Hans-Christer Holmberg

Web Real-Time Data Transport

WebRTC Data Channels

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

16 April 2015

Author(s) Title Number of Pages Date	Hans-Christer Holmberg Web real-time data transport: WebRTC data channels 55 pages + 2 appendices 16 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	
Instructor(s)	Salvatore Loreto, Strategic Product Manager Jarkko Vuori, Principal Lecturer

Recently, a new standard, WebRTC, has been published, which defines a standardized mechanism for establishing direct data connections between browsers without the need for a 3rd party plug-in application. The WebRTC data channel mechanism can be used to send arbitrary data between browsers. The purpose of the final year project was to study and describe the protocols and interfaces used to realize WebRTC data channels, and to describe how to implement applications using WebRTC data channels. As part of the final year project, demo applications using WebRTC data channels have been implemented, in order to test whether the mechanism can be used for exchanging time sensitive data and exchanging large data between browsers. Both the client side and the server side applications were written using the JavaScript programming language.

Keywords	HTML5, WebRTC, RTCWEB, data channel, browser, JavaScript

Contents

1 Introduction

1.1 Background

Modern web applications often rely on frequent data to be exchanged either between the browser and the web server, or between two browsers. There are a number of mechanisms for data to be exchanged between a browser and a web server without having to re-load a complete web page. Legacy mechanisms rely on transporting data in HTTP messages while the latest mechanism, WebSockets, is using a dedicated bi-directional channel for transporting data [1].

Previously, in order to exchange data between two browsers, there were two options: to transport the data via a web server or to download and install a 3rd party plug-in application which allowed sending data directly between the browsers. This thesis describes a new standard mechanism, defined and standardized by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) standard organizations, which allows sending data directly between two browsers without the need for a 3rd party plug-in. The mechanism consists of two parts: an application user interface (API), WebRTC, which web application developers can use to enable the mechanism and a transport mechanism, WebRTC data channel, for delivering the data between the browsers [2;3]. The author of this thesis has been an active participant in the standardization work.

While the WebRTC API is defined for browsers, the transport mechanism can be used by any type of client. This thesis focuses on browsers.

The thesis discusses the protocol mechanisms used to realize the WebRTC data channel and how the WebRTC API can be used to implement web applications using a data channel. The thesis also contains the source code for a couple of example mechanisms, both using a data channel. Section 1.2 discusses some of the HTML5 features that are expected to be used by WebRTC applications.

1.2 HTML5

1.2.1 General

HTML5 is the fifth version of the HTML standard, defined by the World Wide Web Consortium (W3C). The final revision of the HTML5 standard was released in 2014. [4.]

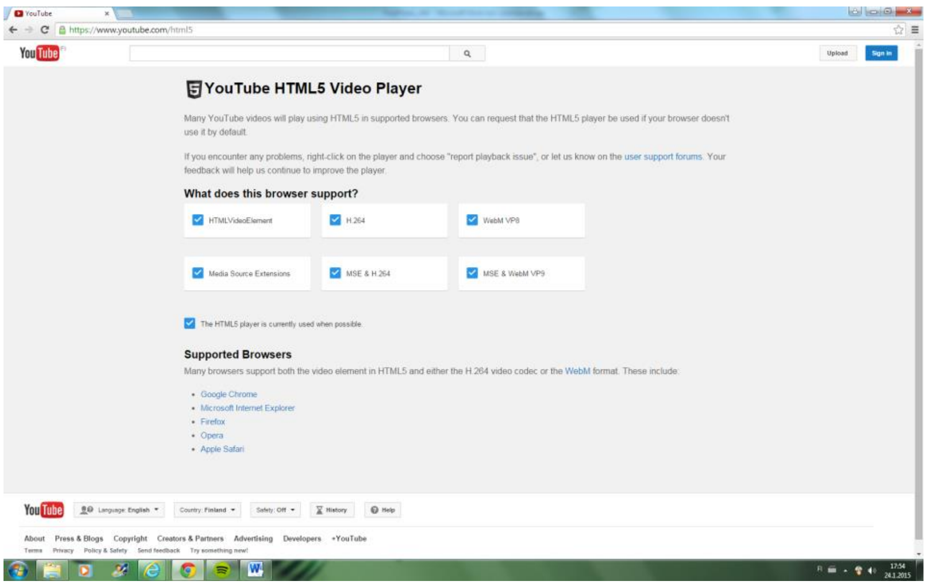
HTML5 introduces several new features needed in order to produce dynamic and rich web applications. In addition, mobile devices and devices with limited memory and CPU resources were taken into consideration when defining HTML5. HTML5 is designed to be backward compatible with older browsers that only support older versions of HTML, so that such browsers will simply ignore any HTML5 element. The previous version, HTML4, was released in 1997.

Some of the most important HTML5 features and the associated APIs are described below. As will be seen, they remove many of the limitations that were associated with web applications before HTML5, which has resulted in a huge number of HTML5 based applications being produced.

The HTML5 mechanisms for transporting data between a browser and a web server are described in section 2.

1.2.2 Multimedia

The HTML5 <*audio*> and <*video*> elements represent two of the most important features of HTML5. They enable usage of audio and video components in web pages, without the need for 3rd party plugins. When HTML5 was standardized, the parties involved in the standardization work could not agree on a mandatory video codec for HTML5. Figure 1 shows a screenshot of the HTML5 version of the YouTube video player.



YouTube HTML5 Video Player

Many YouTube videos will play using HTML5 in supported browsers. You can request that the HTML5 player be used if your browser doesn't use it by default.

If you encounter any problems, right-click on the player and choose "report playback issue", or let us know on the user support forums. Your feedback will help us continue to improve the player.

What does this browser support?

✓ HTMLVideoElement
✓ H.264
✓ WebM VP8
✓ Media Source Extensions
✓ MSE & H.264
✓ MSE & WebM VP9

✓ The HTML5 player is currently used when possible.

Supported Browsers

Many browsers support both the video element in HTML5 and either the H.264 video codec or the WebM format. These include:

- Google Chrome
- Microsoft Internet Explorer
- Firefox
- Opera
- Apple Safari

Language: English
Country: Finland
Safety: Off
History
Help
About Press & Blogs Copyright Creators & Partners Advertising Developers +YouTube
Terms Privacy Policy & Safety Send feedback Try something new!
17:54
24.1.2015

Figure 1: Screenshot of the HTML5 support in YouTube [5].

The getUserMedia API allows web applications to get access to media streams (video and audio) generated locally (for example by a microphone or camera). Locally generated media can then be sent to another browser (using the WebRTC mechanism).

1.2.3 Graphics

The Canvas API allows a user to draw graphics on a web page. The API also allows an application to retrieve information about individual pixels, which can be used to manipulate photos and videos shown on the web page.

The Canvas API has drawn a lot of attention, especially in the game industry and the number of games implemented using the API is constantly growing. In the Pong game, presented in the end of this thesis, the graphics are implemented using the Canvas API. Figure 2 shows a screenshot the HTML5 version of the Angry Birds game.

Figure 2: Screenshot of the HTML5 Canvas version of Angry Birds [6]

1.2.4 Client-Side Data Storage

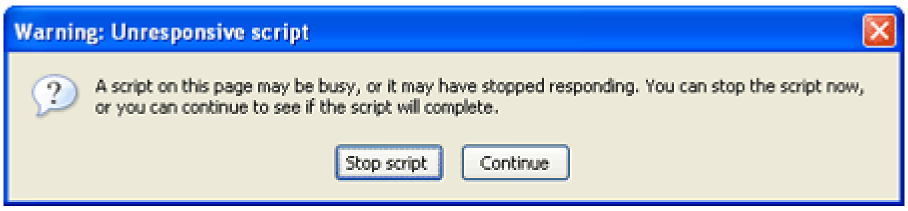
Web applications do not have access to the host file system. In addition, before HTML5, the only standardized, browser-independent and plugin-free mechanism for a web application to store local data was by using cookies. However, the amount of data that can be stored using cookies is very limited and the data is included in each HTTP request sent between the browser and the web server. [7.]

HTML5 defines two APIs for storing data locally: the Web Storage API and the File API. The Web Storage API allows web applications to store large amounts of data locally and the data is never sent to the server (unless the web application explicitly chooses to do so). The API provides two types of storages: session and local. Local data is deleted when a session is terminated (when the browser, or the browser tab, is closed), while the session data remains until it is explicitly deleted by a web application. The stored data is tied to the origin of the web application storing the data, meaning that each web application sharing the same origin will be able to access the same data. In order for the origin to be the same, the domain name, the port number and the protocol (http versus https) must match. [8;9.]

The File API provides access to the local file system. A web application can use the API to read file contents. If the file is large, the API supports file splicing, meaning that only parts of the file can be read (and stored in memory) at any given time. The API does not allow a web application to write to the file system.

1.2.5 Web Workers

Even though CPUs and JavaScript engines are becoming faster and more powerful, JavaScript is still a single-thread programming language and heavy or time consuming functions may have bad impact on the user experience. Figure 3 shows a screenshot of a warning message when the JavaScript processing takes a long time. [10.]


Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, or you can continue to see if the script will complete.

Figure 3: Screenshot of Windows script warning [11]

The Web Worker API enables running JavaScript in the background, in a separate thread, without impacting the parent script and the user interactions. A worker can send data to the parent script posting messages for which event handler functions have been defined. A web worker can also trigger a web worker itself. [12.]

There are two types of web workers: dedicated workers and shared workers. Dedicated workers can only be accessed from the application script that created the worker, while a shared worker can be accessed from multiple scripts.

1.3 WebRTC

WebRTC (RTC being an abbreviation for Real-Time Communications) is a framework which allows direct bi-directional real-time transport of audio, video and data between two browsers, without the need for a 3rd party plug-in.

WebRTC consists of a JavaScript API and mechanisms for transporting media and data between the clients. It also includes tools for Network Address Translation (NAT) traversal and security. NAT traversal refers to mechanisms used to ensure that Internet Protocol (IP) based traffic can pass through NAT gateways, which are commonly used to alter address and port information associated with IP packets [13]. The transported media and data is always encrypted: audio and video streams are encrypted using Secure RTP (SRTP), while data is encrypted using Datagram Transport Layer Security (DTLS) [14;15]. DTLS is based on the Transport Layer Security (TLS) protocol and is designed for unreliable transport-layer protocols [16]. For audio and video, while SRTP is used to encrypt the media, DTLS is used for the key management. This allows the usage of a single key management mechanism for the encryption of both media and data.

WebRTC supports multiplexing of media and data, meaning that all (or parts of) media and data sent between two browsers can be sent using a single 5-tuple. The usage of multiplexing makes NAT traversal easier and also saves DTLS resources.

WebRTC uses the ICE framework to perform NAT traversal. This is a very essential feature, as browsers are often located between NATs.

WebRTC can be used to implement a number of different applications from video conference applications to file transfer applications and multi-user games where data is sent between the participants.

2 Client to Server Data Transport

2.1 General

The Hypertext Transfer Protocol (HTTP) was initially designed for performing simply client/server transactions and to download static pages from a web server to a client. The protocol was not designed for transporting frequent data between the client and the server. In addition, HTTP requests are always initiated by the client, so in order for the server to send data to the client it needs to receive an HTTP request and then insert the data in the associated HTTP response. [17.]

Later, when the content of the web pages became more dynamic and people started to talk about web applications, mechanisms were designed which allowed clients to download information during a session, without having to re-download the whole webpage. [18.]

Before HTML5, the mechanisms for transporting data between a client and a web server were based on the usage of the XMLHttpRequest API. The XMLHttpRequest API enables a browser to send an asynchronous HTTP request to a web server in the background, without blocking the user interface. The web server can insert data in the associated HTTP response, which the browser can use for example to update parts of the web page. This allows more dynamic web pages, without having to re-load the whole web page every time some data is modified. As the XMLHttpRequest API does not define a new transport mechanism between the browser and the web server, but rather uses standard HTTP requests and responses, the restrictions associated with HTTP still apply: the mechanism is still based on the browser sending an HTTP request and the web server including the data in the associated response. [19.]

Sections 2.2, 2.3, 2.4, 2.5 and 2.6 describe the most common mechanisms based on usage of the XMLHttpRequest API to enable data to be sent from the web server to the client. In addition, the section describes the new HTML5 mechanisms: Server-sent Events (SSE) and WebSocket. The mechanisms have different names, but can be separated into the following main groups: polling, long-polling and streaming. [1;20.]

2.2 Polling

Polling is based on the client sending regular HTTP requests to the server and the server immediately sending the associated HTTP response. Any data that the server needs to transport to the client is included in the HTTP response.

While polling is a simple mechanism and work on most clients and web servers, there are a number of disadvantages. First, whenever an HTTP request is to be sent, a new TCP connection towards the server must be established. The connection is then closed when the client has received the associated HTTP response. Depending on the network load, the TCP connection establishment may also take some time. [21.]

Second, there is no guarantee that HTTP messages are received in the same order as they were sent. Web applications need to take that into consideration for example by only having one outstanding HTTP transaction towards the web server at any given time.

Figure 4 shows an example using polling. The server receives an event with data that is to be transported to the client. When the server receives an HTTP request from the client, it includes the data in the associated HTTP response sent back to the client.

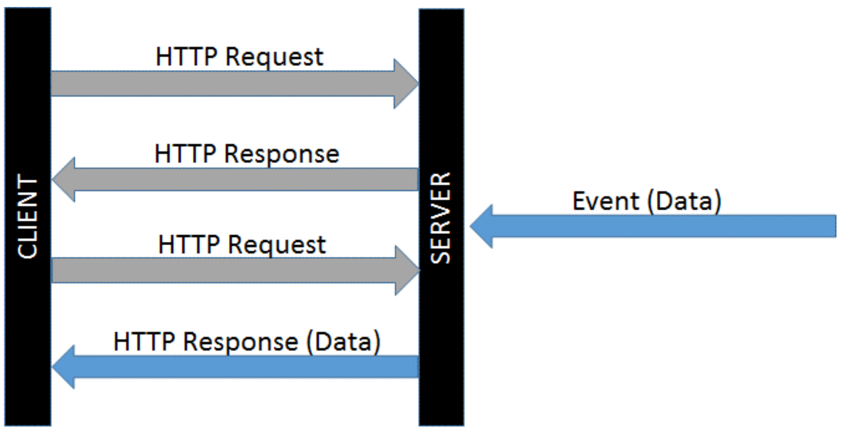


Figure 4: Polling example.

2.3 Long-Polling

Long-polling mechanisms also use HTTP responses for sending data from the web server to the client. The main difference from normal polling is that the server does not immediately send the HTTP response when it receives the request. Instead, the server waits until there is some data to send. Because of this, the TCP connection between the client and the server is kept open for a longer time, removing issues associated with frequent opening and closing of TCP connections. However, if the server needs to send data to the client on a regular basis, the outcome will be similar to polling.

Figure 5 shows an example using long-polling. The server receives an event with data that is to be transported to the client. When the server receives an HTTP request from the client, it does not send the associated HTTP response until it has received data that is to be transported to the client.

