What Is Real-Time Communications?

Real-time communications (RTC) are the near simultaneous exchange of information over any type of telecommunications service from the sender to the receiver in a connection with negligible latency, according to SearchUnified Communications. Examples of real-time communications include:

- Voice over landlines and mobile phones
- VoIP
- Instant messaging (e.g., WhatsApp, WeChat, Facebook Messenger)
- Video and teleconferencing
- Robotic telepresence

It may not use the term "real-time communications," but Vonage's most recent Global Customer Engagement Report nevertheless has a lot of good to say about the concept. It's easy to see areas in the report — which breaks down customer preferences in 11 different communication mediums — where RTC stands out. First, consider the biggest positive changes noted in the chart below.

While customers have trended away from mobile and landline calls, that doesn't mean their preferences have also moved away from real-time contact. Indeed, every item that saw an increase on the list included a real-time component. And yes, that statement includes chatbots, since you're still communicating with *something* in real time.

However, real-time communications does not include email, bulletin boards, blogs or other forms of internet communication where users read content without regard to the time the sender posts it, which can create a significant delay between data transmission and reception.

In addition, real-time communications is never stored in an interim state anywhere between the transmitter and the receiver, as defined by Techopedia. Within that context, think of “real-time communications” as a synonym for “live communications.”

In a communication of this type, there is a return path where the receiver can also communicate with the sender in real time. Real-time communications can take place in half duplex or full duplex:

- Half duplex—communications in one channel in one direction at a time. Sender or receiver can send but not receive at the same time (think of an old-style walkie-talkie where the user has to say “over” to tell the person on the other end that she can speak)
- Full duplex—sender and receiver can send and receive messages simultaneously in two parallel communication pathways

Why Are Real-Time Communications Important?

In short, customers today expect immediacy. That's true whether the customer comes to the company seeking information, support, or sales engagement, and no matter what the format. For example, a passenger who has missed a connecting flight may contact the reservations center via the airline's app or a phone call while stranded at the intervening hub. But they likely won't send an email and hang around waiting for a reply.

RTC also stands out because of its ability to provide information, service, or sales engagement in the right context and with the right timing. In the above example, for instance, measures that push context-important data to agents as the customer calls in can reduce the number of times the customer must repeat their story before achieving resolution. And this matters: Consumers told Vonage that one of their biggest communications hurdles was repeating themselves to different people.

In this context, those systems could:

- Show what section of the app the customer placed a call from
- Automatically populate relevant data, especially that on a timer, such as the customer's flight agenda
- Allow agents to chat — another RTC medium — in real time during handoff, ensuring the right information is available quickly

Naturally, achieving this level of responsiveness in any field takes some time, planning, and investment.

Real-Time Communications for Business

Fast Wi-Fi and mobile data options (and the powerful phones that run on the networks) have made possible a new generation of business apps: network- and provider-agnostic services that allow customers to chat and call directly within the app, with voice data traveling over the data network the device happens to be connected to at the time.

We've seen the market use these expanded capabilities in increasingly creative ways. In Southeast Asia, a real-time ride-hailing app called Grab goes over the top of unreliable local carrier infrastructure, allowing customers to interact directly with the business through voice messages delivered over a Voice API. In turn, this allows the data to travel over a worldwide, low-latency cloud network, adding a greater layer of reliability and service quality and helping overcome the market's unique challenges by proxy.

Of course, there are plenty more business use cases for RTC. In workplaces without a traditional office space, such as restaurants and retailers, RTC can help on-the-go employees, who probably don't have time to respond to a traditional email but can do a quick IM or call from the floor if needed. In other workplaces, real-time messaging within a company app can replace mass-emailing staff organizational announcements, a move

Vonage's report indicates could increase employee engagement. Remember, employees are consumers, too, and their preferences don't exist in a commerce-only vacuum.

The Future of RTC

As the Grab example above shows, the impact of RTC can be measured in two varieties: technologies that end users interface with (apps) and technologies that allow those end-user products to do the things they do by enabling communication on the backend (APIs; cloud).

That last point is important because it gives RTC a strong appeal among developers, IT managers, product managers, and entrepreneurs, among others. APIs and their popularity among businesses are one reflection of this trend; so, too, is the rise of unified communications platforms, which enable companies to build RTC directly into existing systems and workflows.

