: {track}}) => {

  const encoder = createMyVideoEncoderPromiseWrapper(config);
  let counter = 0;

  for await (const frame of track.readable) {
    try {
      await encoder.encode(frame, {keyFrame: ++counter % 130 == 0});
    } finally {
      frame.close();
    }
  }
};
```

(We get this for free with any streams-based proposal)

Appendix B: “But we want multiple readables”

Ways to multiply & reuse today:

1. `track.clone().readable`
2. `track.readable.tee({structuredClone: true}) // has issues but exists!`
3. `await track.readable.pipeTo(sink, {preventCancel: true}); // reuse on abort`
4. Or just use promise callbacks

Tradeoffs:

- Cloning a track means separate constraints, enabled state, & stop()
- Teeing has the above-mentioned issues #1156, #1157, #1158

Q: Do we need (instead?) (or both! like ontrack & addEventListener 🤠🤠)?

5. `track.createReadable()`

WebIDL

```
partial interface MediaStreamTrack {
  [Exposed=DedicatedWorker] ReadableStream createReadable();
};
```

Appendix C: Processed video → tee() in worker

```
// worker.js - a crop transform with high-fps self-view & low-fps send (fiddle)

onmessage = async ({data: {track}}) => {
  const source = new VideoTrackSource();
  parent.postMessage({track: source.track}, [source.track]); // → srcObject

  const readable = track.readable.pipeThrough(new TransformStream({transform: crop}));

  const [r1, r2] = readable.tee({synchronized: true, structuredClone: true}); // issues!

  const wt = new WebTransport("https://webtransport.org:8080/up");

  await Promise.all([
    r1.pipeTo(source.writable), // 60 fps 

    r2.pipeThrough(createFrameDropper()) // releases backpressure to lift fps limit on r1
      .pipeThrough(createMyEncodeVideoStream({codec: "vp8", width: 640, height: 360}))
      .pipeThrough(new TransformStream({transform: mySerializer}))
      .pipeTo(wt.createUnidirectionalStream()) // ~30 fps 
  ]);
```

Discussion (End Time: 9:30 AM)

-

Mediacapture-transform API (Harald)

End Time: 9:40 AM

Recap: Breakout Box API

- MediaStreamTrackProcessor
  - A destination for a MediaStreamTrack
  - Generates a ReadableStream of media frames
- MediaStreamTrackGenerator
  - Exposes a WritableStream of media frames
  - Implements the MediaStreamTrack API
- No frills.

Implementation and Usage

- Shipped in Chrome 94
- Actively used in products
- New features basing themselves on it
- Very few problems reported

Threading model approaches

We believe the threading model should be picked by the app developer, not by the platform developer.

- Streams are Transferrable. This means that they can go wherever they need to go - today.
- We have agreed on Transfer and worker availability for MediaStreamTrack. Correspondingly, we added worker availability for MediaStreamTrackGenerator and MediaStreamTrackProcessor to this proposal.

Needed: Worked examples

People tend to copy examples, not code from first principles.

So the examples are what will be followed.

https://webrtc.github.io/samples/ has a few working ones.

API expansion

Lots of things remain to be done:

- Better control of adaptation at source
- More experience with streams that aren’t cameras (especially synthesized media)
- Scenarios that need nontrivial buffering

These can be worked on. Once we agree on basics.

API proposal comparison

	Mediacapture-transform proposal	**Alternate proposal**
Frame delivery	Streams	Streams
Exposure	Worker and Main	Worker only
Stream generator	Separate class	MediaStreamTrack+
Stream consumer	MediaStreamTrack+	Separate class
Media types	Audio and Video	Video only

The differences seem to be fairly isolated from each other, so can be discussed one by one.

Discussion (End Time: 9:50 AM)

-

Wrap-up and Next Steps (9:50 - 10:00)

- Question 1:
- Question 2:
- Question 3:

Thank you

Special thanks to:

WG Participants, Editors & Chairs