Introduction

After attempting to write out some use cases using the existing RTCRtpSender and RTCRtpReciever objects and parameters for ORTC, some issues were discovered. Specifically, the application developer would need to have a fair amount of knowledge on exactly how to tweak low level parameters for anything beyond very simple use cases. For example, setting up an SVC (Scalable Video Codec) would have required knowing about what codecs support SVC, how the layering is setup for particular codecs, and finally setting up specific geometric (or temporal) attributes and layering relationship details by an application developer.

As a result of the lack of easily configuration of RTP features, the idea came out to give the application developer "preferences" where the developer could choose what they want desire with high level knobs and dials and let the engine (which has explicit knowledge of each codec) configure the low level "parameters" details according to a developer's wishes. The engine could then return the closest set of preferences that could be achieved given the capabilities of the engine and the developer can then choose to proceed or not setting up media flows using these preferences and constructed parameters.

Another important discovery was made in the process of defining "preferences". If two ORTC engines were given the same set of preferences and the capabilities of both sender and receiver, each engine could be made to construct "compatible" sender and receiver "parameters" details without ever exchanging the parameter details over the wire. This small realization about generating "parameters" from capabilities for local consumption by an engine has a huge impact. This generation removes the need for an engine to understand and filter settings that it may not understand created by another engine of unknown origin, which may use proprietary and/or custom settings. A simple "ignore capabilities you don't understand" rule

could replace complex and cumbersome rules that would be otherwise required if "parameters" were to be sent over the wire and later filtered using a set of capabilities.

Parameters can be generated based on the union of sender and receiver capabilities along with application developer preferences being used as a guideline on how to create the parameters. The engine will do it's best to fulfill the preferences and it will return the parameters that are possible given the union of the capabilities.

Two different engines must be able to compute compatible parameters given all the same preferences and capabilities. Fortunately, any two engines that understand the same capabilities can easily follow the same rules to generate compatible parameters. While the parameters created on the sender and receiver are required to be "compatible", they need not be identical. The application developer should call "*createParameters(...)*" on sender to create parameters suitable for the sender. The application developer should call "*createParameters(...)*" on the receiver to create params suitable for a receiver. The calculated "parameters" for both sender and receiver have to be compatible only to the extent that whatever a sender produces a receiver must be capable of decoding.

The application developer has the option to tweak the detailed parameters output by "*createParameters(...)*" but should only do so with extreme caution. The resultant parameters output by "*createParameters(...)*" are only meant for local consumption by the local sender / receiver "start" methods. Sending these created parameters over the wire is discouraged because implementations may produce objects which may not be entirely understable by the remote party, even though the media sent on the wire will be compatible.

Differences from Current Sender / Receiver API

Both models and APIs are more similar than they are different. The subtle differences make important behavioural usage implications.

Both models send and receive based upon "parameter" settings. The difference is in how the "parameters" are generated. The new model generates the "parameters" based on an exchange of capabilities and the application developer is given convenient 'knobs' called "preferences" to perform most common use cases. The "parameters" in the new model are intended for local consumption only and the application developer is not required (and actively discouraged) from marshalling these "parameters" over the wire. The new model proposes marshaling and exchanging "capabilities" and optionally "preferences" and then generating compatible "parameters" based on those exchanges.

In both models, the application developer may choose to tweak low level parameters should specific compatibilities be required. But the "preferences" model allows most application developers to completely ignore the low level parameters.

Advantages of the New Capabilities Model

Overall the proposed capabilities based API has strong advantages. Main advantages are:

A. Simplicity in setup based on "preferences" for the application developer
B. Less brittle designs/implementations since low level parameters are not exchanged, filtered, and interpreted by different browser engines
C. Much less knowledge (and often no pre-knowledge) is required for the application developer to take full advantage of a browser's capabilities

RTCRtpSender / RTCRtpReceiver

```
interface RTCRtpSender {
  // ...

  static RTCRtpParameters createParameters(
    MediaStreamTrack track,
    Capabilities receiverCaps,
    optional (RTCRtpAudioPreferences or
              RTCRtpVideoPreferences or
              RTCRtpSimulcastPreferences) prefs,
    optional Capabilities senderCaps   // optional as system can obtain this
information
  );

  void start(RTCRtpParameters params);

  // ...
);

interface RTCRtpReceiver {
  // ...

  static RTCRtpParameters createParameters(
    DOMString kind,
    Capabilities senderCaps,
    optional (RTCRtpAudioPreferences or
              RTCRtpVideoPreferences or
              RTCRtpSimulcastPreferences) prefs,
    optional Capabilities receiverCaps // optional as system can obtain this
information
  );

  void start(RTCRtpParameters params);

  //...
);
```