Because of this, it's (again) important not to think of RTC as solely a business- or customer-facing concept. In general, the faster we can share information with someone requesting it, the happier, more productive, and more efficient they'll be. In the workplace, this could mean small differences, such as an employee choosing to use IM instead of email for a quick question, or large ones, such as a company overhauling its contact center's unified communications platform.

It's ultimately up to the individual business to decide how it can benefit from RTC — but the use cases are almost certainly there. How would your organization look with a renewed focus on real time?

What is WebRTC?
WebRTC is an HTML5 specification that you can use to add real time media communications directly between browser and devices.
Simply put:
WebRTC enables for voices and video communication to work inside web pages.
And you can do that without the need of any prerequisite of plugins to be installed in the browser.
It was announced in 2011 and since then it has steadily grown in popularity and adoption.
By 2016 there has been an estimate from 2 billion browsers installed that are enabled to work with WebRTC. From traffic perspective, it has seen an estimate of over a billion minutes and 500 terabytes of data transmitted every week from browser communications alone.
WebRTC has increased in popularity and use throughout the COVID-19 pandemic. Quarantines and work from home made remote communications a necessity, indoctrinating billions of users about the use of video calling. The ***end result has been a surge in the use of WebRTC:***
he growth in use of WebRTC during the COVID-19 pandemic
In 2021 WebRTC got officially standardized, removing all doubts about its future prospects. Today, WebRTC is widely popular for video calling but it is capable of so much more.
A few things worth mentioning:

- WebRTC is completely free
- It comes as open source project that has been embedded in browsers but you can take and adopt it for your own needs
- This in turn has created a vibrant and dynamic ecosystem around WebRTC of a variety of open source projects and frameworks as well as commercial offerings from companies that help you to build your products
- WebRTC constantly evolving and improving, so you need to keep an eye on it (e.g. see hiring WebRTC developers)
- See also: The state of WebRTC open source projects

WebRTC's meaning
WebRTC stands for Web Real-Time Communications.
Web is simple – it means that what we are doing works "over the web" and inside a browser. The browser part means that all modern browsers support WebRTC. If you run this inside a native application I will still be considering it as WebRTC. To me, it is the thought that counts, or more accurately, the implementation of WebRTC (or parts of it) are quite popular as a starting point in native applications. This is due to

the quality of the WebRTC media engine (as implemented by Google) AND due to the fact that it makes it easier to communicate across native applications and web applications this way.
RTC, or Real-Time Communications means that whatever WebRTC does – it does in real time. Its focus is on sending the data it has as fast as possible – making sure to use low latency techniques to get things done. Whenever possible.
If we're moving away a bit from the word-meaning of WebRTC, then this is the definition I usually use to define WebRTC:

Lets break it down a bit:

- WebRTC offers real time communication natively from a web browser
    - WebRTC is part of the web browser. Every modern web browser today implements WebRTC
    - It offers the ability to create real time communication applications and experiences
- WebRTC is a media engine with JavaScript APIs
    - WebRTC is a media engine. There were other media engines before WebRTC and there will probably be others after it. In that sense, there's no "innovation" here
    - That said, it is standardized by an API layer defined in JavaScript. This contributes to the ecosystem that has been created around WebRTC

So, how does WebRTC work?

Code and API

It is important to understand from where we are coming from: If you wanted to build anything that allowed for voice or video calling a few years ago, you were most probably used C/C++ for that. This means long development cycles and higher development costs.
WebRTC changes all that: it takes the need for C/C++ and replace it with a Javascript API.
It comes with a Javascript API layer on the top that you can use inside the browser. This makes it far easier to develop and integrate real time communications anywhere.
Internally, WebRTC is still mostly implemented using C/C++, but most developers that use WebRTC won't need to dig deep into these layers in order to develop their applications.

Availability

WebRTC today is available in all modern browsers. Google Chrome, Mozilla Firefox, Apple Safari and Microsoft Edge support it.

You can also “take” it and integrate it into an application or an embedded device without the need of browser at all.

Browsers and operating system support for WebRTC

Media and access

What WebRTC does is allow the access to devices. You can access the microphone of your device, the camera that you have on your phone or laptop – or it can be a screen itself. You can capture the display of the user and then have that screen shared or recorded remotely.
Whatever it does is in real time, enabling live interactions.
WebRTC isn’t limited to voice and video. It allows sending any type of arbitrary data.

