SER 421 ADVANCED PROJECT
WebSockets

Aditya Desai
Shengdong Chen
Maaz Ahmed

Table of Contents

1. Learning Outcomes

After successfully completing tutorial, the student will:

1. Come away with a basic idea of the history of WebSocket
2. Understand the basic concepts of WebSocket
3. Be able to implement a web application using WebSocket

2. History

Creating web applications that require bidirectional communication between a client and a server (for example, instant messaging and gaming apps) has traditionally necessitated abusing HTTP to poll the server for updates while sending upstream alerts as separate HTTP requests [RFC6202]. As a result, various issues arise:

1. For each client, the server is compelled to use several underlying TCP connections: one for sending data to the client and another for every receiving message.
2. Because each client-to-server message requires an HTTP header, the wire protocol includes a large overhead.
3. To track responses, the client-side script is compelled to maintain a mapping from outgoing connections to incoming connections.

Employing a single TCP connection for both ways would be a simpler and easier option. The WebSocket Protocol makes this possible. It provides an alternative to HTTP polling for two-way communication from a web page to a remote server when used in conjunction with the WebSocket API.

Games, market tickers, multi-user applications with simultaneous editing, user interfaces exposing server-side services in real time, and other web applications can all benefit from this method.

RFC 6455 — a proposed standard – is the most recent specification of the Web Socket protocol. Internet Explorer, Mozilla Firefox, Google Chrome, Safari, and Opera are among the browsers that implement RFC 6455.

The WebSocket Protocol is intended to replace existing bidirectional communication technologies that rely on HTTP as a transport layer in order to take advantage of existing infrastructure (proxies, filtering, and authentication). Because HTTP was not initially designed to be used for bidirectional communication [RFC6202], such technologies were implemented as a trade-off between efficiency and reliability.

The WebSocket Protocol tries to address the goals of existing bidirectional HTTP technologies within the context of the present and existing HTTP infrastructure; as a result, it's designed to work over HTTP ports 80 and 443 as well as support HTTP proxies and intermediaries, even if this adds some complexity to the current environment. WebSocket is not limited to HTTP by design, and future implementations could employ a simpler handshake via a dedicated port instead of recreating the protocol entirely. This latter aspect is critical because interactive messaging traffic patterns differ significantly from ordinary HTTP traffic, resulting in unexpected demands on several components.

2.1 HTTP Polling

A client sends regular queries to the server using the classic or "short polling" approach, with each request attempting to "pull" any accessible events or data. The server delivers an empty answer if no events or data are available, and thus the client needs to wait for a while before making another poll request. The polling frequency is determined by the client's tolerance for delay when getting updated information from the server. This approach has the disadvantage of relying heavily on acceptable delay in the transmission of updates from server to client to use resources (server processing and network). If the tolerable latency is not much (on the order of seconds), the polling frequency may place an undue load on the server, network, or both.

In contrast to "short polling," "long polling" aims to cut back both server-client message delivery delay and processing/network resource consumption. The server saves time by responding to requests only if a particular event, condition, or timeout occurs. The client normally sends a fresh long poll request after the server provides a long poll answer. This effectively implies that the server will keep a lengthy poll request open at all times, responding when fresh data is available for the client. As a result, the server can "start" communication in an asynchronous manner.

The steps in polling process are as follows:

1. After making the initial request, the client waits for a response.
2. The server waits for an update or until a certain condition or timeout has occurred before responding.
3. The server delivers a full response when an update is available for delivery to the customer. The client sends a new long poll request, either immediately or after a delay to allow for an appropriate latency duration after getting the response

Both persistent and non-permanent HTTP connections can use the HTTP long polling method. The usage of permanent HTTP connections eliminates the extra expense of creating a new TCP/IP connection for each long poll request.

In short, long polling is a technique where the server elects to hold a client's connection open for as long as possible (usually up to 20 seconds), delivering a response only after either the data becomes available or a timeout threshold is reached.

