WebRTC data channels

WebRTC data channels for high performance data exchange

By Dan Ristic

Published: February 4th, 2014

Updated: February 4th, 2014

Comments: 2

Sending data between two browsers for communication, gaming, or file transfer can be a rather involved process. It requires setting up and paying for a server to relay data, and perhaps scaling this to multiple data centers. In this scenario there is potential for high latency, and it's difficult to keep data private.

These problems can be alleviated by using WebRTC's RTCDataChannel API to transfer data directly from one peer to another. In this article we will cover the basics of how to set up and use data channels, as well as the common use cases on the web today.

Tip!

To make the most of this article, you will need to have some knowledge of the RTCPeerConnection API, and an understanding of how STUN, TURN, and signaling work. We recommend you take a look at Getting Started With WebRTC.

Why another data channel?

We have WebSocket, AJAX and Server Sent Events. Why do we need another communication channel? WebSocket is bidirectional, but all these technologies are designed for communication to or from a server.

RTCDataChannel takes a different approach:

- It works with the RTCPeerConnection API, which enables peer to peer connectivity. This can result in lower latency: no intermediary server, fewer 'hops'.

- RTCDataChannel uses Stream Control Transmission Protocol (SCTP), allowing configurable delivery semantics: out-of-order delivery and retransmit configuration.

RTCDataChannel is available now, with SCTP support, in Chrome, Opera and Firefox for desktop and Android.

A caveat: signaling, STUN and TURN

Although WebRTC enables peer-to-peer communication, it still needs servers:

- **For signaling:** to enable the exchange of media and network metadata to bootstrap a peer connection.
- **To cope with NATs and firewalls**: by using the ICE framework to establish the best possible network path between peers, by working with STUN servers (to ascertain a publicly accessible IP and port for each peer) and TURN servers (if direct connection fails and data relaying is required).

The HTML5 Rocks article WebRTC in the real world: STUN, TURN, and signaling explains in detail how WebRTC works with servers for signaling and networking.

The capabilities

The RTCDataChannel API supports a flexible set of data types. The API is designed to mimic WebSocket exactly, and RTCDataChannel supports strings as well as some of the binary types in JavaScript such as Blob, ArrayBuffer and ArrayBufferView. These types can be helpful when working with file transfer and multiplayer gaming.

	TCP	**UDP**	**SCTP**
Reliability	reliable	unreliable	configurable
Delivery	ordered	unordered	configurable
Transmission	byte-oriented	message-oriented	message-oriented
Flow control	yes	no	yes
Congestion control	yes	no	yes

(From High Performance Browser Networking by Ilya Grigorik.)

RTCDataChannel can work in either unreliable mode (analogous to User Datagram Protocol or UDP) or reliable mode (analogous to Transmission Control Protocol or TCP). The two modes have a simple distinction:

- **Reliable mode** guarantees the transmission of messages and also the order in which they are delivered. This takes extra overhead, thus potentially making this mode slower.
- **Unreliable mode** does not guarantee every message will get to the other side nor what order they get there. This removes the overhead, allowing this mode to work much faster.

Performance for both modes is about the same when there are no packet losses. However, in reliable mode a lost packet will cause other packets to get blocked behind it, and the lost packet might be stale by the time it is retransmitted and arrives. It is, of course, possible to use multiple data channels within the same application, each with their own (un)reliable semantics.

We show below how to configure RTCDataChannel to use reliable or unreliable mode.

Configuring data channels

There are several simple demos of RTCDataChannel online:

- simpl.info/dc
- Transmit text
- Transfer a file
- pubnub.github.io/webrtc (between two PubNub clients)

In these examples the browser is making a peer connection to itself. It is then creating a data channel and sending the message along the peer connection. Finally, your message appears in the box on the other side of the page!

The code to get started with this is short:

```
var peerConnection = new RTCPeerConnection();

// Establish your peer connection using your signaling channel
```

```
here
var dataChannel =
  peerConnection.createDataChannel("myLabel",
dataChannelOptions);

dataChannel.onerror = function (error) {
  console.log("Data Channel Error:", error);
};

dataChannel.onmessage = function (event) {
  console.log("Got Data Channel Message:", event.data);
};

dataChannel.onopen = function () {
  dataChannel.send("Hello World!");
};

dataChannel.onclose = function () {
  console.log("The Data Channel is Closed");
};
```

The `dataChannel` object is created from an already established peer connection. It can be created before or after signaling happens. You then pass in a label to distinguish this channel from others and a set of optional configuration settings:

```
var dataChannelOptions = {
  ordered: false, // do not guarantee order
  maxRetransmitTime: 3000, // in milliseconds
};
```

It is also possible to add a `maxRetransmits` option (the number of times to try before failing) but you can only specify maxRetransmits or maxRetransmitTime, not both. For UDP semantics, set `maxRetransmits` to 0 and `ordered` to `false`. For more information, see RFC 4960 (SCTP) and RFC 3758 (SCTP partial reliability).