There are several reasons WebRTC is a great choice for real time communications

1. First of all, WebRTC is an open source project
    - It is completely free for commercial or private use, so why not use it?
    - Since it is constantly evolving and improving, you are banking on a technology that would service you for years to come
    - WebRTC is a pretty solid choice – It already created a vibrant ecosystem around it of different vendors and companies that can assist you with your application
2. It is available in all modern browsers
    - This has enabled and empowered the creation of new use cases and business models
    - From taking a Guitar or a Yoga lesson – to cloud gaming and social networking – to medical clowns or group therapy – to hosting large scale professional Webinars and live broadcasts; WebRTC is capable of serving all of them and more
3. WebRTC is not limited to only browsers because it is also available for mobile applications
    - The source code is portable and has been used already in a lot of mobile apps
    - SDKs are available for both mobile and embedded environments so you can use WebRTC to run anywhere
4. WebRTC is not only about for voice or video calling
    - It is quite powerful and versatile
    - You can use it to build a group calling service, add recording to it or use it only for data delivery

        - It is up to you to decide what to do with WebRTC
5. WebRTC takes the notion of a communication service and downgrades it into a feature inside a different type of service. So now you can take it and simply add communication in business processes you need within your application or business

What is WebRTC used for?

You can group WebRTC applications into 4 broad categories:

1. **Conversational voice and video** – the obvious one. Applications that need the ability to have a person communicate with others in real time and in a conversational manner. These will more often than not end up using WebRTC
2. **Live streaming** – while WebRTC isn't the most popular choice for streaming, it is one of the best technologies available for low-latency live streaming. If you need to stream something to one or more users and maintain really low latency to enhance the interactivity (things like cloud gaming, gambling, auctions, webinars, etc) – then WebRTC might be a great choice
3. **Data transfer** – you can send voice and video with WebRTC, but you can also send arbitrary data. This can be used to share huge files between machines with little need for server space for example. Or it can be used to create a bittorrent like experience
4. **Privacy** – since WebRTC runs direct between browsers, it is sometimes used to enhance privacy. Doing this by simply not sending the media or data via servers at all

Overview of Use-Cases with WebRTC

The use cases where WebRTC comes in handy seem endless. Every so often, I hear of a new way that WebRTC is being used to solve yet another problem.
Here are some of the main use cases you'll find for WebRTC out there:

- Unified communications – voice and video calling, 1:1 or group sessions
- Contact center communications – client/agent, visual assistance, remote assistance, etc
- Watch parties – watch television or a sports event together
- eCommerce and retail – from one-to-one high touch sales to live broadcast for sales events and promotions
- Telehealth, online education, legal proceedings, remote travel, fitness, dancing, tutoring, coaching, … – conduct remotely and virtually verticalized sessions you would have done in-person in the past

- Teleoperations – drive cars, forklifts, trucks, drones, boats, submarines, … – remotely
- Virtual and hybrid events – conduct webinars, large meetings and events online
- Low latency broadcasting – broadcast a sports game, auction or interactive sessions to a large audience at sub second latency
- Cloud gaming – render the visuals of a game in the cloud and send it in realtime to the player
- Machine remoting – operate a remote machine (high performance machines or highly secured/configured machines) as if it was a local one
- Virtual spaces and the metaverse – meet people in a synthetically rendered virtual environment in 2D or 3D

So what other choice do you really have besides using it?
The idea around WebRTC and what you can use it for are limitless. So go on – start building whatever you need!

WebRTC vs. WebSocket: Key differences and which to use

ebSocket and WebRTC are key technologies for building modern, low-latency web apps. This blog post explores the differences between the two. We'll cover the following:

- What is WebRTC?
- What is WebSocket?
- What are WebRTC's pros and cons?
- What are the advantages and disadvantages of WebSocket?
- What are the key differences between WebRTC and WebSocket?
- When to use WebRTC?
- When to use WebSocket?
- When to use WebRTC and WebSockets together?

What is WebRTC?

Web Real-Time Communication (WebRTC) is a framework that enables you to add real time communication (RTC) capabilities to your web and mobile applications. WebRTC allows the *transmission of arbitrary data (video, voice, and generic data) in a peer-to-peer fashion.*

Almost every modern browser supports WebRTC. Additionally, there are WebRTC SDKs targeting different platforms, such as iOS or Android.