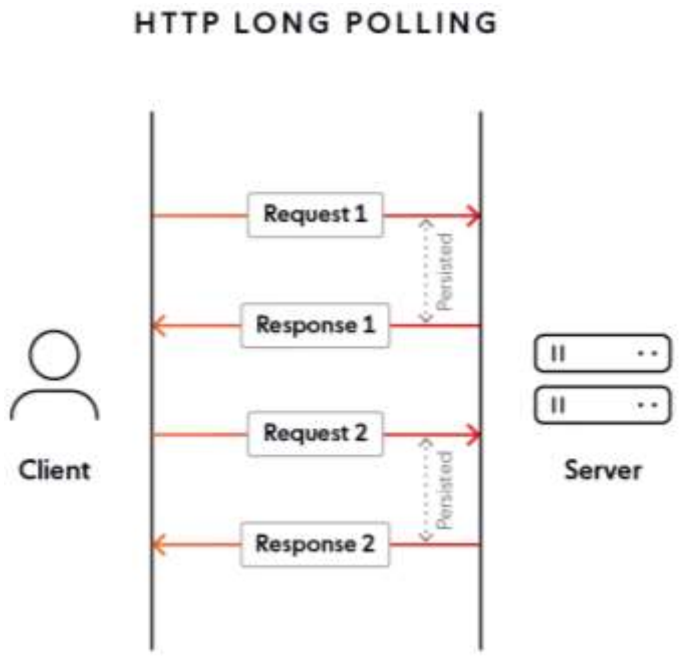

Fig 1: The following is the fundamental life cycle of an application that uses HTTP long polling:

3. Learning Activities

3.1 WebSocket

WebSocket is a protocol providing two-way communication between servers and clients, meaning both sides communicate and exchange data at the same time. From the ground up, this protocol defines a full duplex communication between a client and a server. WebSocket helps in bringing a desktop-like feel and functionality to web browsers. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and hence allowing for messages to be passed back and forth while keeping the connection open. The WebSocket Protocol is an independent TCP-based protocol. It's only relationship to HTTP is that its initial handshake is interpreted by HTTP servers as an Upgrade request. The communications are done over TCP port number 80. It represents an advance in client/server web technology that has been long overdue.

The requests from client and server look as follows:

Client request:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Server response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

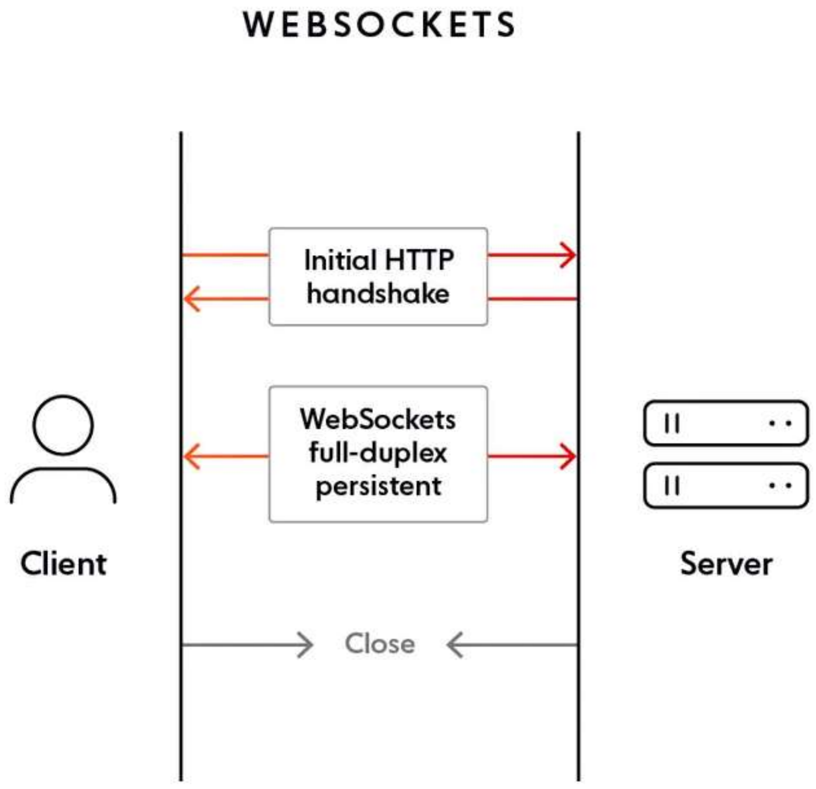

Fig 2. Visual representation of WebSocket protocol

3.1.1 Why Use WebSocket over HTTP?

WebSocket solves a few issues with HTTP:

- In HTTP, the request is always initiated by the client and the response is processed by the server – making HTTP a uni-directional protocol. However, Websocket protocol is a **Bi-Directional Protocol** where either client or server can initiate and send a message to the other party
- **Full-duplex communication** – client and server can exchange messages to each other independently at the same time.
- After upgrading the HTTP connection in the initiation during handshake, client and server communicate over the **Single TCP Connection (Persistent Connection)** throughout the lifecycle of the WebSocket connection.

