Justin Uberti
Version 0.3
Dec 14, 2011

A proposal for http://dev.w3.org/2011/webrtc/editor/webrtc.html

5 The data stream

In addition to the MediaStreams defined earlier in this document, here we introduce the concept of DataStreams for PeerConnection. DataStreams are bidirectional p2p channels for real-time exchange of arbitrary application data in the form of datagrams. DataStreams can either be reliable, like TCP, or unreliable, like UDP, and have built-in congestion control, using a TCP-fair congestion control algorithm.

DataStreams are created via the new PeerConnection.createDataStream method. This method creates a new DataStream object, with specified "label" and "reliable" attributes; these attributes can not be changed after creation. DataStreams can then be added to a PeerConnection, much in the same way that MediaStreams are added; like MediaStreams, DataStreams are unidirectional, either send or receive. Since the semantics of the existing addStream API don't fit perfectly here (i.e. MediaStreamHints), we add the new addDataStream and removeDataStream APIs for this purpose. As with addStream/removeStream, these APIs update the internal session description of the PeerConnection, and cause a new offer to be generated and signaled through the PeerConnection signaling callback. Note that there is no requirement to add a MediaStream first before adding a DataStream; rather, it is expected that many uses of PeerConnection will be solely for application data exchange.

Like MediaStreams, multiple DataStreams can be multiplexed over a single PeerConnection. Each DataStream has a priority, which indicates what preference should be given to each DataStream when a flow-control state is entered. DataStreams with the highest priority are given the first notification and ability to send when flow control lifts. This flow control mechanism is enforced across both DataStreams and MediaStreams **(details TBD).**

In reliable mode, in-order delivery of messaging is guaranteed, which implies head-of-line blocking. In unreliable mode, messages may arrive out of order. Meta-information, including sequence number and timestamp, is provided to allow the application to decide how to handle lost or out of order messages.

There is no maximum size to a datagram that can be sent over the data stream. However, messages are not interleaved on the wire, so a very large message will prevent other messages from being sent until its own send completes. **TBD: should this be true for unreliable streams, or should they have a MTU?**

Encryption of the data stream is required. It is expected that applications that support DataStreams will support DTLS and DTLS-SRTP; while SDES-SRTP, or plain old RTP may be supported for legacy compatibility, there is no need to support DataStreams in these scenarios.

In this draft, there is no inheritance relationship between MediaStream and DataStream, which is intentional due to the lack of a "is-a" relationship. However, to allow the existing stream-oriented APIs on PeerConnection to also work for DataStreams (e.g. localStreams, remoteStreams, onaddstream, onremovestream), we use the proposed Stream base class (see the Streams API proposal) as a common ancestor. This eliminates the need for "data" variants of all the aforementioned functions. **TBD: what should Stream.type be for DataStreams?**

5.1 Changes to PeerConnection

```
interface PeerConnection {
   [...]
   // Adds a datastream to this PeerConnection. Will trigger new signaling.
   void addDataStream (in DataStream stream);
   // Removes a datastream from this PeerConnection. Will trigger new signaling.
   void removeDataStream (in DataStream stream);
   readonly attribute Stream[]  localStreams;
   readonly attribute Stream[]  remoteStreams;
   [...] };
```

5.2 The DataStream interface

See the Streams API document for the definition of the Stream interface

```
[Constructor(DOMString label, boolean reliable)]
interface DataStream : Stream {
   // Label for identifying the stream to the application, as in MediaStream.
   readonly attribute DOMString label;
   // Whether this stream has been configured as reliable.
   readonly attribute boolean reliable;
   // The relative priority of this stream.
   // If bandwidth is limited, higher priority streams get preference.
   // Default priority is zero.
   attribute long priority;

   // States, as in MediaStream and WebSockets
   const unsigned short LIVE = 1;
   const unsigned short ENDED = 2;
   readonly attribute unsigned short readyState;
   attribute Function onReadyStateChange;
   [TBD: replace with onopen/onclose?]

  // Like WebSockets, indicates the amount of buffered data.
  readonly attribute unsigned long bufferedAmount;

   // Sends the supplied datagram.
   // Returns a nonnegative message id if the send was successful.
   // Returns -1 if the stream is flow-controlled.
   long send(in DOMString message);
   [TBD: support ArrayBuffer/Blob, like WebSockets?]

   // Called when a message is received.
   // Arguments: DOMString message
   //            object metadata (with seqnum, timestamp)
   attribute Function onmessage;
   // Called when flow control lifts for this stream.
   // Arguments: None
   attribute Function onreadytosend;
}
```

5.3 Example

```
// standard setup from existing example
var local = new PeerConnection('TURNS example.net', sendSignalingChannel);

// create and attach a data stream
var aLocalDataStream = new DataStream("myChannel", false);
local.addDataStream(aLocalDataStream);
```

```
 // outgoing SDP is dispatched, including a media block like:
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data

 // this SDP is plugged into the remote onSignalingMessage, firing onAddStream
 [remote] onAddStream(aRemoteDataStream);

 // signaling completes, and the data stream goes active on both sides
 [local] onReadyToSend();
 [remote] onReadyToSend();

// we start sending data on the data stream
 var id = aLocalDataStream.send("foo");

 // the message is delivered
 [remote] onMessage("foo", { seqnum: 1, timestamp: <send time> } );

 // the data stream is discarded
 local.removeDataStream(aLocalDataStream)

 // new signaling is generated, resulting in onRemoveStream for the remote
 [remote] onRemoveStream(aRemoteDataStream);
```

5.4 Implementation notes

It is intended that this API map to the wire protocol and layering being defined in the IETF RTCWEB WG for the data channel. One current proposal for said protocol is http://www.ietf.org/id/draft-jesup-rtcweb-data-00.txt, which is believed to match the requirements of this API.

5.5 Security considerations

Protection of data and safe usage by untrusted web pages are requirements for this API.

Protection of data will be accomplished by use of TLS/DTLS encryption in the implementation, using self-signed certificates and fingerprint verification using the PeerConnection (presumably secure) signaling channel.

Protection against untrusted web pages falls into two categories: prevention of unauthorized access, typically on an intranet, and prevention of DoS.

Unauthorized access will be prevented by the security mechanisms associated with the ICE subsystem that PeerConnection runs on top of. Applications will not be allowed to send or receive data on DataStreams until they complete an ICE connectivity check.

DoS is an interesting problem. Presumably, a web page, or IFRAME embedded in a page, will be able to use this API to connect to a compatible endpoint and start exchanging large amounts of data. Web pages already have the ability to do this with XmlHttpRequest or WebSockets, although in those cases, the web application provider has to pay for any bandwidth that is used. More consideration is needed to determine if this is a real problem and what mitigation options make sense.

5.6 Open Issues

- What should Stream.type be for DataStreams?

- How should we handle fragmentation, for unreliable streams? Should we perform reassembly at the received side, or enforce a MTU at the sender side?
  [Suggestion: enforce MTU]
- How shall we integrate MediaStream and DataStream flow control?
  [TBD]
- How shall we we support synchronization of data with audio and video?
  [Each received message has a playout timestamp, but clock sync is needed]
- Should successful or failed delivery result in a callback to the sending application?
  [No use case currently requires this, so this has been left out for now.]

Document History

0.3 Changes for Dec 2011 W3C call; Stream base class; DataStreams are unidirectional; removed onSendResult, enumerated open issues.
0.2 Changes from TPAC feedback; added bufferedAmount, DataStream ctor, security section.
0.1 Initial version.