WebRTC consists of several interrelated APIs. Here are the key ones:

- `RTCPeerConnection`. Allows you to connect to a remote peer, maintain and monitor the connection, and close it once it has fulfilled its purpose.
- `RTCDataChannel`. Provides a bi-directional network communication channel that allows peers to transfer arbitrary data.
- `MediaStream`. Designed to let you access streams of media from local input devices like cameras and microphones. It serves as a way to manage actions on a data stream, like recording, sending, resizing, and displaying the stream's content.

What is WebSocket?

WebSocket is a realtime technology that enables *full-duplex, bi-directional communication between a web client and a web server over a persistent, single-socket connection.*

A WebSocket connection starts as an HTTP request/response handshake. If this initial handshake is successful, the client and server have agreed to use the existing TCP connection that was established for the HTTP request as a WebSocket connection. This connection is kept alive for as long as needed (in theory, it can last forever), allowing the server and the client to independently send data at will.
The WebSocket technology includes two core building blocks:

- The WebSocket protocol. Standardized in December 2011 through RFC 6455, the WebSocket protocol enables realtime communication between a WebSocket client and a WebSocket server over the web. It supports transmission of binary data and text strings.
- The WebSocket API. Allows you to perform necessary actions, like managing the WebSocket connection, sending and receiving messages, and listening for events triggered by the WebSocket server. Almost all modern web browsers support the WebSocket API.

WebRTC: pros and cons

WebRTC advantages

- WebRTC apps provide strong security guarantees; data transmitted over WebRTC is encrypted and authenticated with the help of theSecure Real-Time Transport Protocol (SRTP).
- WebRTC is open-source and free to use. The project is backed by a strong and active community, and it's supported by organizations such as Apple, Google, and Microsoft.
- WebRTC is platform and device-independent. A WebRTC application will work on any browser that supports WebRTC, irrespective of operating systems or the types of devices.

WebRTC disadvantages

- Even though WebRTC is a peer-to-peer technology, you still have to manage and pay for web servers. For two peers to talk to each other, you need to use a signaling server to set up, manage, and terminate the WebRTC communication session. In one-to-many WebRTC broadcast scenarios, you'll probably need a WebRTC media server to act as a multimedia middleware.
- WebRTC can be extremely CPU-intensive, especially when dealing with video content and large groups of users. This makes it costly and hard to reliably use and scale WebRTC applications.
- WebRTC is hard to get started with. There are plenty of concepts you need to explore and master: the various WebRTC interfaces, codecs & media processing, network address translations (NATs) & firewalls, UDP (the main underlying communications protocol used by WebRTC), and many more.

WebSocket: pros and cons

WebSocket advantages

- Before WebSocket, HTTP techniques like AJAX long polling and Comet were the standard for building realtime apps. Compared to HTTP, WebSocket eliminates the need for a new connection with every request, drastically reducing the size of each message (no HTTP headers). This helps save bandwidth, improves latency, and makes WebSockets less taxing on the server side compared to HTTP.
- Flexibility is ingrained into the design of the WebSocket technology, which allows for the implementation of application-level protocols and extensions for additional functionality (such as pub/sub messaging).
- As an event-driven technology, WebSocket allows data to be transferred without the client requesting it. This characteristic is desirable in scenarios where the client needs to react quickly to an event (especially ones it cannot predict, such as a fraud alert).

WebSocket disadvantages

- WebSocket is stateful. This can be tricky to handle, especially at scale, because it requires the server layer to keep track of each individual WebSocket connection and maintain state information.
- WebSockets don’t automatically recover when connections are terminated – this is something you need to implement yourself, and is part of the reason why there are many WebSocket client-side libraries in existence.
- Certain environments (such as corporate networks with proxy servers) will block WebSocket connections.

WebRTC vs WebSockets: What are the key differences?

- WebSocket provides a *client-server* computer communication protocol, whereas WebRTC offers a *peer-to-peer protocol and communication capabilities* for browsers and mobile apps.