3.1.2 WebSockets as a Standard

Web Socket's main advantage is that it allows two-way communication (full duplex) via a single TCP connection.

1. As Web socket protocol is becoming standardized, which implies that it may be used to communicate in real time between web servers and clients.
2. Web sockets are becoming a cross-platform standard for real-time client-server communication.
3. This standard opens the door to new types of applications. Businesses such as multiplayer gaming, real time sports updates, collaborative editing, encrypted message exchanges and similar real-time web applications benefit from this technology.

3.1.3 WebSocket Pros

- WebSocket is an event-driven protocol, which means you can actually use it for truly real time communication. Unlike HTTP, where you have to constantly request updates, with websocket, updates are sent immediately when they are available.
- WebSocket keeps a single, persistent connection open while eliminating latency problems that arise with HTTP request/response-based methods.
- WebSockets generally do not use XMLHttpRequest, and as such, headers are not sent every-time we need to get more information from the server. This, in turn, reduces the expensive data loads being sent to the server.

3.1.4 WebSocket Cons

- 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 client-side libraries in existence.
- Browsers older than 2011 aren't able to support WebSocket connections – but this is increasingly less relevant.

3.2 WebSockets Interfaces and API

3.2.1 Description

The WebSocket API is a cutting-edge technology that allows a two-way interactive communication session to be established between a user's browser and a server. You may use this API to send messages to a server and obtain event-driven answers instead of polling the service.

3.2.2 Interfaces

There are 3 interfaces available for the WebSocket API:

1. WebSocket: This is the **main** interface for connecting to a WebSocket server and then transmitting and receiving data via the connection.
2. CloseEvent: When the connection is **closed**, the WebSocket object sends this event.
3. MessageEvent: When a **message** is received from the server, the WebSocket object fires this event.

3.2.3 Methods

`1.WebSocket(url[, protocols])`

This is the constructor which returns a newly created WebSocket object.

Example:

```
var aWebSocket = new WebSocket(url [, protocols]);
```

2. **`WebSocket.close([code[, reason]]`**

This method is called to close the connection.

Example:

```
WebSocket.close();
```

3. **`WebSocket.send(data)`**

This method enqueues data to be transmitted.

Example:

```
WebSocket.send("Hello server!");
```

3.2.4 Properties

`WebSocket.binaryType`	The binary data type used by the connection.
`WebSocket.bufferedAmount`	(Read only) The number of bytes of queued data.
`WebSocket.extensions`	(Read only) The extensions selected by the server.
`WebSocket.onclose`	An event listener to be called when the connection is closed.
`WebSocket.onerror`	An event listener to be called when an error occurs.
`WebSocket.onmessage`	An event listener to be called when a message is received from the server.
`WebSocket.onopen`	An event listener to be called when the connection is opened.
`WebSocket.protocol`	(Read only) The sub-protocol selected by the server.
`WebSocket.readyState`	(Read only) The current state of the connection.
`WebSocket.url`	(Read only) The absolute URL of the WebSocket.

(cited from https://developer.mozilla.org/en-US/docs/Web/API/WebSocket)

3.2.5 Events

There are four main Web Socket API events: Open, Message, Close, Error. Each event is handled by implementing functions such as onopen, onmessage, onclose, and onerror, as appropriate.

1.Open

To create a websocket, we have to pass the url of the server that it is communicating with.
A Web Socket can be opened with the following code:

```
// Create a new WebSocket for SER 421.
var socket = new WebSocket('ws://echo.websocket.org');
//The url to be put will be of the server for communication
```

The **open** event is fired by the Web Socket instance once the client and server have established a connection. The first communication between the client and the server is known as the handshake. The **onopen** event is triggered once the connection has been established.