- **ordered:** If the data channel should guarantee order or not
- **maxRetransmitTime:** The maximum time to try and retransmit a failed message (forces unreliable mode)
- **maxRetransmits:** The maximum number of times to try and retransmit a failed message (forces unreliable mode)
- **protocol:** Allows a subprotocol to be used, but will fail if the specified protocol

is unsupported
- **negotiated:** If set to true, it removes the automatic setting up of a data channel on the other peer, meaning that you are provided your own way to create a data channel with the same id on the other side
- **id:** Allows you to provide your own ID for the channel

The only options most people will need to use are the first three: `ordered`, `maxRetransmitTime`, and `maxRetransmits`. With SCTP (now used by all browsers that support WebRTC) reliable and ordered is true by default. It makes sense to use full unreliable if you want full control from the application layer, but in most cases, partially reliable is helpful.

Note that, as with WebSocket, RTCDataChannel fires events when a connection is established, closed, or errors, and when it receives a message from the other peer.

Is it safe?

Encryption is mandatory for all WebRTC components. With RTCDataChannel all data is secured with Datagram Transport Layer Security (DTLS). DTLS is a derivative of SSL, meaning your data will be as secure as using any standard SSL based connection. DTLS is standardized and built in to all browsers that support WebRTC. You can find more information about DTLS on the Wireshark wiki.

Change how you think about data

Handling large amounts of data can be a pain point in JavaScript. As the developers of Sharefest have pointed out, this requires thinking about data in a new way. If you are transferring a file that is larger than the amount of memory you have available you have to think about new ways to save this information. This is where technologies such as the FileSystem API come into play, as we show below.

Build a file sharing application

Creating a web application that can share files in the browser is now possible with RTCDataChannel. Building on top of RTCDataChannel means that the transferred file data is encrypted and does not touch an application provider's servers. This functionality, combined with the possibility of connecting to multiple clients for faster sharing, makes WebRTC file sharing a strong candidate for the web.

Several steps are required to make a successful transfer:

1. Read a file in JavaScript using the File API
2. Make a peer connection between clients, using RTCPeerConnection.
3. Create a data channel between clients, using RTCDataChannel.

There are several points to consider when trying to send files over RTCDataChannel:

- **File size:** if file size is reasonably small and can be stored and loaded as one Blob, you can load into memory using the File API and then send the file over a reliable channel as is (though bear in mind that browsers impose limits on maximum transfer size). As file size get larger, things get trickier. A chunking mechanism is required: file chunks are loaded and sent to another peer, accompanied with chunkId metadata so the peer can recognize them. Note that in this case you will also need to save the chunks first to offline storage (for example, using the FileSystem API) and only when you have the file in its entirety save it to the user's disk.
- **Speed:** it is debatable whether unreliable (UDP-like) or reliable (TCP-like) transport is better for the job of file transfer. If the application is a simple one-to-one file transfer, going with unreliable data channels will take some design effort for the ACK/retransmit protocol. You will need to implement this yourself and — even if you're good — it may not be much faster than using reliable transport. Reliable and unordered should provide the best of both worlds, but results may vary when considering multi-party (mesh) file transfer.
- **Chunk size:** these are the smallest 'atoms' of data for your application. Chunking is required, since there is currently a send size limit (though this will be fixed in a future version of data channels). The current recommendation for maximum chunk size is 16KB.

Once the file is fully transferred to the other side, it can be downloaded using an anchor tag:

```
function saveFile(blob) {
  var link = document.createElement('a');
  link.href = window.URL.createObjectURL(blob);
  link.download = 'File Name';
  link.click();
};
```

The file sharing apps at pubnub.github.io/rtc-pubnub-fileshare and github.com/Peer5/ShareFest use this technique; they are both open source and provide a good foundation for a file sharing application based on RTCDataChannel.

So what can we do?

RTCDataChannel opens the doors to new ways to build apps for file sharing, multiplayer gaming, and content delivery.

- Peer to peer file sharing as described above.
- Multiplayer gaming, paired with other technologies like WebGL, as seen in Mozilla's Banana Bread.
- Content delivery: as being reinvented by PeerCDN, a framework that delivers web assets through peer to peer data communication.

Change the way you build applications

We can now provide more engaging applications by using high performance, low latency connections via RTCDataChannel. Frameworks such as PeerJS and the PubNub WebRTC SDK make RTCDataChannel easier to implement, and the API now has wide support across platforms.

The advent of RTCDataChannel can change the way we think about data transfer in the browser.

Find out more

- Getting started with WebRTC
- WebRTC in the real world: STUN, TURN and signaling
- WebRTC resources
- W3C Working Draft
- IETF WebRTC Data Channel Protocol Draft
- How to send a File Using WebRTC Data API
- 7 Creative Uses of WebRTC's Data Channel
- Banana Bread 3D first person shooter game compiled to JS+WebGL, using WebRTC data channels in multiplayer mode

Thanks to Hadar Weiss and Shachar Zohar for their help in preparing this