- While WebSocket works only over TCP, WebRTC is primarily used over UDP (although it can work over TCP as well).
- WebSocket is a better choice when data integrity is crucial, as you benefit from the underlying reliability of TCP. On the other hand, if speed is more important and losing some packets is acceptable, WebRTC over UDP is a better choice.
- WebRTC is primarily designed for streaming audio and video content. It is possible to stream media with WebSockets too, but the WebSocket technology is better suited for transmitting text/string data using formats such as JSON.

When to use WebRTC?

WebRTC is a good choice for the following use cases:

- Audio and video communications, such as video calls, video chat, video conferencing, and browser-based VoIP.
- File sharing apps.
- Screen sharing apps.
- Broadcasting live events (such as sports events).
- IoT devices (e.g., drones or baby monitors streaming live audio and video data).

When to use WebSockets?

We can broadly group Web Sockets use cases into two distinct categories:

- Realtime updates, where the communication is unidirectional, and the server is streaming low-latency (and often frequent) updates to the client. Think of live score updates or alerts and notifications, to name just a few use cases.
- Bidirectional communication, where both the client and the server send and receive messages. Examples include chat, virtual events, and virtual classrooms (the last two usually involve features like live polls, quizzes, and Q&As). WebSockets can also be used to underpin multi-user synchronized collaboration functionality, such as multiple people editing the same document simultaneously.

WebRTC, WebSocket, and Ably

Ably is a serverless WebSocket platform optimized for high-scale data distribution. We make it easy for developers to build live experiences such as chat, live dashboards, alerts and notifications, asset tracking, and collaborative apps, without having to worry about managing and scaling infrastructure. Additionally, you can use our WebSocket APIs to quickly implement dependable signaling mechanisms for your WebRTC apps.

Ably offers:

- Roust and diverse features, including pub/sub messaging, automatic reconnections with continuity, and presence.
- Dependable guarantees: <65 ms round trip latency for 99th percentile, guaranteed ordering and delivery, global fault tolerance, and a 99.999% uptime SLA.
- An elastically-scalable, globally-distributed edge network capable of streaming billions of messages to millions of concurrently-connected devices.
- 25+ client SDKs targeting every major programming language.

WebRTC vs WebSocket FAQs

Is WebRTC faster than WebSockets?

WebRTC and WebSockets are both event-driven technologies that provide sub-second latencies, which makes them suitable for realtime use cases. WebRTC primarily works over UDP, while WebSocket is over TCP. This means that WebRTC offers slightly lower latency than WebSockets, as UDP is faster than TCP. However, the difference is negligible; plus, TCP is more reliable when it comes to packet delivery (in comparison, with UDP some packets may be lost).

Is WebRTC built on WebSockets?

No, WebRTC is not built on WebSockets. WebRTC and WebSockets are distinct technologies. WebSocket provides a client-server computer communication protocol that works on top of TCP, whereas WebRTC offers a peer-to-peer protocol that's primarily used over UDP (although you can use WebRTC over TCP too).

WebRTC vs WebSocket performance: which one is better?

It's not possible to determine a winner, as many factors influence the performance of WebRTC and WebSockets, such as the hardware used, and the number of concurrent users. It might even be a pointless comparison, considering that WebRTC use cases are different from WebSocket use cases.

WebRTC can be extremely CPU-intensive, especially when dealing with video content and large groups of users. This makes it costly and hard to reliably use and scale WebRTC applications. Similarly, there are many challenges in building a WebSocket solution that you can trust to perform at scale.

Web Socket represents a major upgrade in the history of web communications. Before its existence, all communication between the web clients and the servers relied only on HTTP.

Web Socket helps in dynamic flow of the connections that are persistent full duplex. Full duplex refers to the communication from both the ends with considerable fast speed.

It is termed as a game changer because of its efficiency of overcoming all the drawbacks of existing protocols.