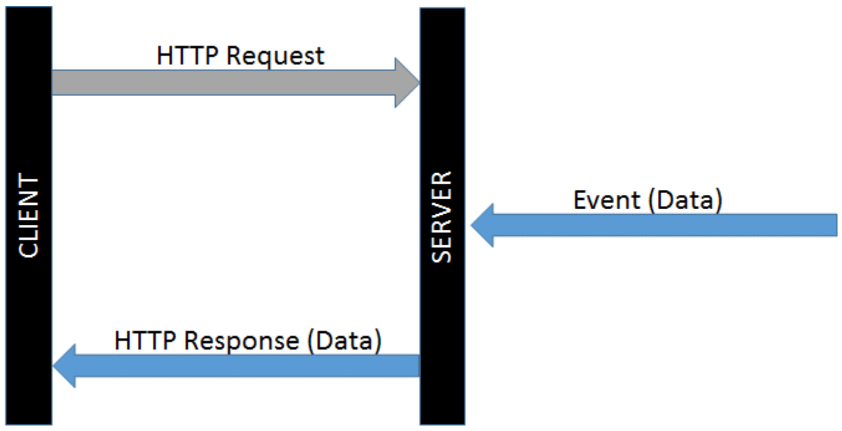


Figure 5: Long-polling example

2.4 Streaming

Streaming is similar to long polling. The main difference is that the TCP connection is never closed, not even after data has been sent by the server. The server sends partial HTTP responses to the client. As the client expects additional partial HTTP responses to arrive, it will not close the TCP connection.

Streaming reduces network latency even further, as the client does not have to open and close TCP connections. However, if the server needs to send data very frequently, the client may not be able to process the partial HTTP responses as fast as they arrive, which may cause data to get lost.

In addition, as for polling and long-polling, using HTTP responses to transport data introduces a considerable amount of overhead, as each HTTP response, in addition to the actual transported data, also contains HTTP headers.

Figure 6 shows an example using streaming.

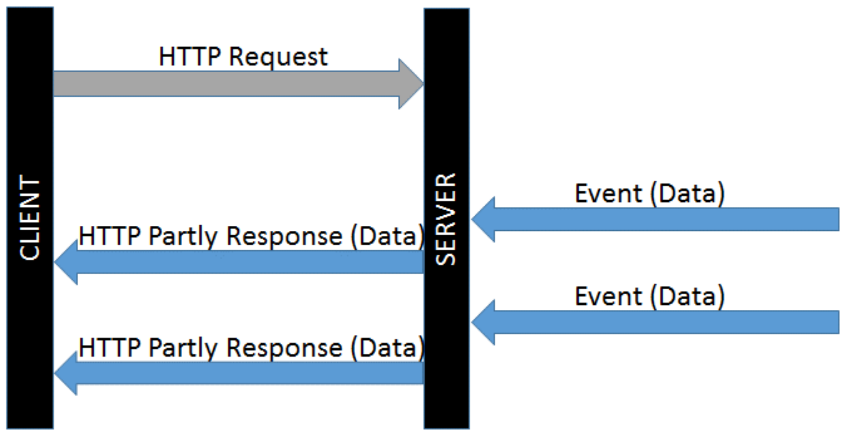


Figure 6: Streaming example

2.5 HTML5 Server-Sent Events (SSE)

SSE is a standardized version of streaming. It is exposed to the developer using a dedicated API, EventSource, which hides for example HTTP details. SSE also defines a format on how data and associated properties, are sent. The SSE mechanism can reuse an existing TCP connection to the server, if available, instead of opening a new one. This is useful, as some client implementations limit the number of simultaneous TCP connections. [20.]

2.6 HTML5 WebSocket

The polling, long-polling and streaming mechanisms all use HTTP responses for transporting data from the server to the client. Thus, they rely on receiving HTTP requests from the client, in order to be able to send responses. Such mechanisms normally work fine, if the data transport interval frequency is known. In addition, the mechanisms assume that the client does not need to send data to the web server very often, meaning the mechanisms do not provide good means for bi-directional transport of data.

The WebSocket mechanism provides a static bi-directional data transport connection between a browser and a web server, where data can be sent in both directions at any given time. HTTP will only be used for negotiating and creating the WebSocket connection, but not for transporting the actual data.

Web application developers can access the WebSocket mechanism using the HTML5 WebSocket API. In order to establish a WebSocket connection, the browser and web server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake. Once the WebSocket connection has been established, data can be transported back and forth between the browser and the server in full-duplex mode at the same time. Data can be sent in text and binary form. [1.]

Figure 7 shows an example of the establishment of a WebSocket connection.

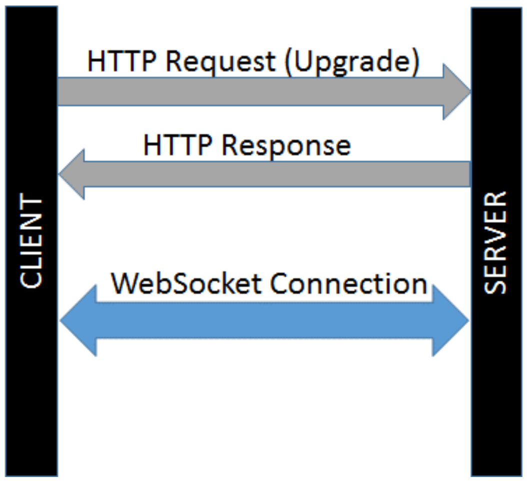


Figure 7: WebSocket example

3 Stream Control Transmission Protocol (SCTP)

3.1 General

SCTP is a transport protocol for IP networks. It provides many features found in other transport protocols, such as UDP and TCP, but also a number of new features. [21;22;23.]

An SCTP association (sometimes also referred to as an SCTP connection) is a logical relationship between two SCTP endpoints. Each endpoint is assigned a single SCTP port value, but may be assigned multiple IP addresses (SCTP multi-homing).

Similar to TCP, SCTP provides ordered and reliable transport of data between two end-points. Similar to UDP, SCTP is message-oriented and therefore preserves message boundaries. In addition, SCTP also supports multi-homing, multi-streaming and graceful shutdown. Multi-homing means that each endpoint of an SCTP association can be associated with multiple network interfaces, using multiple IP addresses and ports. Multi-streaming means that messages can be transported in parallel (on separate SCTP streams), without impacting each other. A graceful shutdown makes sure that any outstanding messages are transported before an SCTP association is closed. Finally, SCTP uses a handshake mechanism based on a cookie exchange for additional security.

3.2 Packet Structure

An SCTP packet contains an SCTP header, followed by one or more SCTP chunks. Figure 8 shows the high level structure of an SCTP packet.

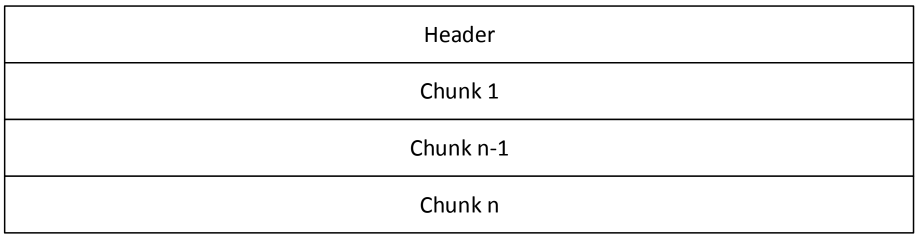


Figure 8: SCTP packet structure

3.2.1 Header

The SCTP header contains port information used to route the SCTP packet. The Verification Tag is a unique value for the SCTP association and is used to detect packets that may be associated with a previous association. Figure 9 shows the SCTP header structure.

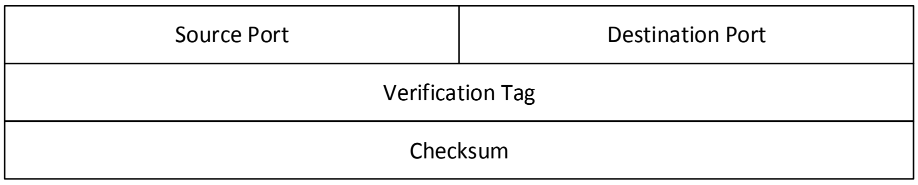


Figure 9: SCTP packet header

3.2.2 Chunk

Within an SCTP packet, a chunk is a unit of information. Chunks contain either user data (DATA chunk) or SCTP control information (control chunk), used for example to initiate and tear down SCTP sessions. Each chunk contains a chunk specific header. Figure 10 shows the general structure of a chunk. The chunk value depends on the chunk type.

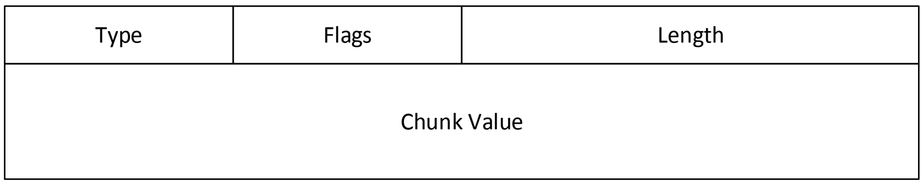


Figure 10: SCTP chunk

The Type parameter contains the value defined for the chunk type. Table 1 lists the names of the basic chunk types and the type value associated with each chunk type.

Table 1: Basic STCP chunk types

Type Value	Name	Usage Description
0	DATA	User data transport.
1	INIT	SCTP association initiation.
2	INIT_ACK	INIT acknowledgement.
3	SACK	Selective acknowledgement of DATA chunks.

4	HEARTBEAT	Heartbeat for testing reachability of the remote SCTP endpoint.
5	HEART-BEAT_ACK	HEARTBEAT acknowledgement.
6	ABORT	Immediate shutdown of an SCTP association.
7	SHUTDOWN	Graceful shutdown of an SCTP association.
8	SHUTDOWN_ACK	SHUTDOWN acknowledgement.
9	ERROR	Reporting an error condition to the remote SCTP endpoint.
10	COOKIE_ECHO	State cookie transport.
11	COOKIE_ACK	COOKIE_ECHO acknowledgement.

3.3 Stream

An SCTP stream is a logical, unidirectional channel within an SCTP association. SCTP user data delivered within a stream is delivered to the receiving SCTP user in order (un-ordered delivery is also supported). Each SCTP stream is identified by a stream identifier value. A stream specific Stream Sequence Number (SSN) value, carried in a chunk header field, is assigned to each SCTP message within the stream. The SSN value is used by the receiving SCTP endpoint to deliver the user data to the SCTP user in order. Figure 11 shows an example of multiple SCTP streams associated with a single SCTP association.

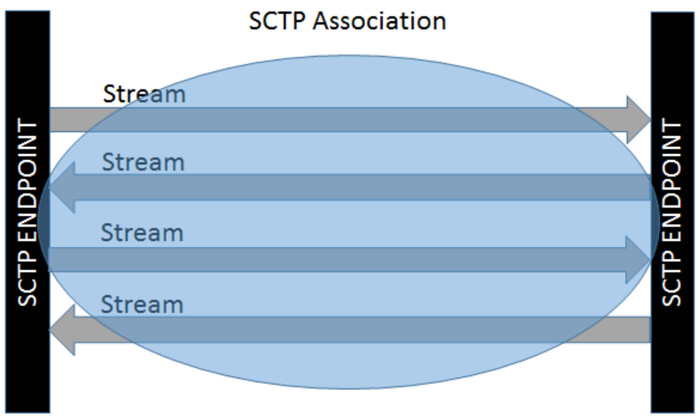


Figure 11: Multiple SCTP streams associated with an SCTP association

SCTP messages associated with a given SCTP stream are in most cases not affected by messages associated with other SCTP streams. For example, if messages associated with one stream are blocked, messages associated with other streams will still be sent. However, if fragmentation of user data occurs, all SCTP messages carrying fragments must be sent in sequence, which means that possible SCTP messages associated with other SCTP streams will be put on hold.

When an SCTP association is established, the SCTP endpoints can negotiate the number of SCTP streams (incoming and outgoing) that they support within the association. Figure 12 shows the location of the stream identifier value within a DATA chunk.

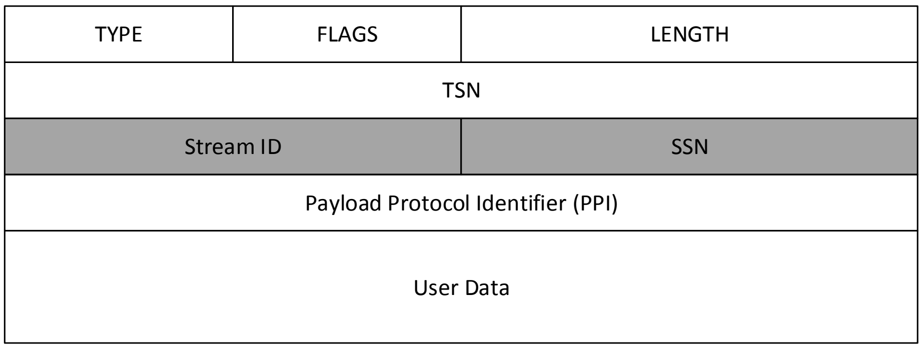


Figure 12: DATA chunk with Stream ID and SSN chunk header fields

3.4 Payload Protocol Identifier (PPID)

Each DATA chunk contains a Payload Protocol Identifier (PPID) value, carried in the Payload Protocol Identifier (PPI) header field, which can be used by the receiving endpoint to de-multiplex incoming messages. Figure 13 shows the position of the PPI value within the DATA chunk header field.

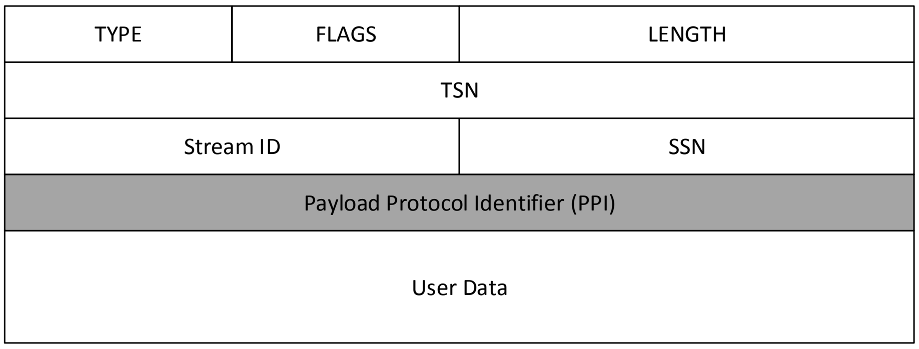


Figure 13: DATA chunk PPI chunk header field

3.5 Fragmentation

In order to ensure that the size of an SCTP message conforms to the Path Maximum Transmission Unit (MTU), the sending SCTP endpoint can fragment user messages and transport the user message fragments within multiple SCTP messages and DATA chunks. The receiving endpoint will reassemble the fragments before the user message is passed to the receiving SCTP user.

