# RTCRtpSender/Receiver

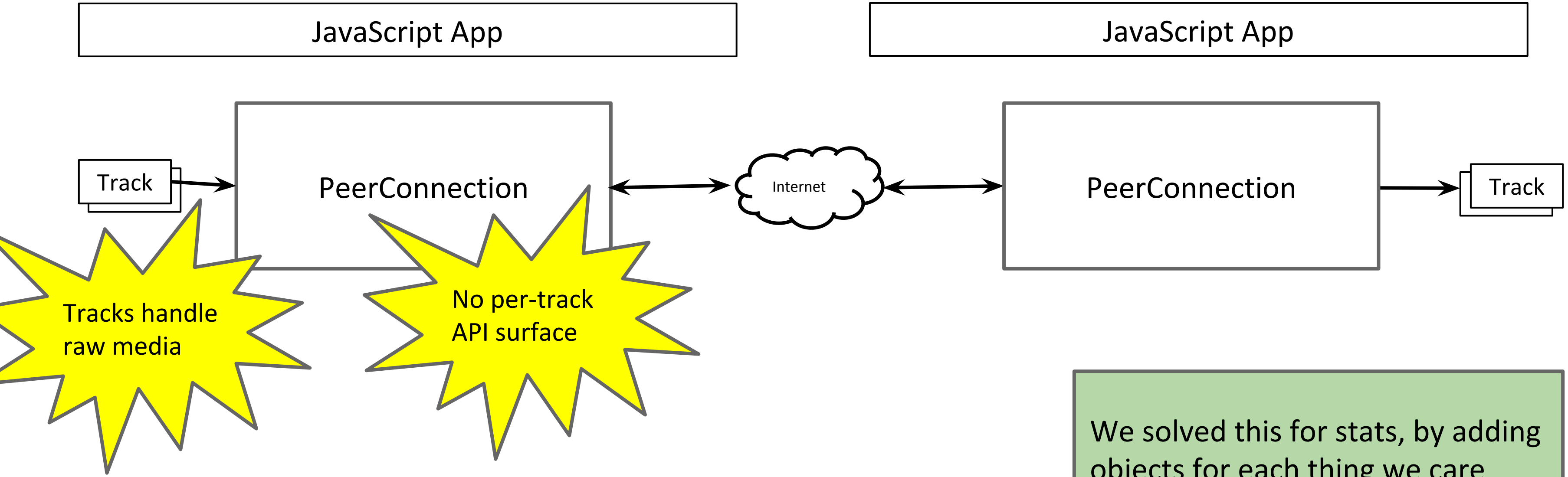
Justin Uberti
Peter Thatcher

Oct 2014

# Recap

*(stuff we already agreed on)*

# Core Issue: Insufficient Object Model

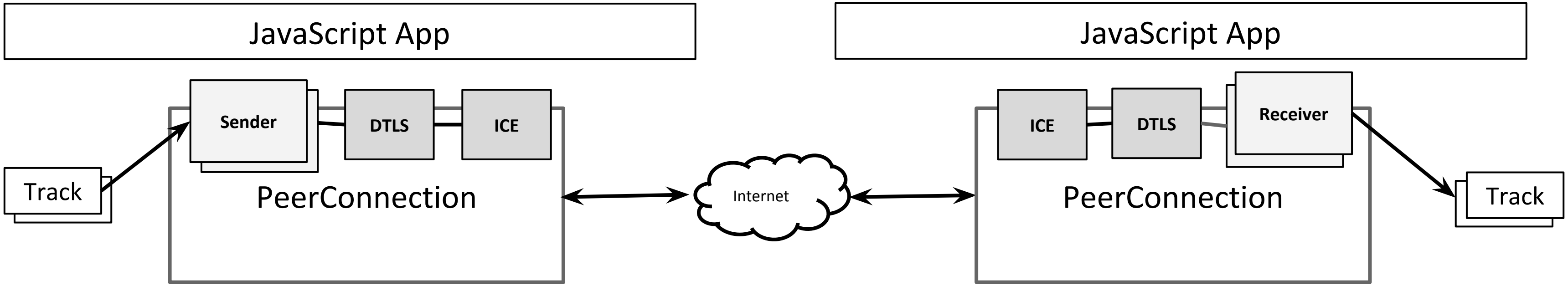


Tracks handle raw media

No per-track API surface

We solved this for stats, by adding objects for each thing we care about, but these objects are hidden inside PeerConnection