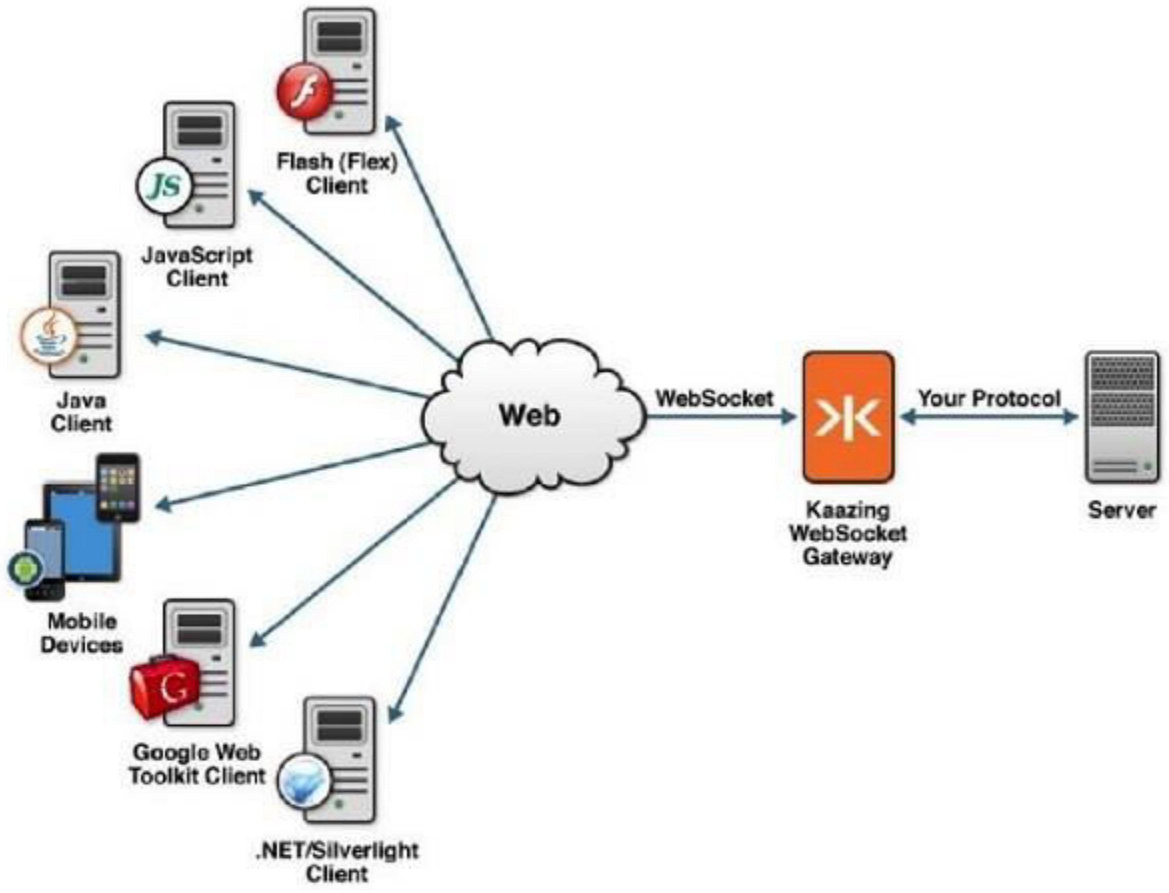
Web Socket for Developers and Architects

Importance of Web Socket for developers and architects −

- Web Socket is an independent TCP-based protocol, but it is designed to support any other protocol that would traditionally run only on top of a pure TCP connection.
- Web Socket is a transport layer on top of which any other protocol can run. The Web Socket API supports the ability to define sub-protocols: protocol libraries that can interpret specific protocols.
- Examples of such protocols include XMPP, STOMP, and AMQP. The developers no longer have to think in terms of the HTTP request-response paradigm.

- The only requirement on the browser-side is to run a JavaScript library that can interpret the Web Socket handshake, establish and maintain a Web Socket connection.
- On the server side, the industry standard is to use existing protocol libraries that run on top of TCP and leverage a Web Socket Gateway.

The following diagram describes the functionalities of Web Sockets −



Web Socket connections are initiated via HTTP; HTTP servers typically interpret Web Socket handshakes as an Upgrade request.

Web Sockets can both be a complementary add-on to an existing HTTP environment and can provide the required infrastructure to add web functionality. It relies on more advanced, full duplex protocols that allow data to flow in both directions between client and server.

Functions of Web Sockets

Web Sockets provide a connection between the web server and a client such that both the parties can start sending the data.

The steps for establishing the connection of Web Socket are as follows −

- The client establishes a connection through a process known as Web Socket handshake.

- The process begins with the client sending a regular HTTP request to the server.
- An Upgrade header is requested. In this request, it informs the server that request is for Web Socket connection.
- Web Socket URLs use the **ws** scheme. They are also used for secure Web Socket connections, which are the equivalent to HTTPs.

