RTCPeerConnection Control Surface

WebRTC 2014-04-28

Bar Setting

- RTCPeerConnection is a terrible API
  - It’s indirect, not imperative
  - It’s opaque
  - The timing is bizarre
  - The interaction model is inconsistent
- So new features don’t have a high bar to meet

Existing Offer/Answer Control Surface

- RTCPeerConnection constructor (RTCConfiguration)
    - STUN/TURNS servers basically
- Arguments to createOffer
    - Whether to open audio/video slots for the answerer to use
- Mutations to SDP input to RTCSessionDescription constructor
    - Section 6 of -jsep describes some aspirational goals
- addStream arguments
    - MediaStreamTrack state (muted, enabled, readyState, id) affect SDP

Doohickamajiggities

- Provides a track-specific control surface
- More granular, easier to do tricky things
  - Originally to distinguish different directionality attribute semantics: a=sendonly/recvonly/inactive
- Opens other possibilities
  - Control: bandwidth, transport placement (bundling, RTCP multiplexing), simulcast, layering, CNAME, …
  - Feedback: statistics and state

What (gUM-like) Constraints Can Do For You

- Constraints have several features that we don't actually have on RTCPeerConnection
    - Ability to discover what is likely to actually do something, i.e., capabilities
    - Ability to discover what has actually been done, i.e., status
- We can build those features
- We should build those features

What (gUM-like) Constraints Cost

- Constraints come with unneeded extras (YAGNI)
    - Like the bit where multiple actors apply constraints on a single resource and the browser mediate between those actors using constraints to find a common mode
        - Or where you let the browser to choose from a set or range of acceptable options (valuable for some cases, like bandwidth)
- Constraints have some drawbacks (Least Surprise)
    - They don’t use the usual feature-detection mechanisms
    - Browser flexibility creates opaqueness, which is only mitigated by the status mechanism

With Constraints

- var canSend = whatsit.getCapabilities()
  .hasOwnProperty(“send”);
- whatsit.applyConstraints({ send: true });
- var isSending = whatsit.getConstraints().send;

Without

- var canSend = typeof whatsit.send !== ‘undefined’;
- whatsit.send = true;
- var isSending = whatsit.send;

What Might Work

- Some of the values we are trying to control work better if the browser is given some leeway
  - Resolution
  - Frame rate (at discrete intervals)
  - Bandwidth (minimum is not particularly useful)
- These might justify the use of constraints
  - Even fallback (“advanced”) might allow for definition of co-dependent settings

Taking It Up a Notch

- Bandwidth truly does need to leave the browser some flexibility:
    - thingamy.applyConstraints({ bandwidth: { max: 100 } });
    - thingamy.setBandwidthLimits(100 /*, undefined */);
- Chicken:
    - thingamy.applyConstraints({ chicken: [“chicken”, “chicken”, chicken] });
    - thingamy.setChicken([“chicken”, “chicken”, chicken]);
- Layers:
    - thingamy.applyConstraints({
      layers: [/* complex stuff */]
      });
    - thingamy.setLayer(0, { /* less complex stuff */ });