When a user message is fragmented into multiple DATA chunks, the sending endpoint assigns the same SSN value to each DATA chunk. The sending endpoint assigns separate Transmission Sequence Number (TSN) values, in a sequence, to each DATA chunk. The sending endpoint uses flag bits to indicate which DATA chunk contains the first fragment and which DATA chunk contains the last fragment.

By default, even if unordered user message delivery is used, a fragmented message will still be reassembled by the receiving endpoint and delivered ordered to the receiving SCTP user. However, if the receiving SCTP endpoint runs out of buffer space for reassembling a complete user message, it may pass partially reassembled messages to the SCTP user.

Figure 14 shows an example, where a single user message is fragmented and transported between two SCTP endpoints using multiple DATA chunks. Each DATA chunk carries one fragment.

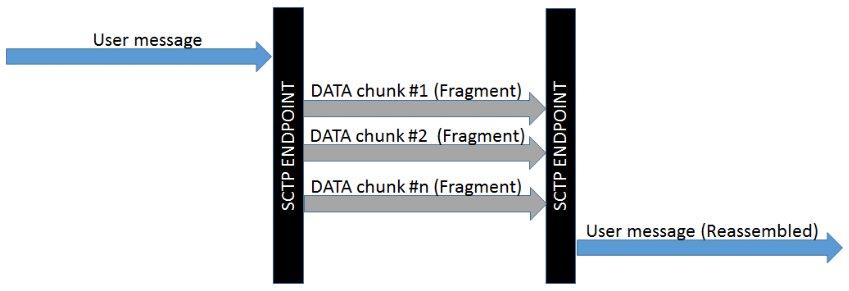


Figure 14: SCTP fragmentation example

3.6 Procedures

3.6.1 Establish Association

In order for an SCTP endpoint (the opening endpoint) to establish an SCTP association with a remote endpoint, it sends an INIT chunk to the remote endpoint. When the remote endpoint receives the INIT chunk, it sends an INIT-ACK chunk back to the opening endpoint. The INIT-ACK chunk contains a cookie, called the State Cookie. In order to create the cookie value, the endpoint uses information associated with the SCTP association being created and uses that information together with a secret key to generate a Message Authentication Code (MAC), which is then used as cookie value. The exact set of information that is used to create the cookie value is not standardized.

When the opening SCTP endpoint receives the INIT-ACK chunk, it sends a COOKIE-ECHO chunk to the remote endpoint. The COOKIE-ECHO chunk contains the cookie value that was received in the INIT-ACK chunk.

When the remote SCTP endpoint receives the COOKIE-ECHO chunk, it verifies that it was the creator of the cookie. If the verification is successful, the endpoint sends a COOKIE-ACK chunk to the opening endpoint. When the opening endpoint receives the COOKIE-ACK chunk, the SCTP association has been successfully established.

Figure 15 shows the basic message flow for establishing an SCTP association between two SCTP endpoints.

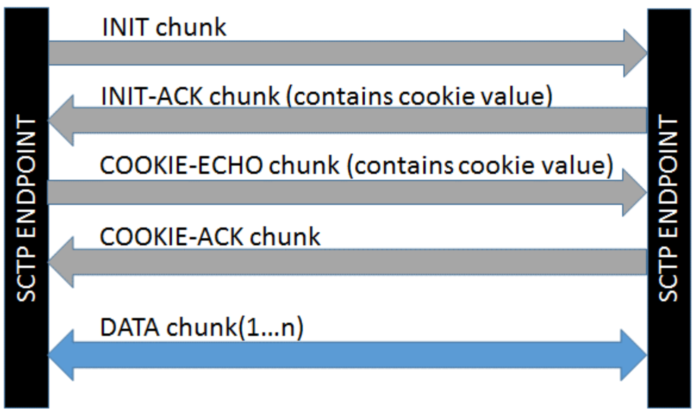

SCTP ENDPOINT
INIT chunk
INIT-ACK chunk (contains cookie value)
COOKIE-ECHO chunk (contains cookie value)
COOKIE-ACK chunk
DATA chunk(1...n)
SCTP ENDPOINT

Figure 15: SCTP Association Establishment

3.6.2 Send Data

In order for an SCTP endpoint to send user data to the remote endpoint, the SCTP endpoint uses a DATA chunk. The DATA chunk contains the user data as payload.

When an SCTP endpoint receives a DATA chunk, it sends back a SACK chunk in order to acknowledge that the DATA chunk was received. A single SACK chunk can be used to acknowledge multiple DATA chunks.

Figure 16 shows the structure of a DATA chunk. The user payload is carried in the User Data portion of the chunk.

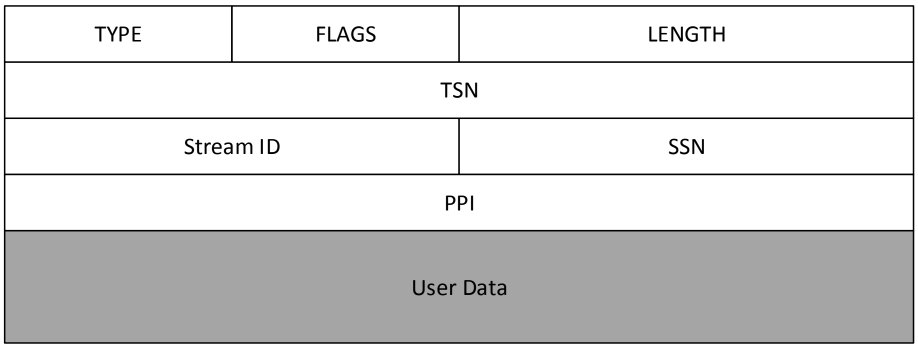


Figure 16: DATA chunk

3.6.3 Close Association

An SCTP association can be closed in a graceful and an immediate manner. In order to close an SCTP association in a graceful manner, an SCTP endpoint (closing endpoint) sends a SHUTDOWN chunk to the remote endpoint. After that, the closing endpoint will no longer accept DATA chunks received from the remote endpoint. The closing endpoint will, however, retransmit any DATA chunk that has yet not been acknowledged (using a SACK chunk) by the remote endpoint.

Once the remote SCTP endpoint has acknowledged all DATA chunks, it sends a SHUTDOWN-ACK chunk to the closing endpoint. When the closing endpoint receives the SHUDOWN-ACK chunk, it sends a SHUTDOWN-COMPLETE chunk to the remote endpoint. After that the SCTP association has been successfully closed.

In order to close an SCTP association in an immediate manner, an SCTP endpoint (closing endpoint) sends an ABORT chunk to the remote endpoint. The SCTP association is considered closed once the remote endpoint has received the ABORT chunk.

3.7 SCTP Extensions

3.7.1 General

In addition to the core SCTP capabilities, the protocol allows new capabilities to be defined as SCTP extensions. Section 3.7 describes the SCTP extensions which can be used when SCTP is used to realize a WebRTC data channel.

3.7.2 Padding

RFC4820 defines a new SCTP chunk type, PAD, and a new parameter, Padding, for the INIT chunk type [24]. The PAD chunk can be used to pad an STCP message to a specific size, while the Padding parameter can be used to pad an INIT chunk to a specific chunk size. The extension can be used by SCTP implementations to determine the MTU, using the mechanism defined in RFC 4821 [25]. Figure 17 shows the structure of the PAD chunk.

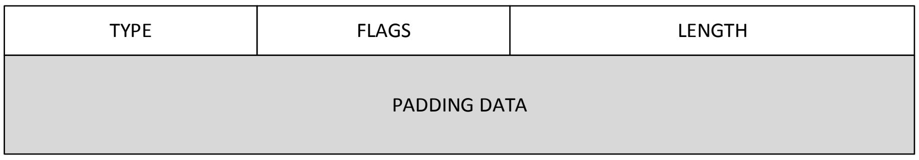


Figure 17: PAD chunk

3.7.3 Stream Reconfiguration

RFC 6525 defines a new chunk type, RE-CONFIG, which can be used to “reset” an SCTP stream by setting the Stream Sequence Number (SSN) value associated with the stream back to zero. This allows re-usage of a given SCTP stream for a different purpose and to reset the SSN value when the new usage is initiated. The mechanism can also be used to reset the SSN values of all SCTP streams associated with an SCTP association. [26.]

3.7.4 Dynamic Address Reconfiguration

SCTP supports multi-homing, which means that an endpoint can allocate multiple IP addresses to a single SCTP association. RFC 6525 defines a mechanism, which allows dynamic allocation of a new IP address to an SCTP association and de-allocation of previously allocated IP addresses.

In addition, RFC 6525 defines a new generic INIT/INIT-ACK chunk parameter, Supported Extensions, which can be used by SCTP endpoints to indicate support for SCTP extensions.

3.7.5 Partial Reliability

By default, SCTP messages are sent reliably between SCTP endpoints by retransmitting each SCTP message until the remote endpoint acknowledges that it has received the message. SCTP messages can also be sent unreliably, in which case they will not be retransmitted by the SCTP layer.

RFC 3758 defines a new chunk type, FORWARD-TSN. An SCTP endpoint can use the chunk to inform the remote SCTP endpoint not to expect certain chunks, as they will no longer be re-transmitted. [27.]

RFC 7496 extends RFC 3758 by defining two additional partial reliability policies: Limited Retransmission and Priority. The Limited Retransmission policy allows limiting the retransmission of the SCTP messages. Once the number of retransmissions of a message has exceeded the value defined by the policy, the sending endpoint must discard the message. It is possible to set the number of retransmissions to zero, which means that a message will not be retransmitted at all. The Priority policy allows discarding of lower priority messages, if a send buffer is running out of space and buffer space is needed for higher priority messages. [28.]

3.7.6 N-DATA

As described earlier, SCTP can fragment large user messages and transport the fragments using multiple DATA chunks. In case of very large user messages, this may result in a large number of DATA chunks. SCTP mandates that each DATA chunk transporting a fragment of a user message have consecutive TSN values, as the TSN value is used by the receiving endpoint to collect and reassemble the user message. Because of this, while the user message fragments are transported, the sending endpoint cannot assign TSN values to DATA chunks (and transport those chunks) associated with other user messages, even if they are associated with another SCTP stream. This can cause head of line blocking.

Figure 18 shows an example, where two user messages are to be sent. User message #1 is very large, why it is fragmented and transported using multiple DATA chunks. User message #2 cannot be transported until the last fragment of user message #1 has been sent.

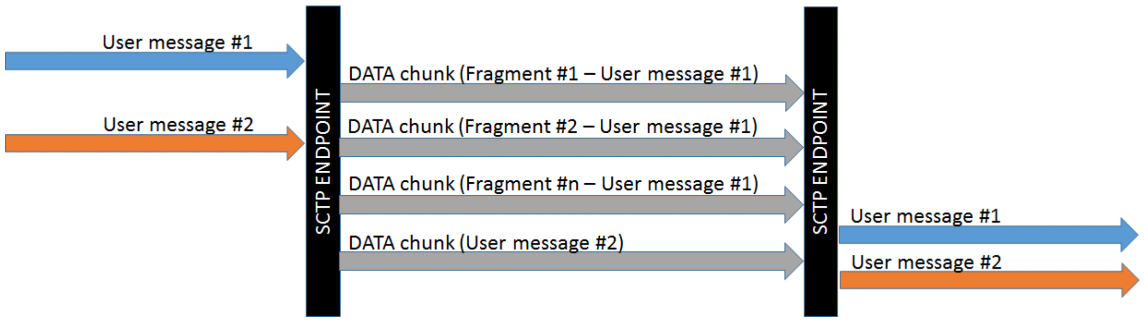

User message #1
User message #2
SCTP ENDPOINT
DATA chunk (Fragment #1 – User message #1)
DATA chunk (Fragment #2 – User message #1)
DATA chunk (Fragment #n – User message #1)
DATA chunk (User message #2)
SCTP ENDPOINT
User message #1
User message #2

Figure 18: Head of line blocking of user message [29]

The IETF standard draft document draft-ietf-tsvwg-sctp-ndata defines a new data chunk type, N-DATA. The N-DATA chunk is otherwise identical to the DATA chunk, but it contains two new parameters: Fragment Sequence Number (FSN) and Message Identifier (MID). The FSN is used to reassemble fragmented user messages, while the MID is used to identify a specific user message. As the TSN value is no more needed for reassemble purpose, fragmented user messages do not need to be transported using consequent TSN values, which means that, while user message fragments are still transported, chunks transporting other user messages can be sent simultaneously. This is referred to as interleaving. Figure 19 shows an example where N-DATA chunks are used and the transport of user message #2 between the SCTP endpoints is not affected by the sending of user message #1. [29.]

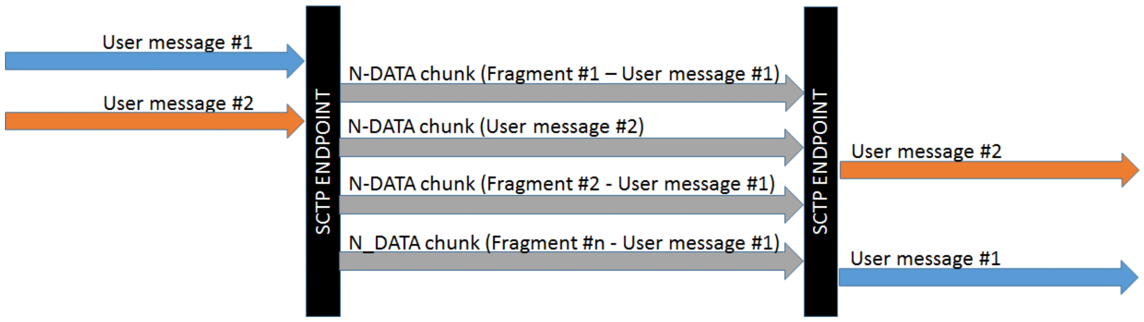

User message #1
User message #2
SCTP ENDPOINT
N-DATA chunk (Fragment #1 – User message #1)
N-DATA chunk (User message #2)
N-DATA chunk (Fragment #2 - User message #1)
N_DATA chunk (Fragment #n - User message #1)
SCTP ENDPOINT
User message #2
User message #1

Figure 19: Fragmentation using N-DATA chunks

The support of the N-DATA chunk is negotiated during the SCTP association establishment. If both SCTP endpoints indicate support of N-DATA chunks, they must be used to transport all user messages associated with the SCTP association, even if the user messages are not fragmented. Figure 20 shows the structure of the N-DATA chunk.

TYPE (64)	FLAGS	LENGTH	
TSN			
Stream ID		SSN	
Payload Protocol Identifier (PPI)			
Message Identifier			
Fragment Sequence Number (FSN)			
User Data			

Figure 20: N-DATA chunk

4 Datagram Transport Layer Security (DTLS)

4.1 General

DTLS is based on the Transport Layer Security (TLS) protocol. Compared to TLS, DTLS has been modified in order to be used with unreliable transport-layer protocols, such as UDP, and transport-layer protocols that contain features not supported by TLS, such as SCTP. [15;16;22.]