RTCRtpMediaPreferences

```
// This is the base dictionary used for both audio and video preferences and
represents
// the set of common preferences that are available for both media types.
dictionary RTCRtpMediaPreferences {
    // If not specified, system will choose value. If specified, this receiverId will
    // be applied to primary SSRC “as is”. If more than one SSRC is needed to encode
    // the stream (e.g. FEC, RTX, MST, simulcast), where the meaning of the RTP
packet
    // with that alternative SSRC cannot be determined by the media flow itself, the
    // alternative SSRCs will construct a receiverId value based upon this receiverId
    // value.
    DOMString            receiverId;

    // This is the primary SSRC to use. Should alternative SSRCs be required (e.g.
FEC,
    // RTX, MST, simulcast), all other SSRCs should be assigned sequentially starting
    // from the chosen SSRC value.
    unsigned int         ssrc;

    // For a sender, force the chosen codec to be the codec within the
RTCRtpCapabilities
    // with this name. If possible to choose this codec, the system will confirm by
    // choosing this codec in the result from "createParameters(...)".
    // This value has no meaning for a receiver since a receiver must be capable
    // of receiving any of the compatible codecs within the union RTCRtpCapabilities.
    // A non specified value indicates the system will choose the preferred sending
    // codec.
    DOMString            codecName;

   // This value indicates the relative importance of the media being sent with a
   // sender versus other media being sent. The logic is that all sent media with
   // the same priority will be treated as having an equal priority. Those with
   // a greater value will be given a greater priority and those with a lower value
   // will be given a lower priority. The value is relative meaning a value of 2.0
   // should be given roughly 2 times the priority vs a 1.0 value and a value of 4.0
   // should be given roughly 4 times the priority vs a 1.0 value.
   double                relativePriority = 1.0;

   // This value indicates the maximum bit rate the media is allowed to output as
   // a combined whole (including all layers, FEC, RTX, etc). The system will filter
   // out codecs that are not capable of delivering below this bit rate unless no
   // codec is possible in which case the system will chose the minimal codec bit
rate
   // possible and will override with a different maximum bit rate in the result of
```

```
   // "createParameters(...)".
   double                maxBitrate;             // engine, keep under this rate

   // These values indicates the preferred treatment of FEC/RTX for the RTP packets.
For
   // audio, some audio codecs have built in FEC/RTX mechanisms in which case if the
   // codec is capable, the codec should enable its FEC/RTX mode if value is set to
all
   // for that codec rather than creating an additional RTP flow.
   RTCRtpRecoveryOptions fec = "none";
   RTCRtpRecoveryOptions rtx = "none";
};

enum RTCRtpRecoveryOptions {
   "all",     // apply to all layers
   "base",    // only apply for base (audio will treat "base" as equivalent to "all")
   "none"     // do not apply to any layer
};
```

RTCRtpAudioPreferences

```
dictionary RTCRtpAudioPreferences : RTCRtpMediaPreferences {
   // If not 0, tells the engine to pick and configure codecs that are capable of
   // the minimum of channels (if possible). If not possible, the minimum number of
   // channels will be returned in the result of "createParameters(...)".
   unsigned int         minChannels = 0;

   // If not 0, tells the engine to pick a codec and configure codecs which are
   // capable of delivering the minimum Hz rate as indicated. If not possible, the
   // minimum Hz rate will be returned in the result of "createParameters(...)"
   unsigned int         minHzRate = 0;

   // The engine will choose and configure the codecs best able to deliver the level
   // of fidelity requested.
   RTCRtpAudioFidelity  fidelity = "speech";
};

enum RTCRtpAudioFidelity {
  "speech",   // speech only is expected so Hz range only need to support the vocal
range
  "music",    // music is expected, choose stereo compatible and minimal 32000 Hz
  "movie"     // music / sound effects expected, choose surround and highest Hz
available
};
```