A simple example of initial request headers is as follows −

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
```

WebRTC API

WebRTC (Web Real-Time Communication) is a technology that enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary. The set of standards that comprise WebRTC makes it possible to share data and perform teleconferencing peer-to-peer, without requiring that the user install plug-ins or any other third-party software.

WebRTC consists of several interrelated APIs and protocols which work together to achieve this. The documentation you'll find here will help you understand the fundamentals of WebRTC, how to set up and use both data and media connections, and more.

WebRTC concepts and usage

WebRTC serves multiple purposes; together with the Media Capture and Streams API, they provide powerful multimedia capabilities to the Web, including support for audio and video conferencing, file exchange, screen sharing, identity management, and interfacing with legacy telephone systems including support for sending DTMF (touch-tone dialing) signals. Connections between peers can be made without requiring any special drivers or plug-ins, and can often be made without any intermediary servers.

Connections between two peers are represented by the `RTCPeerConnection` interface. Once a connection has been established and opened using `RTCPeerConnection`, media streams (`MediaStream`s) and/or data channels (`RTCDataChannel`s) can be added to the connection.

Media streams can consist of any number of tracks of media information; tracks, which are represented by objects based on the `MediaStreamTrack` interface, may contain one of a number of types of media data, including audio, video, and text (such as subtitles or even chapter names). Most streams consist of at least one audio track and likely also a video track, and can be used to send and receive both live media or stored media information (such as a streamed movie).

You can also use the connection between two peers to exchange arbitrary binary data using the RTCDataChannel interface. This can be used for back-channel information, metadata exchange, game status packets, file transfers, or even as a primary channel for data transfer.

Interoperability

WebRTC is in general well supported in modern browsers, but some incompatibilities remain. The adapter.js library is a shim to insulate apps from these incompatibilities.

WebRTC reference

Because WebRTC provides interfaces that work together to accomplish a variety of tasks, we have divided up the reference by category. Please see the sidebar for an alphabetical list.

Connection setup and management

These interfaces, dictionaries, and types are used to set up, open, and manage WebRTC connections. Included are interfaces representing peer media connections, data channels, and interfaces used when exchanging information on the capabilities of each peer in order to select the best possible configuration for a two-way media connection.

Interfaces

RTCPeerConnection

> Represents a WebRTC connection between the local computer and a remote peer. It is used to handle efficient streaming of data between the two peers.

RTCDataChannel

> Represents a bi-directional data channel between two peers of a connection.

RTCDataChannelEvent

> Represents events that occur while attaching a RTCDataChannel to a RTCPeerConnection. The only event sent with this interface is datachannel.

RTCSessionDescription

> Represents the parameters of a session. Each RTCSessionDescription consists of a description type indicating which part of the offer/answer negotiation process it describes and of the SDP descriptor of the session.

RTCStatsReport

Provides information detailing statistics for a connection or for an individual track on the connection; the report can be obtained by calling RTCPeerConnection.getStats().

RTCIceCandidate

Represents a candidate Interactive Connectivity Establishment (ICE) server for establishing an RTCPeerConnection.

RTCIceTransport

Represents information about an ICE transport.

RTCPeerConnectionIceEvent

Represents events that occur in relation to ICE candidates with the target, usually an RTCPeerConnection. Only one event is of this type: icecandidate.

RTCRtpSender

Manages the encoding and transmission of data for a MediaStreamTrack on an RTCPeerConnection.

RTCRtpReceiver

Manages the reception and decoding of data for a MediaStreamTrack on an RTCPeerConnection.

RTCTrackEvent

The interface used to represent a track event, which indicates that an RTCRtpReceiver object was added to the RTCPeerConnection object, indicating that a new incoming MediaStreamTrack was created and added to the `RTCPeerConnection`.

RTCSctpTransport

Provides information which describes a Stream Control Transmission Protocol (**SCTP**) transport and also provides a way to access the underlying Datagram Transport Layer Security (**DTLS**) transport over which SCTP packets for all of an RTCPeerConnection's data channels are sent and received.

Dictionaries

RTCIceServer