TLS assumes that the underlying transport-layer protocol provides reliable data transport, without packet loss or re-ordering of messages. A packet loss, or the receiving of a re-ordered message, will be considered an attack, and the TLS connection will be closed. DTLS makes no assumptions regarding transport reliability and message ordering. DTLS also contains a fragmentation mechanism for large messages.

Even though DTLS messages are sent unreliably, some procedures require each DTLS message to reach the remote DTLS endpoint in order to succeed. In order to ensure the delivery of each DTLS message, a message re-transmission mechanism is used.

DTLS uses public key encryption in order to generate shared secret keys, which will then be used for encrypting the user data.

4.2 Protocol Stack

The DTLS protocol stack contains of two layers: the record layer and the sub-protocol layer. Figure 21 shows the structure of the DTLS protocol stack, on top of the UDP transport protocol.

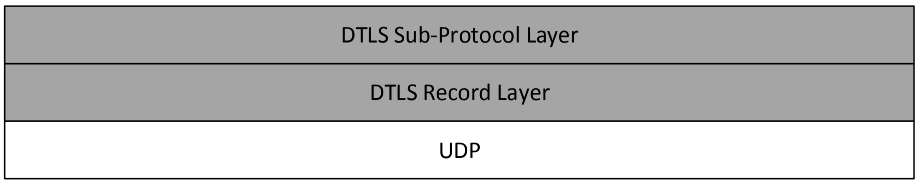


Figure 21: DTLS protocol layers

The record layer payload is carried on the sub-protocol layer. The payload can contain sub-protocol messages or DTLS user data. DTLS defines 3 sub-protocols: Handshake, ChangeCipherSpec and Alert.

4.2.1 Record Layer

Every DTLS message contains the record layer header fields. Figure 22 shows the structure of the record layer.

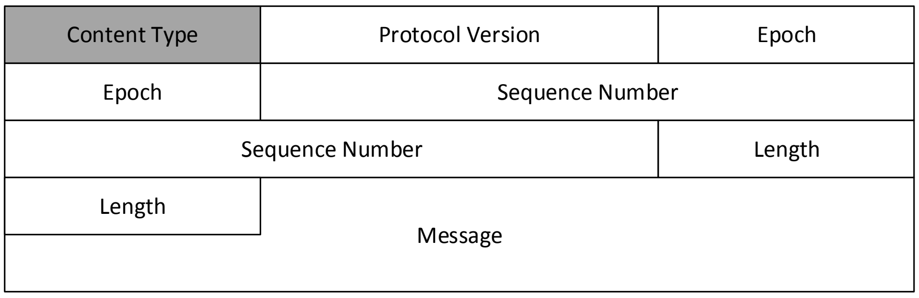


Figure 22: DTLS record layer

The Content Type value indicates which of the sub protocols is carried on the sub protocol layer. The Sequence Number is a unique value, which is increased with every sent message. The Epoch value is increased every time there is a change in the cipher suite state. When the epoch value is increased, the sequence number is set to zero.

4.2.2 Sub-Protocol Layer Protocols

The DTLS message payload can contain sub-protocol messages or DTLS user-data, as shown in figure 23.

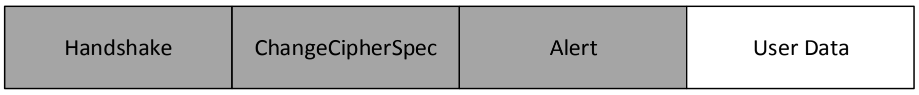


Figure 23: DTLS sub-protocol layer protocols

The Handshake protocol is used to negotiate cipher suites (collections of encryption algorithms) and keys. The Alert protocol is used to inform the remote DTLS endpoint about errors that have occurred. The ChangeCipherSpec protocol is often used together with the Handshake protocol, but is still defined as a separate protocol. It is used to indicate that any subsequent DTLS messages will be encrypted using the previously negotiated cipher suite and keys.

The Alert protocol is used by a DTLS endpoint to inform the remote endpoint about warnings or errors. An error will trigger an immediate closure of the DTLS connection.

The Alert protocol is used to inform the other peer about warnings or errors. An error will trigger an immediate closure of the DTLS connection. The Alert protocol can also be used to trigger a graceful closure of a DTLS connection.

4.3 Fragmentation

In order to prevent network fragmentation of the UDP datagrams used to carry DTLS messages, a DTLS protocol message can be fragmented into several protocol messages, each carried in a separate UDP datagram. Especially a Handshake protocol message can be very large, for example if they carry certificate information.

4.4 Procedures

4.4.1 Establish DTLS Connection

In order for a DTLS endpoint (DTLS client) to establish a DTLS connection, it uses the handshake protocol. The endpoint initiates the handshake by sending a ClientHello message to the remote endpoint (DTLS server). The ClientHello message contains the cipher suites, hash and compression algorithms supported by the client and a random number value.

When the DTLS server receives the ClientHello message, it sends a ServerHello message back to the DTLS client. The ServerHello message contains the cipher suites supported by the server and a random number value (different from the value sent by the client in the ClientHello message). The random numbers will be used to calculate the master key.

Once the server has sent the ServerHello message, it may send a ServerCertificate message, in order to provide its certifications used to authenticate itself. The server may also send a CertificateRequest message, in order to request the client to authenticate itself. Some cipher suites require additional data to be exchanged between the DTLS endpoints. Such data can be sent using the ServerKeyExchange message. Once the server has sent all data, it sends a ServerHelloDone message in order to indicate that it will not send any further messages.

As the DTLS ClientHello message does not require a connection, it can be used as an easy Denial of Service (DoS) attack mechanism. In order for the server to verify the client before further processing the ClientHello message, it can send a HelloVerifyRequest to the client. The message contains a cookie value. The client needs to send a new ClientHello message, which contains the cookie value. If the cookie value is signed and the server can verify it, the server knows that the client uses a valid address. Once the client has sent the new ClientHello message, the handshake procedure continues as normal. Figure 24 shows the DTLS connection establishment procedure.

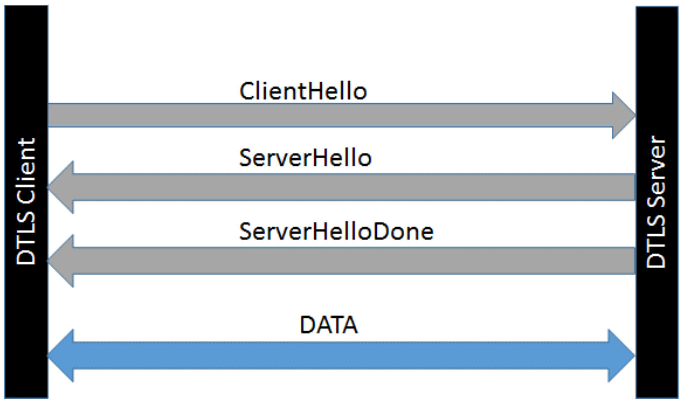


Figure 24: DTLS connection establishment

4.4.2 Close DTLS Connection

When a DTLS endpoint (closing endpoint) wants to close a connection, it sends a CloseNotify message. When the remote DTLS endpoint receives the message, it sends a CloseNotify message back. When the closing endpoint receives the CloseNotify message, the DTLS connection is considered closed.

5 Encapsulating SCTP Over DTLS

5.1 General

The IETF standard draft document draft-ietf-tsvwg-sctp-dtls-encaps defines a mechanism for encapsulating SCTP over DTLS. From a DTLS perspective, each SCTP message is processed as user data. When an SCTP endpoint is going to send an SCTP message, the complete SCTP message, including the common header and all chunks, is passed to the DTLS layer. The underlying transport-layer protocol is UDP. Figure 25 shows the protocol layers. [30.]

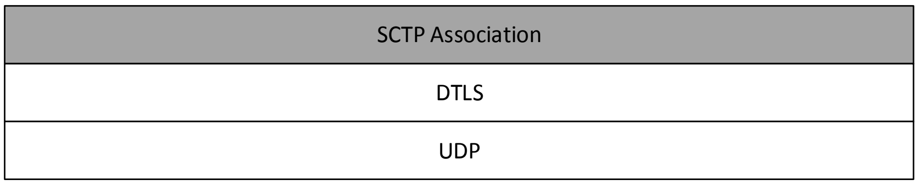


Figure 25: SCTP encapsulation over DTLS

The encapsulating mechanism provides confidentiality, source authenticated and integrity protected data transfers and the SCTP association is encrypted using DTLS. The usage of UDP on the transport-layer makes NAT traversal easier.

Multiple SCTP associations can be established over a single DTLS connection. In such cases, the SCTP port value will be used to distinguish the SCTP associations.

5.2 SCTP Restrictions

When SCTP is encapsulated over DTLS, some restrictions apply due to the fact that SCTP is not used as a transport-layer protocol (SCTP is carried on top of DTLS, which is carried on top of UDP).

As the SCTP layer does not have any information regarding which IP version is used on the transport protocol layer (below the DTLS layer), the SCTP INIT and INIT-ACK chunks must not contain IPv4 or IPv6 address parameters.

The SCTP application must not rely on the usage of ICMP packets. Received ICMP packets may not be provided by the transport protocol layer to the SCTP layer and the SCTP layer may not be able to trigger the sending of ICMP packets.

The DTLS connection must be established before the SCTP association can be established.

5.3 SCTP Extensions

Implementations of the SCTP encapsulating mechanism must support the SCTP padding extension in order to perform MTU discovery. Other SCTP extensions can be used, but no other extensions are required to be supported by implementations.

6 WebRTC Data Channel Realization

6.1 General

A WebRTC data channel is realized using two unidirectional SCTP streams, each using the same stream identifier value. The SCTP association that provides the SCTP streams is encapsulated over DTLS. Figure 26 shows the protocol structure of a WebRTC data channel realization. [3.]

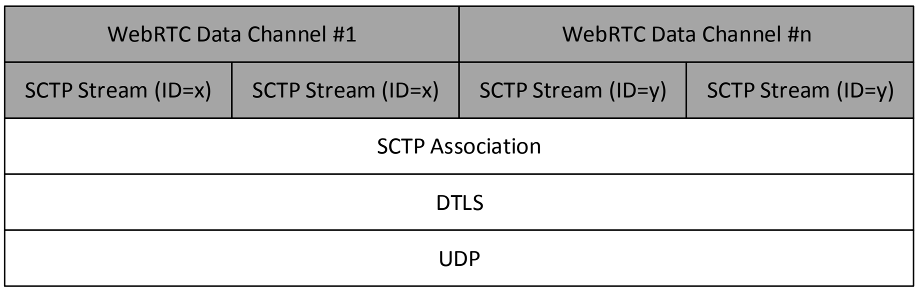


Figure 26: WebRTC data channel protocol stack

The data channel does not define a layer on top of the SCTP layer. Instead, two SCTP streams form the data channel and the data to be transported on the data channel is passed to the SCTP layer as SCTP user data.

Using a set of SCTP extensions, different properties can be provided to a data channel. Data delivery can be ordered or unordered, with different degrees of reliability.

The data channel mechanism does not limit the number of data channels that can be simultaneously opened (the maximum number of streams defined by SCTP is 65535). Any limit is dependent on a particular data channel endpoint implementation.

6.2 SCTP Considerations

6.2.1 Restrictions

The restrictions described earlier also apply when SCTP is used to realize a data channel.

6.2.2 PPID Values

When used to realize a WebRTC data channel, the SCTP PPID value is used to distinguish between UTF-8 encoded user data, binary user data and the Data Channel Establishment Protocol (DCEP). There are also PPID values in case the user data payload is empty.

If a data channel endpoint receives an SCTP message with a non-supported PPID value, the data channel should be closed. Table 2 shows the valid PPID values.

Table 2: WebRTC data channel PPID values

SCTP PPID Value	Description
50	DCEP
51	UTF-8 string (WebRTC string)
53	Binary string (WebRTC binary)
56	String empty
57	Binary empty

6.2.3 Extensions

A WebRTC data channel endpoint must support most SCTP extensions listed in section 3.7. The support and usage of the N-DATA chunk type is optional. In some cases only certain pieces of an extension are used. The SCTP stream reconfiguration mechanism is only used to close a data channel. The dynamic address reconfiguration mechanism is not used, but data channel endpoints use the Supported Extensions parameter defined as part of the mechanism as a generic tool for indicating support for SCTP extensions.

6.3 Procedures

6.3.1 Open Data Channel

The WebRTC data channel mechanism does not define signalling procedures for negotiation and opening a WebRTC data channel. A data channel can be negotiated and opened using an in-band mechanism (sending messages on the SCTP stream which will be used to realize the data channel), or using an out-of-band mechanism. The IETF standard draft document draft-ietf-rtcweb-data-protocol defines an in-band mechanism, Data Channel Establishment Protocol (DCEP) [31].

6.3.2 Send Data

In order to send user data on a data channel, the data is provided to the SCTP layer as SCTP user data. The SCTP message is sent on one of the SCTP streams associated with the data channel, using the properties (for example related to reliability) selected for that data channel.

6.3.3 Close Data Channel

In order to close a WebRTC data channel, the SCTP stream reconfiguration mechanism is used. The closing endpoint sends an SCTP RE-CONFIG chunk, where the Stream Sequence Number (SSN) value is set to zero (“reset”), on the SCTP stream association with the data channel. If the complete SCTP association used to realize the data channel is closed, each open data channel is automatically closed.

6.4 Interactive Connectivity Establishment (ICE)

6.4.1 General

When a browser (client) connects to a Web server, NAT traversal is normally not an issue, as the Web server is typically not located behind a NAT. However, when a client connects to another client, one or both clients are likely to be located behind NATs, which prevents the establishment of a connection (for example a WebRTC data channel) between the clients. For that reason, a mechanism to pass the NATs is needed. [13.]

Section 6.4 describes the Interactive Connectivity Establishment (ICE) NAT traversal mechanism, which is used by data channel endpoints in order to create data channel connections when one or both endpoints are located behind NATs. [32.]

ICE is a collection of different mechanisms for traversing NATs and defines procedures for selecting the best mechanism for any given scenario.

When a connection is to be established, the ICE endpoint first collects candidates. A candidate is an IP address and port. Each endpoint then sends (using some signalling mechanism) its candidates to the remote endpoint. The endpoints will then, from each of its own candidate, try to connect to each of the remote endpoint's candidate. If the connection is successful, the pair of candidates can be used to establish a connection between the endpoints. In case multiple candidate pairs are successful, ICE defines procedures on how to prioritize and select the best candidate. Sometimes a new set of candidates needs to be collected and tested during a session. That is referred to as an ICE re-start.

6.4.2 Candidate Types

A host candidate contains the local IP address and the port of the ICE client. That represents the address that the host would use if it did not use ICE.