RTCRtpVideoPreferences (and related)

```
dictionary RTCRtpVideoPreferences : RTCRtpMediaPreferences {

   // minFrameRate, minScale, and minQuality each indicate that the engine must do
   // it's best effort to keep the frame rate, scale or quality above a certain
minimal
   // level. When using SVC, these values will hint at the requirements typically
needed
   // for the base layer.
   //

   // minFrameRate is specified in frames per second.
   double     minFrameRate = 0;    // please engine, keep equal or above this rate
   // minScale is a relative value from 0.0 to 1.0 where 1.0 represents full input
stream
   // width/height is requested and 0.0 represents no minimize size is requested.
   // The value of minScale is multiplied by the source video window width and height
   // to calculate a minimal width and height that is relative to source size.
   double     minScale = 0;        // please engine, keep equal or above this scale
   // Alternatively, a specific fixed minimal width and height can be requested.
   double     minWidth = 0;        // please engine, keep above X pixels wide
   double     minHeight = 0;       // please engine, keep above Y pixels high
   // minQuality is a relative value from 0.0 to 1.0 where 1.0 means maximum output
   // quality is requested for a given codec and 0.0 allows any minimal codec quality
   // output is deemed acceptable.
   double     minQuality = 0;      // please engine, keep equal or above this quality

   // The engine needs values to help decide what to sacrifice when network
conditions
   // are not ideal. The frameRatePriority, scalePriority, and qualityPriority
indicate
   // the relative importance of each aspect of the video relative to the other (or
   // 0.0 which means the video aspect has no significance (with exclusion to the
minimum
   // above). The values are relative to each other thus a value of 2.0 vs 1.0 has
   // roughly 2 times the importance and a value of 4.0 vs 1.0 has roughly 4 times
the
   // importance (relatively speaking).
   double    frameRatePriority = 1.0; // priority of frame rate
   double    scalePriority = 1.0;     // priority of scale
   double    qualityPriority = 1.0;   // priority of quality

   // If a type of SVC layering is desired, the frameRateScalabilityOptions,
   // scalingScalabilityOptions, and qualityScalabilityOptions should be set to a
   // non-null value for each SCV type desired. The details of the
```

```
   // RTCRtpScalabilityOptions dictionary will indicate the desired details for
   // each individual SVC type requested.
   //
   // Default of null indicates no SVC of specific type is requested.
   RTCRtpScalabilityOptions? frameRateScalabilityOptions = null;
   RTCRtpScalabilityOptions? scalingScalabilityOptions = null;
   RTCRtpScalabilityOptions? qualityScalabilityOptions = null;
};

dictionary RTCRtpScalabilityOptions {
   // If the alternative value other than the default value of null is specified,
this
   // indicates to the engine the precise number of layers desired (if possible for a
   // given codec to deliver these layers). If null, the engine is free to choose
   // the default layering statically or dynamically dependent upon the codec
   // capabilities.
   unsigned int?     layers = null;
};
```

RTCRtpSimulcastPreferences

```
dictionary RTCRtpSimulcastPreferences {
   // This value indicates the maximum bit rate all media is allowed to output as
   // a combined for all simulcast streams.
   double?               maxBitrate = null;           // engine, keep under this rate

   sequence<RTCRtpVideoPreferences> simulcastStreams;
};
```

RTCRtpParameters

```
// Typically this object is constructed by the RTCRtpSender for local consumption by
// the RTCRtpSender and by the RTCRtpReceiver for local consumption by a
RTCRtpReceiver.
// This is a "shotgun" object, meaning the developer is given the power of a
"shotgun"
// pointed at their feet and they can mess with this object at their own peril should
// they need to modify it for unusual compatibility reasons. Normal use cases should
not
// require modifying the values within this structure and marshalling this structure
for
// remote consumption by another browser engine is highly discouraged.
dictionary RTCRtpParameters {
   // When returned as a result, the system will express the actual chosen
preferences
   // possible to best fulfill the preferences given the capabilities. In other
words,
   // the developer can't always get what they want; but if they try sometimes, they
will
   // get what they need.
   (RTCRtpAudioPreferences or
    RTCRtpVideoPreferences or
    RTCRtpSimulcastPreferences) preferences;

   // the capabilities of both sender and receiver [value "as is" when passed
   // "createParameters(...)]"
   RTCRtcCapabilities senderCapabilities;
   RTCRtcCapabilities receiverCapabilities;

   // This value contains all the particularly low level details of how the engine
   // will encode the media on the wire.
   (RTCRtpParameterAudioDetails or
    RTCRtpParameterVideoDetails or
    RTCRtpParameterSimulcastDetails)  details;

   // The chosen RTP features based upon the union of the capabilities.
   Settings rtpFeatures;

   // The chosen RTP extensions and configurations based upon the union of
   // the capabilities.
   sequence<RTCRtpHeaderExtensionParameters>? headerExtensions = null;
};
```