# Solution Diagram



Applications now have an API surface with the right multiplicity to do per-track operations

# API: Recap

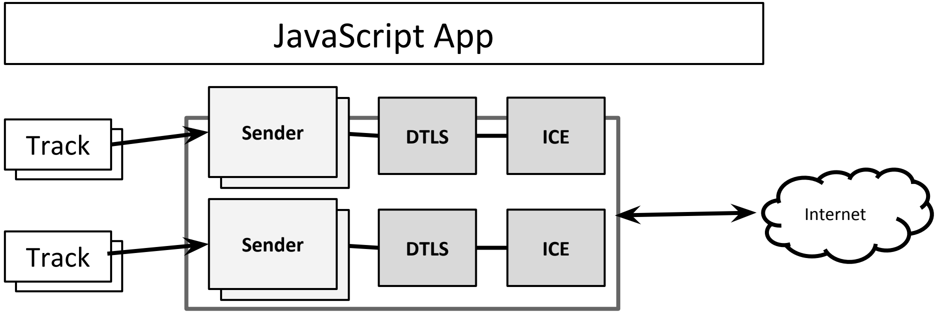
```
interface RTCRtpSender {
  readonly attribute MediaStreamTrack track;
};

interface RTCRtpReceiver {
  readonly attribute MediaStreamTrack track;
};
interface TrackEvent : Event {
  readonly attribute RtpReceiver receiver;
  readonly attribute MediaStreamTrack track;
  sequence<MediaStream> getStreams();
};

partial interface RTCPeerConnection {
  // |streams| parameter indicates which particular streams should be referenced in signaling
  // Fails if |track| has already been added
  RTCRtpSender addTrack(MediaStreamTrack track, MediaStream... streams);  // replaces addStream
  void removeTrack(RTCRtpSender sender);    // replaces removeStream
  sequence<RTCRtpSender> getSenders();      // replaces getLocalStreams
  sequence<RTCRtpReceiver> getReceivers();  // replaces getRemoteStreams
  EventHandler ontrack;  // replaces onaddstream

};
```

# Next Steps

# Transports

- Like RTP streams, transports are also not exposed well from **PeerConnection**; hard to get commonly needed data
  - per-transport ICE state
  - active local and remote candidates
  - Remote DTLS certificates
- Easy to fix with our object model
  - **RTCRtpSender** and **RTCRtpReceiver** add a .**transport** property, yielding a **RTCDtlsTransport** object
  - Multiple senders can share a **RTCDtlsTransport**
  - **RTCDtlsTransport** connects to an **RTCIceTransport** object

# Example Diagram



# API: Transports

```
partial interface RTCRtpSender {
  readonly attribute RTCDtlsTransport transport;
};
partial interface RTCRtpReceiver {
  readonly attribute RTCDtlsTransport transport;
};

interface RTCDtlsTransport {
  readonly attribute RTCIceTransport transport;    // the associated ICE transport
  readonly attribute RTCDtlsTransportState state; // current DTLS state (e.g. connected, failed)
  sequence<ArrayBuffer> getRemoteCertificates();   // the certs in use by the remote side
  attribute EventHandler? onstatechange;
};
interface RTCIceTransport {
  readonly attribute RTCIceConnectionState state; // the current ICE state
  RTCIceCandidatePair? getSelectedCandidatePair(); // the currently active candidate pair
  attribute EventHandler? onstatechange;
  attribute EventHandler? onselectedcandidatepairchange;
};
```

# EncodingParameters

- Now that we have **RTCRtpSender**, what can we do with it?
    - Read the current encoding parameters
    - Make some changes to the track encoding
    - Some changes don't require negotiation:
        - e.g. changing max send bitrate
    - Changes that do require negotiation result in **onnegotiationneeded**, and don't take effect until **setLocalDescription**:
        - e.g. pausing a MST, results in "a=sendonly"
    - Cannot change things that would be inconsistent with SDP
        - e.g. changing the send codec
- Any functionality that is needed must have no negotiation, or have well-defined SDP