A server reflexive candidate contains an IP address and port of the ICE client, seen by a STUN server. If the ICE client is located behind a NAT, the server reflexive candidate contains the public address that has been assigned to the ICE client by the NAT. The ICE client obtains the server reflexive candidate by sending a request, using the Session Traversal Utilities for NAT (STUN) protocol, to the STUN server. The STUN server detects the source address (for example the public address of a NAT) of the request and copies that into the response sent back to the ICE client. In case there are multiple NATs between the ICE client and the STUN server, the server reflexive candidate represents the public address of the NAT located nearest the STUN server. In case there is no NAT between the ICE client and the STUN server, the server reflexive candidate address will be identical to the host candidate. It is important to note that the connection between the ICE clients will not traverse the STUN server. The server is only used for handling the STUN request that is used to provide the server reflexive candidate to the ICE client.

A relayed candidate contains an IP address and port provided by a dedicated relay entity, referred to as a Traversal Using Relays around NAT (TURN) relay. If a relayed candidate is used to establish a connection between the ICE clients, the connection will also traverse the TURN relay. The STUN protocol is used between the ICE client and the TURN relay, in order to authenticate the client with the TURN relay and to obtain the relay candidate address from the relay. Figure 27 shows the ICE architecture and from where the different types of candidates are retrieved.

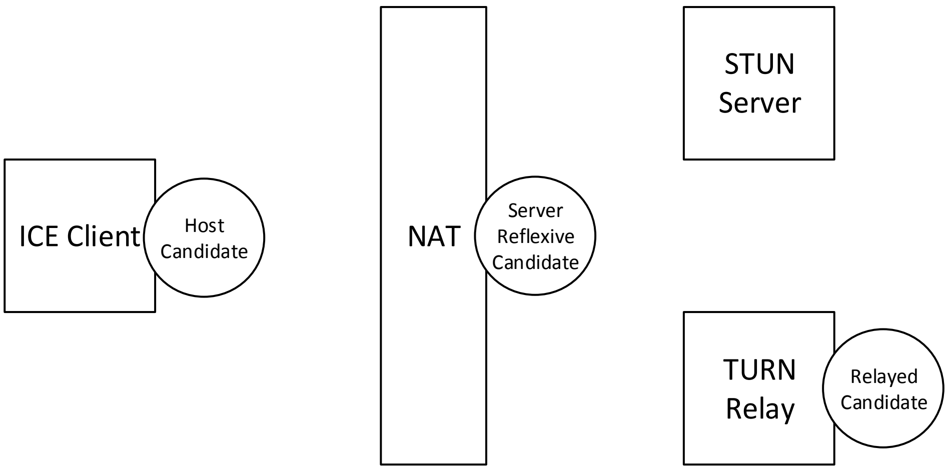


Figure 27: ICE architecture

ICE does not define any mechanisms for ICE clients to find STUN servers and TURN relays. Currently, the servers often have to be manually configured on the client. The WebRTC mechanism allows a webserver to provide a list of servers to the client within the JavaScript code.

6.4.3 Connectivity Checks

Once a candidate pair has been selected, STUN is between the ICE clients to perform periodic keep-alives throughout the duration of the connection. It is also used to verify that the remote client is still willing to receive data.

6.5 Data Channel Establishment Protocol (DCEP)

6.5.1 General

DCEP is an in-band protocol which can be used to open a WebRTC data channel. Each DCEP message is used on the SCTP stream which will be used to realize the data channel. [31.]

Currently, two message types have been defined for DCEP: DATA_CHANNEL_OPEN and DATA_CHANNEL_ACK.

DCEP cannot be used to close a data channel. Instead, the SCTP reconfiguration mechanism is used to close a data channel (see section 3.7.3).

6.5.2 SCTP Considerations

A DCEP message must be sent using ordered and reliable delivery. Different properties may be used for subsequent data transported on the WebRTC data channel. The SCTP PPID value for a DCEP message is 50.

6.5.3 DATA_CHANNEL_OPEN

The DATA_CHANNEL_OPEN message is used to open a WebRTC data channel. It contains information regarding the SCTP characteristics associated with the data channel and information about the application protocol for which the data channel is going to be used.

The data channel endpoint that sends the DATA_CHANNEL_OPEN selects an available SCTP stream value and sends the DATA_CHANNEL_OPEN message on that stream. Figure 28 shows the structure of the DCEP DATA_CHANNEL_OPEN message.

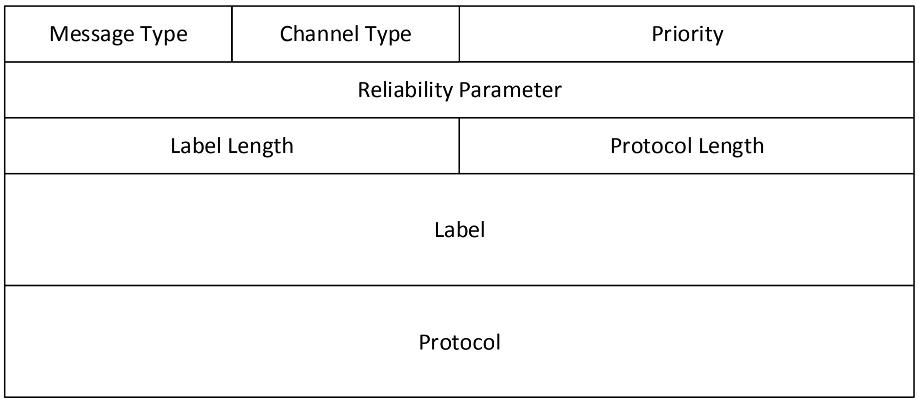


Figure 28: DCEP DATA_CHANNEL_OPEN

The Message Type parameter specifies the message type value assigned for the DATA_CHANNEL_OPEN message. The Channel Type parameter specifies the data channel type. The following types have been defined:

- DATA_CHANNEL_RELIABLE
- DATA_CHANNEL_RELIABLE_UNORDERED
- DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT
- DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED
- DATA_CHANNEL_PARTIAL_RELIABLE_TIMED

- DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED

The Priority field specifies the priority of the data channel.

The Reliability Parameter field semantics depends on the value of the Channel Type field. It either indicates the maximum number of times that a message will be retransmitted or it indicates the maximum timer value in which case the message will be transmitted until the timer value expires.

The Label field is optional and can contain a user defined name of the data channel

The Protocol field is optional and can contain a registered value of a protocol for which the data channel is used.

6.5.4 DATA_CHANNEL_ACK

The DATA_CHANNEL_ACK message is sent by the receiver of the DATA_CHANNEL_OPEN message, to acknowledge that the DATA_CHANNEL_OPEN message was received and accepted.

When a WebRTC data channel endpoint receives a DATA_CHANNEL_OPEN message, it sends back a DATA_CHANNEL_ACK message, if it accepts the opening of the data channel.

6.5.5 Security

DCEP does not provide any privacy, integrity or authentication. When used for a WebRTC data channel, security is provided by the SCTP encapsulating mechanism used to realize the data channel.

7 WebRTC Data Channel API

7.1 General

WebRTC 1.0: Real-time Communications between Browsers defines a JavaScript API, which can be used by JavaScript applications to create and use a WebRTC data channel. Section 7 describes the steps for creating and using a data channel using the API. [2.]

7.1.1 Roles

When a data channel is going to be established between two endpoints, each endpoint is assigned one of the following roles: offerer and answerer. The endpoint that initiates the negotiation in order to establish a data channel becomes the offerer. The address information that the offerer will send to the answerer is referred to as an offer, while the address information that the answerer will return is referred to as the answer.

Applications need to handle the case where both endpoints try to initiate the negotiation in order to establish a data channel, as only one endpoint can be the offerer for any give session.

7.1.2 Signalling

In order for the endpoints to exchange address and ICE candidate information, there needs to be a signalling mechanism for transporting the information (unless the information is configured in the endpoints).

The WebRTC specification does not define a signalling mechanism for exchanging the information. Between the browser and the JavaScript application the address information is exchanged using the Session Description Protocol (SDP) format, but between the endpoints the information can be exchanged using any format and protocol (as long as the endpoints understand it). [33.]

Before the address information has been exchanged, an endpoint normally does not know the location of the other endpoint. Therefore, endpoints normally inform a Web server about their presence. Then, when an endpoint sends the address and ICE information to the Web server, it forwards the information to the other endpoint.

Figure 29 shows the architecture, where the signalling between the WebRTC data channel endpoints traverse a web server, while the WebRTC data channel is established directly between the WebRTC endpoints.

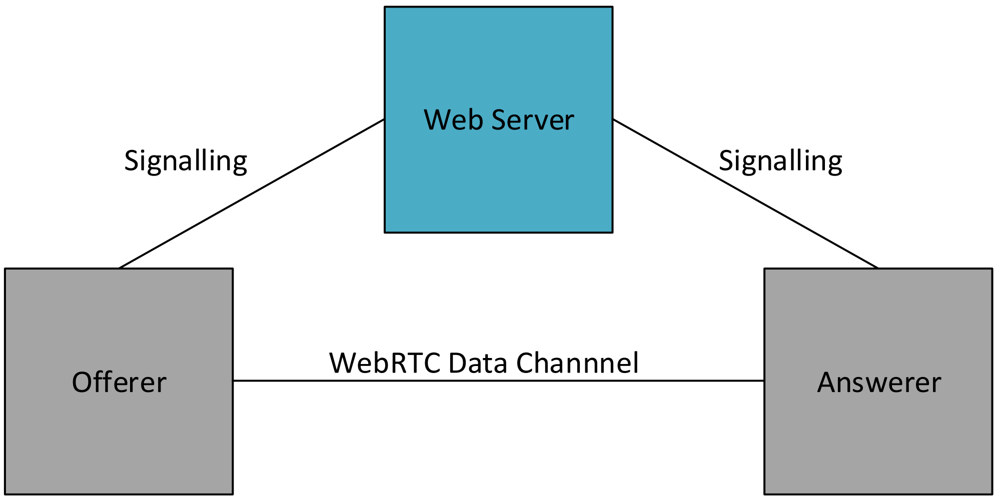


Figure 29: WebRTC high-level architecture

7.2 Procedures

7.2.1 ICE Configuration

In order for an endpoint to collect ICE candidates, it needs to have information about available STUN servers and TURN relays. Currently, a mechanism to discover those does not exist, so the JavaScript application must provide the information to the browser. The servers and relays are configured in an array. [32;34;35.]

The code below shows an example of how information about STUN servers and TURN relays are provided using JavaScript. The information contains the domain name and the port associated with the STUN server or TURN relay. In the case of a TURN relay, the information also contains the credentials needed to use the TURN relay.

```
var myICEServers = {
            'iceServers': [
            {'urls':'stun:stun.l.google.com:19302'},
            {'urls':'stun:stun1.l.google.com:19302'},
            {'urls':'stun:stun2.l.google.com:19302'},
            {'urls':'stun:stun3.l.google.com:19302'},
            {'urls':'stun:stun4.l.google.com:19302'},
            {'urls':'stun:stun01.sipphone.com'},
            {'urls':'stun:stun.ekiga.net'},
            {'urls':'stun:stun.fwdnet.net'},
            {'urls':'stun:stun.ideasip.com'},
            {'urls':'stun:stun.iptel.org'},
            {'urls':'stun:stun.rixtelecom.se'},
            {'urls':'stun:stun.schlund.de'},
            {'urls':'stun:stunserver.org'},
            {'urls':'stun:stun.softjoys.com'},
            {'urls':'stun:stun.voiparound.com'},
            {'urls':'stun:stun.voipbuster.com'},
            {'urls':'stun:stun.voipstunt.com'},
            {'urls':'stun:stun.voxgratia.org'},
            {'urls':'stun:stun.xten.com'},{'urls':'turn:numb.viagenie.ca:3478','creden-
            tial':'<password>','username':'<username>'}
            ]
        }
```

7.2.2 Create Offer (Offerer)

The RTCPeerConnection class is the main interface provided by the WebRTC API. First, the offerer creates an RTCPeerConnection class instance. The array containing the STUN server and TURN relay information is provided as an input parameter to the class constructor.

The code below shows how an RTCPeerConnection class instance is created in JavaS-cript. The list of STUN servers and TURN relays is provided as an input parameter when the class instance is created.

myPC = new ***RTCPeerConnection****(myICEServers);*

Then, the offerer calls the RTCPeerConnection.createDataChannel() function in order to retrieve a DataChannel class instance. The offerer implements a set of event callback functions for the DataChannel class instance: onopen (when the data channel with the offerer has been opened), onmessage (when a message has been received on the data channel), onclose (when the data channel has been closed) and onerror (when an error has occurred on the data channel).

The code below shows how a data channel class instance is created and how the references to the event handling functions are assigned to the different event types associated with the class instance.

```
myDC = myPC.createDataChannel();
myDC.onmessage = eventDCMessage;
myDC.onopen = handleDataChannelOpen;
myDC.onclose = handleDataChannelClosed;
```

Then, the offerer needs to send the address information (offer) associated with the data channel to the answerer. The offer contains the IP address and port that will be used by the answerer to establish the SCTP association used to realize the data channel towards the offerer. In order to retrieve the offer, the offerer calls the RTCPeerConnection.createOffer() function. Once the offer has been created by the browser, an event callback function is called. The callback function calls the RTCPeerConnection.setLocalDescription() function in order to explicitly apply the offer to the browser.

The code below shows how the browser is requested to create an offer. The callback function, which the browser calls once it has created the offer, is provided as an input parameter.

```
myPC.createOffer(function(offer) {
                myPC.setLocalDescription(offer);
});
```

After that, the offerer sends the offer (either in the offer SDP format or mapped into some other format) to the answerer.

7.2.3 Send and Receive ICE Candidates (Offerer)

The offerer retrieved when calling the RTCPeerConnection.createOffer() does not contain the ICE candidates collected by the browser. Instead, the offerer needs to implement the RTCPeerConnection.onicecandidate() event callback function that will be called whenever a new local ICE candidate has been collected by the browser. The ICE candidates must also be sent to the answerer.

The code below shows how a callback function that the browser will call whenever it has collected a new ICE candidate, is provided to the browser. The callback function will typically send the candidate to the remote endpoint using some signalling mechanism.

```
myPC.onicecandidate = function (event) {
            var candidate = event.candidate;
            // Process candidate
}
```

When the offerer receives a candidate from the answerer, it calls the RTCPeerConnection.addIceCandidate() function in order to apply the candidate to the browser.

The code below shows how an ICE candidate that has been received from the remote endpoint is provided to the browser.

```
myPC.addIceCandidate(new RTCIceCandidate(candidate));
```

7.2.4 Receive Offer (Answerer)

When the answerer receives the offer, it also creates an RTCPeerConnection class instance.

The code shows how the answerer creates an RTCPeerConnection class instance. The code is otherwise identical to the code shown in section 7.2.2, but each endpoint may have a different list of STUN servers and TURN relays.

```
myPC = new RTCPeerConnection(myICEServers);
```

Unlike the offerer, the answerer does not call the RTCPeerConnection.createDataChannel() function. Instead, it implements a callback function for the RTCPeerConnection.ondatachannel event that will be called once the answerer endpoint has created a data channel handler. The callback function will receive a data channel class instance and must implement the same set of data channel callback functions as the offerer.