RTCRtpParameterDetails

```
// This is the base dictionary of common parameters needed for both audio and video
media
// types. Audio and video will each have their own set of specific parameters
depending
// upon the media type.
dictionary RTCRtpParameterDetails {
   DOMString        receiverId = "";    // use this receiver ID for RTP stream ("" =
N/A)
   unsigned int     ssrc = null;        // using this SSRC for RTP stream

   DOMString        fecReceiverId = ""; // use this receiver ID for FEC RTP ("" =
N/A)
   unsigned int?    fecSsrc = null;     // using this SSRC for FEC (null = N/A)
   Settings         fec;                // modes of operation related to FEC

   DOMString        rtxReceiverId = ""; // use this receiver ID for RTX RTP ("" =
N/A)
   unsigned int?    rtxSsrc = null;     // using this SSRC for FEC (null = N/A)
   Settings         rtx;                // modes of operation related to RTX

   // null for a sender. For a receiver, this must contain the source SSRC to
   // use for RTCP Receiver Reports (RRs).
   unsigned int?    rtcpSsrc = null;

   // If true, the engine will mux RTCP with RTP on the same RTCIceTransport. If
false,
   // the engine will send RTCP reports on the associated RTCP RTCIceTransport
component.
   boolean          rtcpMux = true;
};
```

RTCRtpParameterAudioDetails (and related)

```
dictionary RTCRtpParameterAudioDetails : RTCRtpParameterDetails {

   // Contains a list of audio codec options per possible to use codecs. The order
   // of the codecs is in preferred order.
   sequence<RTCRtpParameterAudioCodecDetails> codecDetails;
};

dictionary RTCRtpParameterCodecDetails {
   // The name of the codec as related to the codec name(s) contained within the
codecs
   // listed within the RTCRtpCapabilities dictionaries.
   DOMString       codecName;

   unsigned byte   payloadType;       // actual payload type sent on wire
   Settings        formatsParameters; // detailed settings chosen for related codec
};

dictionary RTCRtpParameterAudioCodecDetails : RTCRtpParameterCodecDetails {
   // nothing anything required at this time?
};
```

RTCRtpParameterVideoDetails (and related)

```
dictionary RTCRtpParameterVideoDetails : RTCRtpParameterDetails {

   double           scale = 1.0;        // 0..1 relative scale from source
   double           frameRate = 1.0;    // 0..1 relative frame rate from source
   double           quality = 1.0;      // 0..1 relative quality from source

   // Contains a list of video codec options per possible to use codecs. The order
   // of the codecs is in preferred order.
   sequence<RTCRtpParameterVideoCodecDetails> codecDetails;
};

dictionary RTCRtpParameterVideoCodecDetails : RTCRtpParameterCodecDetails {

   // When layering is used, this value contains a sequence containing the layer
   // information as needed for the related codec.
   sequence<RTCRtpParameterVideoLayerDetails>? layers = null;

};

dictionary RTCRtpParameterVideoLayerDetails {
   // Value is set if required for describing the dependency tree information for
   // the codec's layers.
   DOMString             layerId = "";

   // Value is null for the base layer or if dependencies are not needed to be
   // described (as may be the case for dynamic SCV codecs). If set, the value
   // contains a list of layers this layer is dependent upon (thus allowing a
   // dependency tree/graph to be created).
   sequence<DOMString>?   layerIdDependencies = null;

   RTCRtpScalabilityType? layerScalabilityType = null;  // null would be for base

   DOMString             receiverId = "";  // use this receiver ID in layer ("" =
N/A)
   unsigned int?         ssrc = null;      // if layer uses its own SSRC (null =
N/A)

   double?               frameRate = null; // framerate for layer (for temporal SVC)
   double?               scale = null;     // scale applied to layer (for spatial
SVC)
   double?               quality = null;   // quality applied to layer (for quality
SVC)

   DOMString             fecReceiverId = ""; // receiver ID for FEC RTP ("" = N/A)
```

```
   unsigned int?          fecSsrc = null;     // using this SSRC for FEC (null = N/A)
   Settings               fec;                // modes of operation related to FEC

   DOMString              rtxReceiverId = ""; // receiver ID for RTX RTP ("" = N/A)
   unsigned int?          rtxSsrc = null;     // using this SSRC for FEC (null = N/A)
   Settings               rtx;                // modes of operation related to RTX
};

enum RTCRtpScalabilityType {
   "temporal",
   "spatial",
   "quality"
};
```

RTCRtpParameterSimulcastDetails (and related)

```
dictionary RTCRtpParameterSimulcastDetails {
   // This sequence contains the details of each simulcasted stream when simulcasting
   // is used or will contain exactly 1 video stream details when not simulcasting.
   sequence<RTCRtpParameterVideoDetails>? simulcastStreams;
};
```

RTCRtpCodec Dictionary Tweak

```
dictionary RTCRtpCodec {
    DOMString     name = "";

    // Added to be able to pick payload type based upon sender or receiver so they
match
    // when creating both the sender and receiver parameters.
    unsigned byte preferredPayloadType;

    unsigned int? clockRate = null;
    unsigned int? numChannels = 1;
    Capabilities  formats;
};
```