# API: EncodingParameters (1.0)

```
partial interface RTCRtpSender {
  RTCRtpParameters getParameters();
  // Specifies the details of what to send (e.g. bitrate)
  // do .get() -> change -> .set()
  void setParameters(RTCRtpParameters parameters);
};

dictionary RTCRtpParameters {
  // In 1.0, only N=1 encodings are allowed. To change encodings,
  // in the future, N can be > 1, for simulcast or layered coding
  sequence<RTCRtpEncodingParameters> encodings;
}
dictionary RTCRtpEncodingParameters {
   unsigned int       ssrc;                // identifies the encoding; readonly
   boolean            active;              // sending or "paused/onhold"
   unsigned int       maxBitrate = null;   // maximum bits to use for this encoding
};
```

# Example

```
// put stream on hold
var sender = pc.getSenders()[0];
var params = sender.getParameters();
params[0].active = false;
sender.setParameters(params);
pc.onnegotiationneeded = () =>
    pc.createOffer().then(offer => pc.setLocalDescription(offer).then(() => signal(offer)));

// turn it up to 11 (Mbps)
var sender = pc.getSenders()[0];
var params = sender.getParameters();
params[0].maxBitrate = 11000000;
sender.setParameters(params);
```

# Capabilities

- Problem:  I can't know what the browser is capable of without calling **createOffer** and inspecting the SDP.
  - e.g.: does the browser support VP9?

- Solution: Why don't you just tell me what you support?  
  => **RTCRtpSender.getCapabilities**
  - just like **MediaDevices.getSupportedConstraints**

# API: getCapabilities

```
partial interface RTCRtpSender {
  static RTCRtpCapabilities getCapabilities(
      optional DOMString kind);
};
partial interface RTCRtpReceiver {
  static RTCRtpCapabilities getCapabilities(
      optional DOMString kind);
};

dictionary RTCRtpCapabilities {
  sequence<RTCRtpCodecCapability> codecs;
  sequence<RTCRtpHdrExtCapability>
    headerExtensions;
};
```

```
dictionary RTCRtpCodecCapability {
    DOMString     kind;  // audio | video
    DOMString     name;  // e.g. PCMU
    unsigned long clockRate;   // sampling
    unsigned long numChannels; // 1 or 2
};

dictionary RTCRtpHdrExtCapability {
    DOMString kind;  // audio | video
    DOMString uri;   // ...ssrc-audio-level
};
```

# Example

```
var videoCodecs = RTCRtpSender.getCapabilities("video").codecs;
var supportsVP9 = false;
for (var i = 0; i < codecs.length; ++i) {
  if (codecs[i].name.toLowerCase() === "vp9") {
    supportsVP9 = true;
  }
}
```

# API: RtpSender.track

- readonly in current API
- If we make it mutable, it makes for an easy solution to a long-existing problem: **how to switch between front and back camera**?

```
getUserMedia(video: {facingMode: "front"}) (stream) =>
    pc.addTrack(stream.getVideoTracks()[0]);

getUserMedia(video: {facingMode: "back"}) (stream) =>
   pc.getSenders()[0].track = stream.getVideoTracks()[0];
```

# Observations

- No signaling needed when track is changed
- MSID remains the same in future signaling
- Implies that MSID is actually a property of the **RTCRtpSender/Receiver**
- But **RTCRtpSender/Receiver** already are associated with a m= line, e.g. a MID
- As such, do we still need MSID for correlation?