The code below shows how the reference to the generic data channel event is set to the RTCPeerConnection class instance. Within that function, the references to event handling functions for specific data channel events are set.

```
myPC.ondatachannel = eventDC;

function eventDC(e) {
        myDC = e.channel;
        myDC.onmessage = eventDCMessage;
        myDC.onopen = eventDCOpen;
        myDC.onclose = eventDCClosed;
}
```

Then, the answerer calls the RTCPeerConnection.setRemoteDescription() function and gives the offer as input parameter.

The code below shows how the offer, received from the remote endpoint, is set for the RTCPeerConnection class instance.

```
myPC.setRemoteDescription(new RTCSessionDescription(offer));
```

7.2.5 Create Answer (Answerer)

Similar to the offerer, the answerer also needs to send its address information (answer) back to the offerer. It calls the RTCPeerConnection.createAnswer() function and provides a callback function. Once the answer has been created by the browser, the callback function is called. The callback function calls the RTCPeerConnection.setLocalDescription() function in order to explicitly apply the answer to the browser.

The code below shows how the answerer requests the browser to create an answer. It provides a callback function as an input parameter that the browser calls once it has created the answer.

```
myPC.createAnswer(function(answer) {
        myPC.setLocalDescription(answer);
});
```

When the answerer has created the answer, it sends the answer back to the offerer (using the same format in which it received the offer).

7.2.6 Send and Receive ICE Candidates (Answerer)

The answer also needs to collect ICE candidates, send them to the offerer and handle candidates received from the offerer. The answerer procedures are identical to the offerer procedures.

7.2.7 Receive Answer (Offerer)

When the offerer receives the answer, the offerer applies it by calling the RTCPeerConnection.setRemoteDescription() function.

The code shows how the answer received from the remote endpoint is set to the RTCPeerConnection class instance.

```
myPC.setRemoteDescription(new RTCSessionDescription(answer));
```

After this, once the ICE procedures have finished, the SCTP association used to realize the data channel has been established between the endpoints and the data channel has been opened, the endpoints can start sending data on the data channel. When an endpoint receives data on the data channel, the onmessage event callback function will be called.

```
function eventDCMessage(event) {
            // The data channel message is stored in event.data
}
```

7.2.8 Send Data

In order to send data on the data channel, the endpoints calls the DataChannel.send() function, providing the content to be sent as input parameter.

The code shows the send function used to send data on a data channel, using the send() function provided by the data channel class instance. The data content is provided as an input parameter.

*myDC.**send**(content);*

7.2.9 Close Data Channel

In order to send data on the data channel, an endpoint call the DataChannel.close() function.

The code shows how a data channel is closed, using the close() function provided by the data channel class instance.

*myDC.**close**();*

8 Example Applications

8.1 General

Section 8 describes two demo applications that were implemented as part of the final year project. It contains a WebSocket server, which is used by the browsers to exchange data needed in order to establish a WebRTC data channel and two browser applications (a game and a file share application) using a data channel to exchange data.

Most source code was written for the purpose of this final year project. However, the source code for the **saveToDisk()** function and part of the source code for the **onRead-File()** function, both used in the file share application, were copied from https://www.webrtc-experiment.com/docs/how-file-broadcast-works.html.

8.2 Simple WebSocket Server

8.2.1 Description

The Simple WebSocket server is a WebSocket server implemented in node.js [36]. It stores information about WebSocket connections and when it receives a message on a WebSocket connection it broadcasts the message (unmodified) on each WebSocket connection (including the connection on which the message was received). Listing 1 in section 8.2.3 contains the JavaScript source code for the WebSocket server.

The Pong- and the File Transfer applications use the WebSocket server in order to exchange SDP- and ICE candidate information with each other, in order to be able to establish a data channel between the browser endpoints.

8.2.2 Node.js

Node.js is an open-source framework to build server-side applications. The programming language for node.js is JavaScript, which historically has only been used for client-side applications. Node.js is designed to be platform independent and the runtime is available for all major operating systems. [36.]

8.2.3 Source Code

This section contains the source code for the simple WebSocket server. When the source code is executed, a server starts to listen for incoming HTTP requests. The wsServer object contains a reference to a WebSocket handler associated with the server. Once an WebSocket connection is requested to be established, the **we-Server.on()** handler function is called. For each incoming WebSocket connection, a connection object is created and stored in a list of all connections. The **connection.on()** event handler function whenever data is received on that given connection, or when a connection is closed. When data is received on a given connection, the event handler function forwards it on each of the other connections.

```
var count = 0;
var clients = {};

//Create server
var http = require('http');
var server = http.createServer(function(request, response) {});

//Listen on WebSocket port
server.listen(1234, function() {
    console.log((new Date()) + ' Server is listening on port 1234');
});

//Create WebSocket server
var WebSocketServer = require('websocket').server;
wsServer = new WebSocketServer({
    httpServer: server
});

wsServer.on('request', function(r){

        //Accept incoming connection
        var connection = r.accept('echo-protocol', r.origin);

        //Store client connection reference
        var id = count++;
        clients[id] = connection

        // Message event listener
        connection.on('message', function(message) {

                      // Received message string
                      var msgString = message.utf8Data;

                      // Broadcast message string to each
                      // connected client
                      for(var i in clients){
                                    clients[i].sendUTF(msgString);
                      }
        });
        //Client listener
        connection.on('close', function(reasonCode, description) {
                delete clients[id];
        });
});
```

Listing 1: Simple WebSocket Server JavaScript source code

8.3 Pong

8.3.1 Description

Pong is an application, written in JavaScript, implementing a simple version of the classic Pong game. Each player is controlling a shooter, which can be moved up and down. The purpose is to hit a ball which is moving between the players. If a player misses the ball and the ball hits the wall behind the player's shooter, the other player gets a point.

In the game, one browser is acting as a master. The master calculates the position of the ball and performs collision detection. The position of the master's shooter and the position of the ball, is sent over a data channel to the remote browser. The remote browser sends back the position of its shooter.

The Pong application uses reliable and ordered SCTP delivery and for those reasons the application to verify whether a message has been transmitted to the remote browser.

The application graphics is produced using HTML5 canvas. The canvas is painted 25 times per second. [37]

8.3.2 Usage

Below the functional steps that are taken by the Pong application are described.

1. Connect to WebSocket server: Once the HTML file (webrtc_web-socket_pong.html) has been downloaded, the browser automatically creates a Web-Socket connection with the configured WebSocket server.

2. Connect to remote browser: By pressing the “CONNECT” button, a browser sends SDP and ICE candidate information on the WebSocket connection. The WebSocket server will forward the information to the remote browser (if it has connected to the WebSocket server). The remote browser will send back its SDP and ICE candidate information. Once the information has been exchanged, a data connection will automatically be established between the browsers.

3. Start the game: The user that created the data channel connection between the browsers (see step 2) can start the game by pressing the “START” button.

4. Game control: Each player moves the shooter using the “UP” and “DOWN” buttons. The colour of the shooter that a player controls is purple, while the colour of the remote player’s shooter is white.

5. Exit: By pressing the “EXIT” button a user can terminate the application.

8.3.3 Source Code

The application contains two files: webrtc_websocket_pong.html and webrtc_websocket_pong.js. The source code for the application can be found in Appendix 1.

The webrtc_websocket_pong.html file is downloaded from a web server and contains the layout of the application. It then calls the webrtc_websocket_pong.js file, which contains the game logic and the functions for communicating over a WebSocket connection and a WebRTC data channel.

A description of the main functions defined in the webrtc_websocket_pong.js source code file is given below:

The **init()** function is called once the document has been loaded. It sets the click event handlers for the browser buttons and the buttons used to control the shooters.

The **sendOffer()**, **handleOffer()**, **handleAnswer()** and **handleCandidiate()** functions are called when the browsers exchange the information via the WebSocket server, needed in order to establish a WebRTC data channel. The functions take care of creating a PeerConnection within each browser, requesting the browser to reserve resources for a data channel and sending the information needed by the remote browser using the

WebSocket connection. The functions also request the browser to initiate ICE procedures and collect ICE candidates.

The **createDC()** function sets the event handlers associated with the data channel.

The **position()** function is the main game loop function, which is called 25 times per second. The function calculates the position of the game elements (the shooters and the ball), based on local user input and shooter position data received over the data channel from the remote browser. The function also calls the **collision()** and the **paint()** functions.

The **startGame()** function initiates the game loop when a user presses the “START” button.

The **exitGame()** function stops the game loop and terminates the game, when a user presses the “EXIT” button.

The **collision()** function performs collision detection between the shooters and the ball. If a collision is detected, the function modifies the variables describing the direction of the ball accordingly.

The **paint()** function performs all graphic tasks. It re-draws the canvas, using the positions of the shooters and the ball, as well as the current user scores, as input.

8.4 File Share

8.4.1 Description

File share is an application, written in JavaScript, which uses a data channel to transfer files between two browsers.

As a file can be large, it cannot be sent as one big single SCTP application message. For that reason, it needs to be split into smaller pieces (data chunks). Each data chunk is sent over the data channel as an SCTP application message.

Currently, as browsers do not support any overload mechanism for the data channel, the application needs to make sure it does not overload the data channel (which can lead to a data channel failure). For that reason, the File transfer application waits for 500 milliseconds between sending each data chunk.

The File share application uses reliable- and ordered SCTP delivery and for that reason the sender application does not need to verify that a given data chunk has been delivered and the receiving application does not need to keep order of the received data chunks.

8.4.2 Usage

Below the functional steps that are taken by the Fileshare application are described.

5. Connect to WebSocket server: Once the HTML file (webrtc_websocket_fileshare.html) has been downloaded, the browser automatically creates a WebSocket connection with the configured WebSocket server.

2. Connect to remote browser: By pressing the “CONNECT” button, a browser sends SDP and ICE candidate information on the WebSocket connection. The WebSocket server will forward the information to the remote browser (if it has connected to the WebSocket server). The remote browser will send back its SDP and ICE candidate information. Once the information has been exchanged, a data connection will automatically be established between the browsers.

3. Select file to send: Both browsers can select a file to be sent to the remote browser. Each browser can only send one file at any given time, but it is possible to send and receive a file at the same time.

4. Send file: Once the file has been selected, the transmission of the file towards the remote browser starts by pressing the “SEND” button. Once a file has been transmitted, a new file can be selected and sent.

5. Exit: By pressing the “EXIT” button a user can terminate the application. Any ongoing file transmissions will also be terminated and both the WebSocket connection and the data connection will be terminated.

8.4.3 Source Code

The application contains of two files: webrtc_websocket_fileshare.html and webrtc_websocket_fileshare.js. The source code for the application can be found in Appendix 2.

The webrtc_websocket_fileshare.html file is downloaded from a web server and contains the layout of the application. It then calls the webrtc_websocket_fileshare.js file, which contains the application logic and the functions for communicating over a WebSocket connection and a WebRTC data channel.

A description of the main functions defined in the webrtc_websocket_pong.js source code file is given below:

The **init()** function is called once the document has been loaded. It sets the click event handlers for the buttons and initializes the function used to select a local file to be transported to the remote browser.

The **sendOffer()**, **handleOffer()**, **handleAnswer()**, **handleCandidiate()** and **createDC()** functions are identical as for the Pong application (see Appendix 1).

The **sendFile()** function requests the browser to read the file to be transported to the remote user. Once the file has been read into a buffer, the onReadFile() callback function is called by the browser.

The **onReadFille()** function is called when a file to be transported has been read and copied into a buffer. The function splits the buffer content into smaller pieces (“chunks”) and sends each chunk to the remote browser using the data channel.

The **saveToDisk()** function is called when a browser has received a file. The function stores the file onto the harddisk. Whenever a browser receives a chunk, it stores it in a buffer. Once the last chunk associated with the file has been received, the saveToDisk() function is called.

The **leaveApp()** function terminates the application. Any possible ongoing file transfer till be terminated.

9 Conclusion

As mentioned in the introduction, due to the introduction of HTML5, there has been a big growth of browser based applications. The support of real-time media and data transmission, without the need for installing a 3rd party plug-in application, provided by the WebRTC mechanism, is expected to fuel that growth even further.

Until now, much focus has been on using WebRTC to provide audio and video features to browser applications, and a number of WebRTC based audio/video chat applications can be found. However, the data channel feature opens the doors for innovation and completely new types of applications. While the industry so far has focused pretty much on applications between two browsers, the mechanism can also be used to implement multi-participant applications, as the standard does not limit the number of PeerConnections that can be created within a browser.

The Pong and Fileshare demo applications, using the WebRTC data channel, that were implemented as part of the final year project and described in this thesis show that the data channel mechanism can be used both for transporting big amounts of data (the Fileshare application) and that the mechanism is fast enough also for exchanging time-critical information (the Pong application). Since browsers currently do not support any overload mechanism, the application designer needs to take some care when sending frequent portions of big data on a data channel.

The applications have been tested with the participating browsers being located in different networks, behind different types of NATs, which has shown that the NAT traversal mechanisms used by the data channel mechanism work well.

At the moment, all browsers do not yet support the WebRTC mechanism. However, due to the interest that has been shown in the industry and due to the fact that all major browser vendors have actively been participating in the WebRTC standardization work, the mechanism is expected to be introduced to all browsers within the near future. Apart from desktop browsers, some mobile device browsers also already support the mechanism.

Currently, there is a big industry interest in WebRTC with conferences and focus groups focusing on the mechanism. There is also ongoing standardization work on how to access next-generation, IP-based, telephone networks using the WebRTC mechanism.

There are also ongoing discussions about the standardization of the next version of WebRTC and what features and possible changes it will contain.

The author of this thesis strongly believes that WebRTC, and especially the data channel mechanism, will enable a big number of new applications, and that the mechanism will also be used within non-browser clients.

References

1 WebSocket, W3C Candidate Recommendation [online]. 20 September 2012.
URL: http://www.w3.org/TR/websocket.
Accessed 16 April 2015.

2 WebRTC 1.0: Real-time Communication between Browsers, W3C Working Draft [online]. 10 February 2015.
URL: http://www.w3.org/TR/webrtc.
Accessed 16 April 2015.

3 Jessup R, Loreto S, Tuexen M. WebRTC Data Channels. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13 [online]. Accessed 16 April 2015.

4 HTML5, W3C Recommendation [online]. 28 October 2014.
URL: http://www.w3.org/TR/html5.
Accessed 16 April 2015.

5 YouTube [online]. YouTube, LLC.
URL: http://www.youtube.com.
Accessed 16 April 2015.

6 Angry Birds [online]. Rovio Entertainment Ltd.
URL: http://chrome.angrybirds.com.
Accessed 16 April 2015.

7 HTTP Cookie [online]. Wikipedia.
URL: http://en.wikipedia.org/wiki/HTTP_cookie.
Accessed 16 April 2015.

8 Web Storage: W3C Recommendation [online]. 30 July 2013.
URL: http://www.w3.org/TR/webstorage.
Accessed 16 April 2015.

9 File API: W3C Working Draft [online]. 21 April 2015.
URL: http://www.w3.org/TR/FileAPI.
Accessed 16 April 2015.

10 JavaScript [online]. Wikipedia.
URL: http://en.wikipedia.org/wiki/JavaScript.
Accessed 16 April 2015.

11 MS Windows [online]. Microsoft Corporation.
URL: http://www.microsoft.com.
Accessed 16 April 2015.

12 Web Workers: W3C Candidate Recommendation [online]. 1 May 2012.
URL: http://www.w3.org/TR/workers.
Accessed 16 April 2015.

