Justin Uberti
Version 0.4
Jan 26, 2011

A proposal for http://www.w3.org/TR/webrtc/, section 5

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 unidirectional 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.addDataStream method. This method creates a new DataStream object, attached to the PeerConnection, with specified "label" and "reliable" attributes; these attributes can not be changed after creation. The existing PeerConnection.addStream API is not used here, because addStream, with its MediaStreamHints argument, is somewhat media-specific, and there is no clear need for a DataStream to exist separate from a PeerConnection.

Like MediaStreams, DataStreams are unidirectional, either send or receive, and multiple DataStreams can be multiplexed over a single PeerConnection, which is particularly useful in a multiparty scenario. To facilitate this, when a DataStream is added or removed, the local session description of the PeerConnection is updated, resulting in a new offer being 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.

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.

In reliable mode, 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. In unreliable mode, to prevent fragmentation, a maximum datagram size is enforced, and exposed through the maxMessageSize property on the DataStream.

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 clear "is-a" relationship (it's not clear how record() or tracks() are meaningful for a DataStream). However, given the many similarities, it makes sense to allow the existing stream-oriented APIs on PeerConnection (except for addStream) to also work for DataStreams (specifically, remoteStream, localStreams, remoteStreams, onaddstream, and onremovestream). To make this work, we introduce a BaseStream class as a common ancestor. This eliminates the need for "data" variants of all the aforementioned functions.

5.1 Changes to PeerConnection

```
interface PeerConnection {
   [...]
   // Adds a data stream to this PeerConnection. Will trigger new signaling.
   DataStream addDataStream(DOMString label, boolean reliable);
   // Removes a media or data stream from this PeerConnection.
   // Will trigger new signaling.
   void removeStream (in BaseStream stream);
   readonly attribute BaseStream[]  localStreams;
   readonly attribute BaseStream[]  remoteStreams;
   [...]};
```

5.2 The DataStream interface

```
interface BaseStream {
   // Type of stream, either "media" or "data".
   readonly attribute DOMString type;
};

interface DataStream : BaseStream {
   [NoConstructor]
   // 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 onended;

   // Indicates the amount of buffered data, as in WebSockets.
   // Only applicable in reliable mode.
   readonly attribute unsigned long bufferedAmount;

   // Indicates the maximum size, in bytes, of outbound messages.
   // Only applicable in unreliable mode.
   readonly attribute unsigned long maxMessageSize;

   // Sends the supplied datagram, as in WebSockets.
   // Returns a nonnegative message id if the message can be sent.
   // Throws a TBD exception if the message is too large.
   long send(in DOMString message);
```

```
    long send(in ArrayBuffer data);

    // Called when a message is received, as in WebSockets.
    // Arguments: DOMString/ArrayBuffer message (depending on type of message)
    //            object metadata (with seqnum, timestamp)
    attribute Function onmessage;}
```

5.3 Examples

5.3.1 Simple unreliable example

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

// create and attach an unreliable data stream
var aLocalDataStream = local.addDataStream("myChannel", false);

// outgoing SDP is dispatched, including a media block like:
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data
a=datachannel:1 label:myChannel

// this SDP is plugged into the remote onSignalingMessage, firing onAddStream
// of a new unreliable DataStream with label "myChannel"
[remote] onAddStream(aRemoteDataStream);

// signaling completes, and the data stream goes active on both sides;
// we start sending data on the data stream
aLocalDataStream.send("foo");  // sends with seqnum S, timestamp T0

// the message is delivered
[remote] onmessage("foo", { seqnum: S, timestamp: T0 } );

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

// new signaling is generated
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data

// resulting in onRemoveStream for the remote
[remote] onremovestream(aRemoteDataStream);
```

5.3.2 Simple reliable example

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

// create and attach a reliable data stream
var aLocalDataStream = local.addDataStream("myChannel", true);

// outgoing SDP is dispatched, including a media block like:
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data
a=datachannel:1 label:myChannel reliable

// this SDP is plugged into the remote onSignalingMessage, firing onAddStream
// of a new reliable DataStream with label "myChannel"
[remote] onAddStream(aRemoteDataStream);

// signaling completes, and the data stream goes active on both sides;
```

```
// send the state of an input field every 100 ms to the remote side, unless
// we're flow-controlled
setInterval(function() {
  if (aLocalDataStream.bufferedAmount == 0) {
    aLocalDataStream.send(inputElement.value);
  }
}, 100);

// the messages are delivered; seqnum is not filled in for reliable streams
[remote] onmessage("", { timestamp: T0 } );
[remote] onmessage("h", { timestamp: T0 + 100 } );
[remote] onmessage("he", { timestamp: T0 + 200 } );
[remote] onmessage("hel", { timestamp: T0 + 300 } );
[remote] onmessage("hell", { timestamp: T0 + 400 } );
[remote] onmessage("hello", { timestamp: T0 + 500 } );

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

// new signaling is generated
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data

// resulting in onRemoveStream for the remote
[remote] onremovestream(aRemoteDataStream);
```

5.3.3 Multiparty unreliable example

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

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

// outgoing SDP is dispatched, including a media block like:
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data
a=datachannel:1 label:myChannel

// remote side later replies with its own SDP
m=application 49200 <TBD> 127
a=rtpmap:127 application/html-peer-connection-data
a=datachannel:2 label:BsChannel
a=datachannel:3 label:CsChannel

// this results in onaddstream being fired for each data stream
onaddstream(remoteDataStreamB);
onaddstream(remoteDataStreamC);

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

// the message is delivered to B and C
[remote-B] onmessage("foo", { seqnum: S1, timestamp: T0 } );
[remote-C] onmessage("foo", { seqnum: S2, timestamp: T0 } );
```

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-01.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.

Open Issues

- How should we integrate MediaStream and DataStream flow control?
  [TBD]
- How should we we support synchronization of data with audio and video?
  [Each received message has a playout timestamp, but clock sync is needed. Perhaps this is an argument for a DataStream being a track in an existing MediaStream?]
- 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.]

Informative References

- The WebSocket API, http://dev.w3.org/html5/websockets/

Document History

0.4: Changes for Jan 2012 W3C meeting; tied DataStreams to a PeerConnection; changed DataStream base class; added MTU for unreliable streams; better WebSocket alignment (added other send() variants, removed onreadytosend); added reliable and multiparty examples
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.