Example:

```
connection.onopen = function(event) {
   console.log(“WebSocket established”);
   // Display user friendly messages for the successful establishment
of connection
   var.label = document.getElementById(“status”);
   label.innerHTML = ”Connection established”;
}
```

2.Message

When the server **sends** data, the message event occurs. Plain text messages, binary data, and pictures are examples of communications transmitted from the server to the client. The **onmessage** function is called whenever data is transmitted.

Example:

```
socket.onmessage = function(e){
   var server_message = e.data;
   console.log(server_message);
}
```

3.Close

The communication between the server and the client comes to a stop with the close event. With the aid of the **onclose** event, you may close the connection. No messages can be sent between the server and the client once the **onclose** event has marked the end of communication. Poor connectivity might also cause the event to be closed.

Example:

```
socket.onclose = function(){
   socket = null;
   console.log("Close occurred.");
}
```

4.Error

Error marks are used to indicate a mistake that occurs during conversation. The **onerror** event is used to identify it. The connection is always terminated after an onerror.

```
socket.onerror = function(event) {
   console.log("Error occurred.");
   // Inform the user about the error.
   var label = document.getElementById("status-label");
   label.innerHTML = "Error: " + event;
}
```

3.2.6 CloseEvents

Methods:
`CloseEvent()`
This is the constructor which returns the newly created CloseEvent.
`CloseEvent.initCloseEvent()`
This method initializes the value of a CloseEvent created.
Although this method is legal, CloseEvent() can substitute it.
Properties:

`CloseEvent.code`	(Read Only)Returns an unsigned short containing the close code sent by the server.
`CloseEvent.reason`	(Read Only)Returns a DOMString indicating the reason the server closed the connection. This is specific to the particular server and sub-protocol.
`CloseEvent.wasClean`	(Read Only)Returns a boolean value that Indicates whether or not the connection was cleanly closed.

(cited from https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent)

3.2.7 MessageEvents

Methods:
`MessageEvent()`
This is the Constructor which creates a new MessageEvent.
Property:

`MessageEvent.data`	(Read only)The data sent by the message emitter.
`MessageEvent.origin`	(Read only)A USVString representing the origin of the message emitter.
`MessageEvent.lastEventId`	(Read only)A DOMString representing a unique ID for the event.
`MessageEvent.source`	(Read only)A MessageEventSource (which can be a WindowProxy, MessagePort, or ServiceWorker object) representing the message emitter.
`MessageEvent.ports`	(Read only)An array of MessagePort objects representing the ports associated with the channel the message is being sent through (where appropriate, e.g. in channel messaging or when sending a message to a shared worker).

(cited from https://developer.mozilla.org/en-US/docs/Web/API/MessageEvent)

3.3 Implementation

Kindly refer to our two applications made using WebSocket API and NPM package – ws on Node.js

4 Analysis and Summary

In this report, we began by understanding the history of the various protocols and the various developments that were made to result in WebSocket. Next, we went through some technical concepts of the WebSocket protocol and understood how the protocol works. We also compare how WebSocket is different from HTTP-based technologies. Furthermore, we learn more about the WebSocket API and the various interfaces provided by the API.
After working with WebSocket, we find that whenever you need a better low-latency connection between a client and server, WebSocket is your best option. However, WebSocket can be frustrating to integrate into your existing web infrastructure as it requires a change in architecture as well as increasing security around the open sockets.
The framework shows better output than traditional web protocols such as HTTP during real time message exchanges. However, when a long-lived connection or frequent messages exchanges are not required between server and client, HTTP is better suited for communication.

5 Reference

https://datatracker.ietf.org/doc/html/rfc6455#page-4
https://www.tutorialspoint.com/websockets/index.htm
https://ably.com/blog/websockets-vs-long-polling
https://datatracker.ietf.org/doc/html/rfc6202#page-5
https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

6 Additional Training

In our implementation we have used Nodejs as a server and browser WebSocket API for client communication. Explore other frameworks such as Flask, PHP to implement similar applications. Also explore different mechanisms to achieve web sockets such socket.io and SignalR
Try developer more complex multiplayer games with exchanging simple real-time message between players.

Have Fun!