13 NAT Traversal [online]. Wikipedia.
URL: http://en.wikipedia.org/wiki/NAT_traversal.
Accessed 16 April 2015.

14 Baugher M, Carrara E, McGrew D, Näslund M, Norrman K. The Secure Real-time Transport Protocol (SRTP) [online].
URL: https://www.ietf.org/rfc/rfc3711.
Accessed 16 April 2015.

15 Rescorla E, Modadugu N. Datagram Transport Layer Security Version 1.2 [online].
URL: https://tools.ietf.org/rfc/rfc6347.
Accessed 16 April 2015.

16 Dierks T, Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.2 [online].
URL: https://tools.ietf.org/rfc/rfc5246.
Accessed 16 April 2015.

17 Fielding R, Gettys J, Mogul J, Nielsen H, Masinter L, Leach P, Berners-Lee T. Hypertext Transfer Protocol HTTP/1.1, Internet Standard [online].
URL: http://tools.ietf.org/html/rfc2616.
Accessed 16 April 2015.

18 Dynamic Web Page [online]. Wikipedia.
URL: http://en.wikipedia.org/wiki/Dynamic_web_page.
Accessed 16 April 2015.

19 XMLHttpRequest, W3C Working Draft [online]. 30 January 2014.
URL: http://www.w3.org/TR/XMLHttpRequest.
Accessed 16 April 2015.

20 Server-Sent Events: W3C Recommendation [online]. 3 February 2015.
URL: http://www.w3.org/TR/eventsource.
Accessed 16 April 2015.

21 Postel J. Transmission Control Protocol, Internet Standard [online].
URL: https://www.ietf.org/rfc/rfc793.
Accessed 16 April 2015.

22 Stewart R. Stream Control Transmission Protocol [online].
URL: https://tools.ietf.org/html/rfc4960.
Accessed 16 April 2015.

23 Postel J. User Datagram Protocol, Internet Standard [online].
URL: http://tools.ietf.org/html/rfc768.
Accessed 16 April 2015.

24 Tuexen M, Stewart R, Lei P. Padding Chunk and Parameter for the Stream Control Transmission Protocol (SCTP) [online].
URL: https://tools.ietf.org/html/rfc4820.
Accessed 16 April 2015.

25 Mathis M, Heffner J. Packetization Layer Path MTU Discovery [online].
URL: https://www.ietf.org/rfc/rfc4821.
Accessed 16 April 2015.

26 Stewart R, Tuexen M, Lei P. Stream Control Transmission Protocol (SCTP) Stream Configuration [online].
URL: https://tools.ietf.org/html/rfc6525.
Accessed 16 April 2015.

27 Stewart R, Ramalho M, Xie Q, Tuexen M, Contrad P. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension [online].
URL: https://tools.ietf.org/html/rfc3758.
Accessed 16 April 2015.

28 Tuexen M, Seggelmann R, Stewart R, Loreto S. Additional Policies for the Partial Reliability Extension of the Stream Control Transmission Protocol [online].
URL: https://tools.ietf.org/rfc/rfc7496.
Accessed 16 April 2015.

29 Stewart R, Tuexen M, Loreto S, Seggelmann R. Stream Schedulers and a New Data Chunk for the Stream Control Transmission Protocol [online].
URL: https://tools.ietf.org/html/draft-ietf-tsvwg-sctp-ndata-02.
Accessed 16 April 2015.

30 Tuexen M, Stewart R, Jessup R, Loreto S. DTLS Encapsulation of SCTP Packets [online].
URL: https://tools.ietf.org/html/draft-ietf-tsvwg-sctp-dtls-encaps-08.
Accessed 16 April 2015.

31 Jessup R, Loreto S, Tuexen M. WebRTC Data Channel Establishment Protocol [online].
URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol-09.
Accessed 16 April 2015.

32 Rosenberg J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols [online].
URL: https://tools.ietf.org/rfc/rfc5245.
Accessed 16 April 2015.

33 Handley M, Jacobson V, Perkins C. SDP: Session Description Protocol [online].
URL: https://tools.ietf.org/html/rfc4566.
Accessed 16 April 2015.

34 Rosenberg J, Mahy R, Matthews P, Wing D. Session Traversal Utilities for NAT (STUN) [online].
URL: https://tools.ietf.org/rfc/rfc5389.
Accessed 16 April 2015.

35 Mahy R, Matthews P, Rosenberg J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN) [online].
URL: https://tools.ietf.org/html/rfc5766.
Accessed 16 April 2015.

36 Nguyen D. Jump Start: Node.JS. Vic, Australia: SitePoint Pty, Ltd; 2012.

37 Geary D. Core HTML5 Canvas. Upper Saddle river, USA: Prentice Hall, Inc; 2012.

Appendix 1
Pong Source Code

webrtc_websocket_pong.html:

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8">
  <title>WebSocket/WebRTC Pong Demo</title>
  <style>
   canvas {
    position: absolute;
    top: 100px;
    left: 0px;
    background: transparent;
   }
   </style>
   <script src="webrtc_websocket_pong.js"></script>
  </head>
  <body>
   <input type="button" id="connect" value="CONNECT">
   <input type="button" id="start" value="START">
   <input type="button" id="exit" value="EXIT">
   <p id="dc_status"></p>
   <canvas id="canvas" width="300" height="300">
     Sorry, canvas not supported
   </canvas>
  </body>
</html>
```

webrtc_websocket_pong.js:

```
//Canvas parameter
var ctx;

//Communication parameters
var offerer = false;      //Role (offerer or answerer)

//WebRTC parameters
var myPC = null;          //WebRTC PeerConnection
var myDC = null;          //WebRTC DataChannel

// Game parameters
var interval_id;          //Game loop frequency
var bx=225;               //Ball x position
var by=25;                //Ball y position
var bdx=5;                //Ball x delta (per game loop)
var bdy=5;                //Ball y delta (per game loop)
var bx_old;               //Ball helper variable
var by_old;               //Ball helper variable
var b_rad = 5;            //Ball radius

var dy = 5;               //Shooter y delta (per key press)
var lx=5;                 //Left shooter x position
var ly=25;                //Left shooter y position
var l_points=0;           //Points (offerer)
var rx;                   //Right shooter x position
var ry=25;                //Right shooter y position
var r_points=0;           //Points (answerer)
var shooter_width = 10;   //Shooter width
var shooter_height = 100; //Shooter heights

//Browser specifics (Chrome)
RTCPeerConnection = webkitRTCPeerConnection;
RTCIceCandidate = window.RTCIceCandidate;
RTCSessionDescription = window.RTCSessionDescription;

//ICE STUN and TURN servers
var myServers = [];
var myServers = {
 'iceServers': [
 {'urls':'stun:stun.l.google.com:19302'},
 {'urls':'stun:stun1.l.google.com:19302'},
 {'urls':'stun:stun2.l.google.com:19302'},
 {'urls':'stun:stun3.l.google.com:19302'},
 {'urls':'stun:stun4.l.google.com:19302'},
 {'urls':'stun:stun01.sipphone.com'},
 {'urls':'stun:stun.ekiga.net'},
 {'urls':'stun:stun.fwdnet.net'},
 {'urls':'stun:stun.ideasip.com'},
 {'urls':'stun:stun.iptel.org'},
 {'urls':'stun:stun.rixtelecom.se'},
 {'urls':'stun:stun.schlund.de'},
 {'urls':'stun:stunserver.org'},
 {'urls':'stun:stun.softjoys.com'},
 {'urls':'stun:stun.voiparound.com'},
 {'urls':'stun:stun.voipbuster.com'},
 {'urls':'stun:stun.voipstunt.com'},
 {'urls':'stun:stun.voxgratia.org'},
 {'urls':'stun:stun.xten.com'},
 {'urls':'turn:numb.viagenie.ca:3478','credential':'kTTptrn','username':'chris-
ter.holmberg@ericsson.com'},
 {'urls':'turn:numb.viagenie.ca:3478;transport=tcp','creden-
tial':'kTTptrn','username':'christer.holmberg@ericsson.com'}
 ]
}

var myConstraints = {
 'mandatory' :
 {
  'OfferToReceiveAudio' : false,
  'OfferToReceiveVideo' : false,
  'iceTransports': 'relay'
 }
}

//Generate unique browser identifier
var myBrowserId = Math.random().toString().replace('.', '');

//Create WebSocket connection
var ws = new WebSocket("ws://88.129.244.10:1234","echo-protocol");

//INIT FUNCTION
//=============

function init()
{
  var canvas = document.getElementById("canvas");
  if (canvas.getContext) {
   ctx = canvas.getContext("2d");
   paint(true);
   rx=canvas.width-shooter_width-5;
   window.onkeydown = keypressed;
   document.querySelector("#connect").onclick = sendOffer;
   document.querySelector("#start").disabled = false;
   document.querySelector("#start").onclick = startGame;
   document.querySelector("#start").disabled = true;
   document.querySelector("#exit").onclick = exitGame;
   document.querySelector("#exit").disabled = false;
  }
}

//WEBSOCKET FUNCTIONS
//===================

ws.onmessage = function(e){
 var msg = e.data;
 var wsJSONObj = JSON.parse(e.data);

 if(wsJSONObj.id == myBrowserId){
  console.log("Received own WebSocket message");
 }else{
  console.log("Received WebSocket message: " + msg);
  switch(wsJSONObj.type){
   case "offer":
    if(!offerer){
     handleOffer(wsJSONObj);
    }
    break;
   case "answer":
    if(offerer){
     handleAnswer(wsJSONObj);
    }
    break;
   case "candidate":
    handleCandidate(wsJSONObj);
    break;
   default:
    console.log("Received unknown WebSocket message");
  }
 }
}

ws.onclose= function(e){
 console.log("WebSocket disconnected");
}

ws.onerror= function(e){
 console.log("Error");
}

ws.onopen= function(e){
 console.log("WebSocket open");
}

function sendOffer(){
 document.querySelector("#connect").disabled = true;
 myPC = new RTCPeerConnection(myServers);
 offerer = true;

 createDC();

 myPC.createOffer(function(sdp) {
  myPC.setLocalDescription(sdp);
  var wsOfferJSON = {
   id:myBrowserId,
   type:"offer",
   data:sdp
  };
  var wsOfferString = JSON.stringify(wsOfferJSON);
  console.log('Sending offer:' + wsOfferString);
  ws.send(wsOfferString);
 });

 myPC.onicecandidate = function(event) {
  var candidate = event.candidate;
  if (candidate) {
   var wsCandJSON = {
    id:myBrowserId,
    type:"candidate",
    data:candidate
   };
   var wsCandString = JSON.stringify(wsCandJSON);
   console.log('Sending ICE candidate (offerer): ' + wsCandString);
   ws.send(wsCandString);
  } else {
   console.log('ICE candidate error (offerer)');
  }
 }
}

function handleOffer(objJSON){
 document.querySelector("#connect").disabled = true;
 var sdp = objJSON.data;
 console.log('Creating PeerConnection (answerer)');
 myPC = new RTCPeerConnection(myServers);
 createDC();
 myPC.setRemoteDescription(new RTCSessionDescription(sdp));

 myPC.createAnswer(function(sdp) {
  myPC.setLocalDescription(sdp);
  wsAnswerJSON = {
   id:myBrowserId,
   type:"answer",
   data:sdp
  };
  wsAnswerString = JSON.stringify(wsAnswerJSON);
  console.log("Sending answer: " + wsAnswerString);
  ws.send(wsAnswerString);
 });

 myPC.onicecandidate = function(event) {
  var candidate = event.candidate;
  if (candidate) {
   var wsCandJSON = {
    id:myBrowserId,
    type:"candidate",
    data:candidate
   };
   var wsCandString = JSON.stringify(wsCandJSON);
   console.log('Sending ICE candidate (answerer): ' + wsCandString);
   ws.send(wsCandString);
  }else{
   console.log('ICE candidate error');
  }
 }
}

function handleAnswer(objJSON) {
 var sdp = objJSON.data;
 myPC.setRemoteDescription(new RTCSessionDescription(sdp));
}

function handleCandidate(objJSON) {
 var candidate = objJSON.data;
 myPC.addIceCandidate(new RTCIceCandidate(candidate));
}

//DATA CHANNEL FUNCTIONS
//======================

function createDC() {
 if(!offerer){
  console.log('Answerer waiting for DataChannel');
  myPC.ondatachannel = eventDC;
 }else{
  console.log('Offerer creating DataChannel');
  myDC = myPC.createDataChannel('myDataChannel');
  myDC.onmessage = eventDCMessage;
  myDC.onopen = eventDCOpen;
  myDC.onclose = eventDCClosed;
  myDC.onerror = eventDCError;
 }
}

function eventDC(event) {
 myDC = event.channel;
 myDC.onmessage = eventDCMessage;
 myDC.onopen = eventDCOpen;
 myDC.onclose = eventDCClosed;
}

function eventDCMessage(event) {
 var dcJSON = JSON.parse(event.data);

 if(offerer)
 {
  ry = dcJSON.y;
 }else{
  ly = dcJSON.y;
  bx = dcJSON.bx;
  by = dcJSON.by;
  l_points = dcJSON.lpts;
  r_points = dcJSON.rpts;

  paint(false);

  var dcJSONTx = {y:ry};
  var dcJSONStringTx = JSON.stringify(dcJSONTx);
  if(myDC.readyState == "open") {
   myDC.send(dcJSONStringTx);
  }
 }
}

function eventDCOpen() {
 document.getElementById("dc_status").innerHTML = "Data channel open";
 if(offerer){
  document.querySelector("#start").disabled = false;
 }
}

function eventDCClosed() {
 if(ws)
 {
  ws.close();
 }
 clearInterval(interval_id);
 paint(true);
 document.getElementById("dc_status").innerHTML = "Application closed";
 document.querySelector("#connect").disabled = true;
 document.querySelector("#start").disabled = true;
 document.querySelector("#exit").disabled = true;
}

function eventDCError(event) {
 if(ws)
 {
  ws.close();
 }
 clearInterval(interval_id);
 paint(true);
 document.getElementById("dc_status").innerHTML = "Data channel error";
 document.querySelector("#connect").disabled = true;
 document.querySelector("#start").disabled = true;
 document.querySelector("#exit").disabled = true;
```