Defines how to connect to a single ICE server (such as a STUN or TURN server).

RTCRtpContributingSource

Contains information about a given contributing source (CSRC) including the most recent time a packet that the source contributed was played out.

Events

bufferedamountlow

The amount of data currently buffered by the data channel—as indicated by its bufferedAmount property—has decreased to be at or below the channel's minimum buffered data size, as specified by bufferedAmountLowThreshold.

close

The data channel has completed the closing process and is now in the closed state. Its underlying data transport is completely closed at this point. You can be notified *before* closing completes by watching for the closing event instead.

closing

The RTCDataChannel has transitioned to the closing state, indicating that it will be closed soon. You can detect the completion of the closing process by watching for the close event.

connectionstatechange

The connection's state, which can be accessed in connectionState, has changed.

datachannel

A new RTCDataChannel is available following the remote peer opening a new data channel. This event's type is RTCDataChannelEvent.

error

An RTCErrorEvent indicating that an error occurred on the data channel.

error

An RTCErrorEvent indicating that an error occurred on the RTCDtlsTransport. This error will be either dtls-failure or fingerprint-failure.

gatheringstatechange

The RTCIceTransport's gathering state has changed.

icecandidate

An RTCPeerConnectionIceEvent which is sent whenever the local device has identified a new ICE candidate which needs to be added to the local peer by calling setLocalDescription().

icecandidateerror

An RTCPeerConnectionIceErrorEvent indicating that an error has occurred while gathering ICE candidates.

iceconnectionstatechange

Sent to an RTCPeerConnection when its ICE connection's state—found in the iceconnectionstate property—changes.

icegatheringstatechange

Sent to an RTCPeerConnection when its ICE gathering state—found in the icegatheringstate property—changes.

message

A message has been received on the data channel. The event is of type MessageEvent.

negotiationneeded

Informs the `RTCPeerConnection` that it needs to perform session negotiation by calling createOffer() followed by setLocalDescription().

open

The underlying data transport for the `RTCDataChannel` has been successfully opened or re-opened.

selectedcandidatepairchange

The currently-selected pair of ICE candidates has changed for the `RTCIceTransport` on which the event is fired.

track

The `track` event, of type RTCTrackevent is sent to an RTCPeerConnection when a new track is added to the connection following the successful negotiation of the media's streaming.

signalingstatechange

Sent to the peer connection when its signalingstate has changed. This happens as a result of a call to either setLocalDescription() or setRemoteDescription().

statechange

The state of the `RTCDtlsTransport` has changed.

statechange

The state of the `RTCIceTransport` has changed.

statechange

The state of the `RTCSctpTransport` has changed.

Types

RTCSctpTransport.state

Indicates the state of an RTCSctpTransport instance.

Identity and security

These APIs are used to manage user identity and security, in order to authenticate the user for a connection.

RTCIdentityProvider

Enables a user agent is able to request that an identity assertion be generated or validated.

RTCIdentityAssertion

Represents the identity of the remote peer of the current connection. If no peer has yet been set and verified this interface returns `null`. Once set it can't be changed.

RTCIdentityProviderRegistrar

Registers an identity provider (idP).

RTCCertificate

Represents a certificate that an RTCPeerConnection uses to authenticate.

Telephony

These interfaces and events are related to interactivity with Public-Switched Telephone Networks (PTSNs). They're primarily used to send tone dialing sounds—or packets representing those tones—across the network to the remote peer.

Interfaces

RTCDTMFSender

Manages the encoding and transmission of Dual-Tone Multi-Frequency (DTMF) signaling for an RTCPeerConnection.

RTCDTMFToneChangeEvent

Used by the tonechange event to indicate that a DTMF tone has either begun or ended. This event does not bubble (except where otherwise stated) and is not cancelable (except where otherwise stated).

Events

tonechange

Either a new DTMF tone has begun to play over the connection, or the last tone in the `RTCDTMFSender`'s toneBuffer has been sent and the buffer is now empty. The event's type is RTCDTMFToneChangeEvent.

REF :

https://www.vonage.com/resources/articles/real-time-communications/

What is web RTC **https://bloggeek.me/what-is-webrtc/**

https://ably.com/topic/webrtc-vs-websocket

https://gist.github.com/1322534/c1c0b108a6c6b53eadb1b370ae82ed891a19c6a8

https://github.com/prashantsihag03/simple-WebRTC-Implementation

https://www.tutorialspoint.com/websockets/websockets_functionalities.htm

https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API