# API: RtpSender.mid/msid

- MID correlates of sender/receiver with generated SDP
- Can also have MSID (open for discussion)

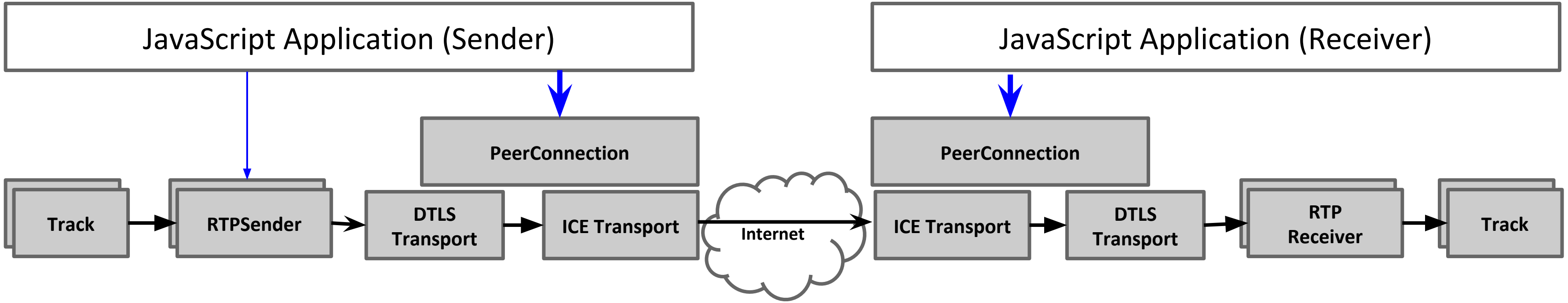
```
partial interface RTCRtpSender {
  attribute MediaStreamTrack track;
  readonly attribute DOMString mid;   // MID of sender; used in RTP hdrext
  readonly attribute DOMString msid;  // MSID of initial track; may not == track.id
};
partial interface RTCRtpReceiver {
  readonly attribute MediaStreamTrack track;
  readonly attribute DOMString mid;   // MID from signaling
  readonly attribute DOMString msid;  // MSID from signaling; always == track.id
};
```

# Consensus?

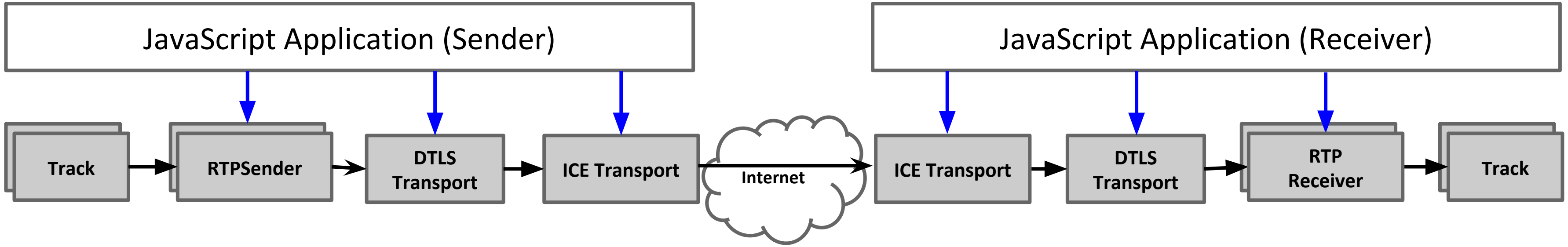
# Now that we have all these objects

The next logical step is to allow apps to configure the objects directly.  Advanced apps don't need a **PeerConnection** to use an **RtpSender** with a **DtlsTransport** and an **IceTransport** (although simple apps will likely want to use **PeerConnection**)

# WebRTC 1.0: Configuration via PeerConnection.setLD/setRD



# WebRTC 1.1: Direct configuration via .setParameters



# Benefits of direct control

- **RTCRtpSender.setParameters** can do more, because it's unconstrained by the rule of "any functionality that is needed must have no negotiation, or have well-defined SDP"
- More of a "do what I say" API; any negotiation logic handled in JS
- More flexible for different forms of signalling

# Example

```
var ice = new RTCIceTransport();
var dtls = new RTCDtlsTransport(ice);
var sender = new RTCRtpSender(dtls);
sender.setParameters({
  codecs: [
    {name: "vp8", payloadType: 100, ...}
  ],
  encodings: [
    // Simulcast
    {ssrc: 1, scale: 0.25, ...}
    {ssrc: 2, scale: 0.5, ...}
    {ssrc: 3, scale: 1.0, ...}
  ],
  rtcp: { cname: "doohickey" }
});
signal(sender.getParameters()); // Let the remote side know.
```