```
}

//GAME FUNCTIONS
//==============

function keypressed(e) {
 // up key
 if(e.keyCode == 38){
  if(offerer){
   ly = ly - dy;
  }else{
   ry = ry - dy;
  }
 }
 // down key
 if(e.keyCode == 40){
  if(offerer){
   ly = ly + dy;
  }else{
   ry = ry + dy;
  }
 }
}

function startGame()
{
 if(offerer) {
  document.querySelector("#start").disabled = true;
  interval_id = setInterval(position, 25); //Start game loop
 }
}

function exitGame() {
 if(ws){
  console.log("Closing web socket");
  ws.close();
 }
 if(myDC){
  myDC.close();
 }else{
  eventDCClosed();
 }
 //clearInterval(interval_id); //Clear game loop interval
}

function paint(background_only)
{
 // Draw canvas background
 ctx.fillStyle = "#000000";
 ctx.fillRect (0,0,canvas.width,canvas.height);

 if(!background_only){
  // Draw ball
  ctx.fillStyle = "#FFFFFF";
  ctx.beginPath();
  ctx.arc(bx,by,b_rad,0,2 * Math.PI,false);
  ctx.fill();

  // Draw left shooter
  if(offerer){
   ctx.fillStyle = "#FF00FF";
  }else{
   ctx.fillStyle = "#FFFFFF";
  }
  ctx.fillRect(lx,ly,shooter_width,shooter_height);

  // Draw right shooter
  if(offerer){
   ctx.fillStyle = "#FFFFFF";
  }else{
   ctx.fillStyle = "#FF00FF";
  }
  ctx.fillRect(rx,ry,shooter_width,shooter_height);

  // Print points
  ctx.font="50px Arial";
  ctx.fillStyle = "#FFFFFF";
  ctx.fillText(l_points,(canvas.width/2-40),40);
  ctx.fillText(r_points,(canvas.width/2+30),40);
 }
}

function collision(){
 //Check if ball hits left shooter
 if((bx-b_rad <= lx+shooter_width) && (by+b_rad >= ly) && (by-b_rad <= ly+shooter_height))
 {
  if(bx_old-b_rad >= (lx+shooter_width))
  {
   bx = bx_old;
   bdx = -bdx;
  }else{
   by = by_old;
   bdy = -bdy;
  }
 }

 //Check if ball hits right shooter
 if((bx+b_rad >= rx) && (by+b_rad >= ry) && (by-b_rad <= ry+shooter_height))
 {
  if((bx_old+b_rad) <= rx)
  {
   bx = bx_old;
   bdx = -bdx;
  }else{
   by = by_old;
   bdy = -bdy;
  }
 }
}

function position(){

 // Ball position
 bx_old = bx;
 by_old = by;
 bx = bx + bdx;
 by = by + bdy;

 // Check collision between ball and shooters
 collision();

 //Check if ball hits boundaries
 if(bx-b_rad<=0)
 {
  r_points++;
  bdx=-bdx;
 }
 if ((bx+b_rad)>=canvas.width)
 {
  l_points++;
  bdx=-bdx;
 }
 if ((by+b_rad<= 0) || ((by+b_rad)>=canvas.height))
 {
  bdy=-bdy;
 }

 //Check if left shooter hits boundaries
 if (ly <= 0)
 {
  ly = 0;
 }
 if ((ly+shooter_height) >= canvas.height)
 {
  ly = (canvas.height-shooter_height);
 }
 // Check if right shooter hits boundaries
 if (ry <= 0)
 {
  ry = 0;
 }
 if ((ry+shooter_height) >= canvas.height)
 {
  ry = (canvas.height-shooter_height);
 }

 //Send coordinates and points to remote peer
 var dcCordOffJSON = {
  y:ly,
  bx:bx,
  by:by,
  lpts:l_points,
  rpts:r_points
 };

 var dcCordOffString = JSON.stringify(dcCordOffJSON);
 if(myDC.readyState == "open"){
  myDC.send(dcCordOffString);
 }
 paint(false);
}

onload=init;
```

Appendix 2
File transfer source code

webrtc_websocket_fileshare.html

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8">
  <title>WebSocket/WebRTC Fileshare Demo</title>
  <script src="webrtc_websocket_fileshare.js"></script>
 </head>
 <body>
  <input type="button" id="connect" value="Connect">
  <input type="button" id="send" value="Send file">
  <input type="button" id="exit" value="Exit">
  <input type="file" id="my_file" value="Browse">
  <p id="dc_status"></p>
  <p id="tx_status"></p>
  <p id="rx_status"></p>
 </body>
</html>
```

webrtc_websocket_fileshare.js

```
//Communication parameters

var offerer = false;    // Role (offerer or answerer)
var myPC = null;        // WebRTC PeerConnection
var myDC = null;        // WebRTC DataChannel

// File parameters

var myFileReader = new window.FileReader(); // File reader
var chunkLength = 1000;                     // Chunk length
var file;                                   // Input file
var chunkTxCounter;                         // Chunk counter (Tx)
var chunkRxCounter;                         // Chunk counter (Rx)

var chunkArray = [];                        // Buffer to store received file
var fileName;                               // Name of transmitted file

//Browser specifics (Chrome)
RTCPeerConnection = webkitRTCPeerConnection;
RTCIceCandidate = window.RTCIceCandidate;
RTCSessionDescription = window.RTCSessionDescription;
```

```
//ICE STUN and TURN servers
var myServers = [];
var myServers = {
 'iceServers': [
  {'urls':'stun:stun.l.google.com:19302'},
  {'urls':'stun:stun1.l.google.com:19302'},
  {'urls':'stun:stun2.l.google.com:19302'},
  {'urls':'stun:stun3.l.google.com:19302'},
  {'urls':'stun:stun4.l.google.com:19302'},
  {'urls':'stun:stun01.sipphone.com'},
  {'urls':'stun:stun.ekiga.net'},
  {'urls':'stun:stun.fwdnet.net'},
  {'urls':'stun:stun.ideasip.com'},
  {'urls':'stun:stun.iptel.org'},
  {'urls':'stun:stun.rixtelecom.se'},
  {'urls':'stun:stun.schlund.de'},
  {'urls':'stun:stunserver.org'},
  {'urls':'stun:stun.softjoys.com'},
  {'urls':'stun:stun.voiparound.com'},
  {'urls':'stun:stun.voipbuster.com'},
  {'urls':'stun:stun.voipstunt.com'},
  {'urls':'stun:stun.voxgratia.org'},
  {'urls':'stun:stun.xten.com'},
  {'urls':'turn:numb.viagenie.ca:3478','credential':'kTTptrn','username':'chris-
ter.holmberg@ericsson.com'},
  {'urls':'turn:numb.viagenie.ca:3478;transport=tcp','creden-
tial':'kTTptrn','username':'christer.holmberg@ericsson.com'}
 ]
}

var myConstraints = {
 'mandatory' :
 {
  'OfferToReceiveAudio' : false,
  'OfferToReceiveVideo' : false,
  'iceTransports': 'relay'
 }
}

//Generate unique browser identifier
var myBrowserId = Math.random().toString().replace('.', '');

//Create WebSocket connection
var ws = new WebSocket("ws://88.129.244.10:1234","echo-protocol");

//INIT FUNCTION
//=============

function init()
```

```
{
 document.getElementById("dc_status").innerHTML = "Data channel closed";
 document.getElementById("tx_status").innerHTML = "No data sent";
 document.getElementById("rx_status").innerHTML = "No data received";
 document.querySelector("#connect").onclick = sendOffer;
 document.querySelector("#connect").disabled = false;
 document.querySelector("#send").onclick = sendFile;
 document.querySelector("#send").disabled = true;
 document.querySelector("#exit").onclick = leaveApp;
 document.querySelector("#exit").disabled = false;
 document.querySelector("#my_file").disabled = true;
 document.querySelector("#my_file").onchange = function() {
  file = this.files[0];
  console.log('Selected file name: ' + file.name);
  document.querySelector("#send").disabled = false;
 };
 myFileReader.onload = onReadFile;
}

//WEBSOCKET FUNCTIONS
//===================

ws.onmessage = function(e){
 var msg = e.data;
 var wsJSONObj = JSON.parse(e.data);

 if(wsJSONObj.id == myBrowserId){
  console.log("Received own WebSocket message");
 }else{
  console.log("Received WebSocket message: " + msg);
  switch(wsJSONObj.type){
   case "offer":
    if(!offerer){
     handleOffer(wsJSONObj);
    }
    break;
   case "answer":
    if(offerer){
     handleAnswer(wsJSONObj);
    }
    break;
   case "candidate":
    handleCandidate(wsJSONObj);
    break;
   default:
    console.log("Received unknown WebSocket message");
  }
 }
}
```

```
ws.onclose= function(e){
 console.log("WebSocket disconnected");
}

ws.onerror= function(e){
 console.log("Error");
}

ws.onopen= function(e){
 console.log("WebSocket open");
}

function sendOffer(){
 document.querySelector("#connect").disabled = true;
 myPC = new RTCPeerConnection(myServers);
 offerer = true;

 createDC();

 myPC.createOffer(function(sdp) {
  myPC.setLocalDescription(sdp);
  var wsOfferJSON = {
   id:myBrowserId,
   type:"offer",
   data:sdp
  };
  var wsOfferString = JSON.stringify(wsOfferJSON);
  console.log('Sending offer:' + wsOfferString);
  ws.send(wsOfferString);
 });

 myPC.onicecandidate = function(event) {
  var candidate = event.candidate;
  if (candidate) {
   var wsCandJSON = {
    id:myBrowserId,
    type:"candidate",
    data:candidate
   };
   var wsCandString = JSON.stringify(wsCandJSON);
   console.log('Sending ICE candidate (offerer): ' + wsCandString);
   ws.send(wsCandString);
  } else {
   console.log('ICE candidate error (offerer)');
  }
 }
}
```

```
function handleOffer(objJSON){
 document.querySelector("#connect").disabled = true;
 var sdp = objJSON.data;
 console.log('Creating PeerConnection (answerer)');
 myPC = new RTCPeerConnection(myServers);
 createDC();
 myPC.setRemoteDescription(new RTCSessionDescription(sdp));

 myPC.createAnswer(function(sdp) {
  myPC.setLocalDescription(sdp);
  wsAnswerJSON = {
   id:myBrowserId,
   type:"answer",
   data:sdp
  };
  wsAnswerString = JSON.stringify(wsAnswerJSON);
  console.log("Sending answer: " + wsAnswerString);
  ws.send(wsAnswerString);
 });

 myPC.onicecandidate = function(event) {
  var candidate = event.candidate;
  if (candidate) {
   var wsCandJSON = {
    id:myBrowserId,
    type:"candidate",
    data:candidate
   };
   var wsCandString = JSON.stringify(wsCandJSON);
   console.log('Sending ICE candidate (answerer): ' + wsCandString);
   ws.send(wsCandString);
  }else{
   console.log('ICE candidate error');
  }
 }
}

function handleAnswer(objJSON) {
 var sdp = objJSON.data;
 myPC.setRemoteDescription(new RTCSessionDescription(sdp));
}

function handleCandidate(objJSON) {
 var candidate = objJSON.data;
 myPC.addIceCandidate(new RTCIceCandidate(candidate));
}

//DATA CHANNEL FUNCTIONS
//======================
function createDC() {
 if(!offerer){
  console.log('Answerer waiting for DataChannel');
  myPC.ondatachannel = eventDC;
 }else{
  console.log('Offerer creating DataChannel');
  myDC = myPC.createDataChannel('myDataChannel');
  myDC.onmessage = eventDCMessage;
  myDC.onopen = eventDCOpen;
  myDC.onclose = eventDCClosed;
  myDC.onerror = eventDCError;
 }
}

function eventDC(event) {
 myDC = event.channel;
 myDC.onmessage = eventDCMessage;
 myDC.onopen = eventDCOpen;
 myDC.onclose = eventDCClosed;
}

function eventDCMessage(event) {
 document.getElementById("rx_status").innerHTML = "Receiving data";
 var recvJSONObj = JSON.parse(event.data);

 if(recvJSONObj.name){
  console.log('Received file name: ' + recvJSONObj.name);
  fileName = recvJSONObj.name;
  chunkRxCounter = 0;
 }

 chunkArray.push(recvJSONObj.message);
 chunkRxCounter++;
 console.log("Received chunk #: " + chunkRxCounter);

 if(recvJSONObj.last){
  console.log("Last chunk received");
  saveToDisk(chunkArray.join(''),fileName);
  document.getElementById("rx_status").innerHTML = "No data received";
  document.querySelector("#my_file").disabled = false;
  chunkArray = [];
 }
}

function eventDCOpen() {
 document.getElementById("dc_status").innerHTML = "Data channel open";
 document.querySelector("#my_file").disabled = false;
}
```

```
function eventDCClosed() {
 chunkArray = [];
 if(ws)
 {
  ws.close();
 }
 document.getElementById("dc_status").innerHTML = "Application closed";
 document.getElementById("rx_status").innerHTML = "No data received";
 document.getElementById("tx_status").innerHTML = "No data sent";
 document.querySelector("#connect").disabled = true;
 document.querySelector("#send").disabled = true;
 document.querySelector("#exit").disabled = true;
 document.querySelector("#my_file").disabled = true;
}

function eventDCError(e) {
 document.getElementById("dc_status").innerHTML = "Data channel error";
 document.getElementById("rx_status").innerHTML = "No data received";
 document.getElementById("tx_status").innerHTML = "No data sent";
}

function sendFile() {
 document.getElementById("tx_status").innerHTML = "Sending data";
 document.querySelector("#send").disabled = true;
 document.querySelector("#my_file").disabled = true;
 myFileReader.readAsDataURL(file);
}

function leaveApp() {
 if(myDC){
  myDC.close();
 }else{
  eventDCClosed();
 }
}

function onReadFile(event, text) {
 var data = {};
 if(myDC.readyState == "open"){
  if (event){
   text = event.target.result;
   data.name = file.name;
   chunkTxCounter = 0;
  }

  if (text.length > chunkLength) {
   data.message = text.slice(0, chunkLength);
  }else{
   data.message = text;
   data.last = true;
   console.log("Last chunk to be sent");
   document.getElementById("tx_status").innerHTML = "No data sent";
   document.querySelector("#my_file").disabled = false;
  }
  chunkTxCounter++;

  var sendJSONString = JSON.stringify(data);
  console.log("Sent chunk #: " + chunkTxCounter);
  myDC.send(sendJSONString);

  var remainingDataURL = text.slice(data.message.length);
  if (remainingDataURL.length) setTimeout(function () {
   onReadFile(null, remainingDataURL);
  }, 500)
 }else{
  console.log("Data channel closed - cannot send");
 }
}

//FILE SAVE FUNCTION
//==================

function saveToDisk(fileUrl, fileName) {
 var save = document.createElement('a');
 save.href = fileUrl;
 save.target = '_blank';
 save.download = fileName || fileUrl;
 var event = document.createEvent('Event');
 event.initEvent('click', true, true);
 save.dispatchEvent(event);
 (window.URL || window.webkitURL).revokeObjectURL(save.href);
}

onload=init;
```