Comparison of HTTP Server Sent Events and WebSockets for Bidirectional Communication on the Web

Patrick Richeal

Abstract

The HTTP protocol was initially designed so a client (generally a web browser) can make requests to a server for data, whether that be a single webpage or additional data to dynamically use within an already loaded webpage [1]. Many modern and more complicated web applications can benefit heavily from the direct transfer of data from server to client (e.g. for multiplayer games or chat applications [2]), but the request-response nature of the HTTP protocol was not built for this. To remedy this issue, *server sent events* and *WebSockets* (two separate technologies) were created. Server sent events were created as a part of the HTML5 specification and enables the server to send data directly to the client over an HTTP connection, allowing for bidirectional communication over the HTTP protocol between the client and server [7]. WebSockets provide a new protocol entirely that allows for direct bidirectional communication between the client and server over a single TCP connection [3]. This paper aims to develop a method for comparing the performance, efficiency, and security between these two methods along with analysis of the execution of the developed method.

1. Introduction

Server sent events and WebSockets both allow for data to be sent directly from the server to the client (without request from the client). WebSockets specifically also allow for the client to send data to the server, in the same manner as the server does to the client (over the same TCP connection). This means that WebSockets allow for bidirectional communication on their own without the need for any other methods of data transfer. Server sent events, on the other hand, only allow for data to be sent directly from the server to the client [6]. If data needs to be sent from the client to the server, a standard HTTP request must be used. Server sent events only provide bidirectional communication when used in conjunction with standard HTTP requests, then, as opposed to WebSockets that provide bidirectional communication out of the box. This is the basis for the comparison that will take place between these two methods.

1.1 Server sent events

Server sent events allow for server initiated data transfer to a client over an HTTP connection. The HTTP connection is initiated by the client, and then the server holds the connection open indefinitely so it is available to stream data down to the client.

Server sent events were standardized as a part of the HTML5 specification and are supported by all major web browsers [4]. The web browser itself is responsible for handling the received event, and makes it available to listen to via the JavaScript *EventSource* object. Here is an example of how to open a connection to a specific server and listen for messages.

```
let server = new EventSource('server.php');
server.addEventListener('message', (e) => {
    console.log(e.data);
});
```

It is then up to the server to properly send data in the correct format via this HTTP

connection. The HTTP response from the server uses the *text/event-stream* content type header along with the *keep-alive* connection header to signify to the browser that the data returned is a server sent event and that the connection should be held open, even after the response is received [5]. Plaintext data is added to the response in the format seen in the following example server implementation.

```
function handler(response) {
    response.writeHead(200, {
        'Content-Type': 'text/event-stream',
        'Cache-Control': 'no-cache',
        'Connection': 'keep-alive'
    });
    response.write('id:
UniqueIdentifier\n');
    response.write('data: TheData\n\n');
}
```

The two subsequent new line characters in the last *response.write* function call indicate that the data is complete.

1.2 WebSockets

WebSockets allow for data transfer from server to client and from client to server via a single TCP connection. This TCP connection is established through a handshake process that involves an initial HTTP request with an *upgrade* header, indicating to the server that the client would like to establish a WebSocket connection.

The WebSocket protocol was standardized in 2011 by the IETF as RFC 6455 [3]. WebSockets, like server sent events, are supported by all major web browsers [10].

WebSockets can be utilized on the client in a very similar manner to server sent events, as seen by the below example. The below example also shows how the client can send data to the server.

```
let socket = new
WebSocket(ws://example.com);
socket.addEventListener('message', (e) => {
    console.log(e.data);
});
socket.addEventlistener('open', () => {
    socket.send('Message to server');
});
```

The server implementation for WebSockets is more complicated than for server sent events, which simply need to respond with a standard plaintext HTTP response. The general idea is that the server needs to be able to send and receive data in a specific format specified by the WebSocket specification on a TCP connection (in addition to handling the WebSocket handshake). The reality of writing a WebSocket server is using one of the many WebSocket libraries for the server's programming language, with which you would get a syntax closer to client's with a listener and send function.

2. Methodology

Different methods will be used to compare the performance, efficiency, and security between server sent events and WebSockets. For the sake of these comparisons, performance will more specifically refer to the speed at which data can be sent from client to server and server to client. Efficiency refers to the amount of data actually being transferred when sending the same base data from client to server and server to client. Both of these metrics, performance and efficiency, can be compared by analyzing a simulation of sent WebSockets, HTTP requests (client to server for server sent event testing), and server sent events (server to client for server sent event testing). The below sections for performance and efficiency will outline a framework that involves building a small simulation that sends data bidirectionally (from client to server and server to client). That simulation will be used to analyze and compare the performance and efficiency between server sent events and WebSockets.

Comparing the security between server sent events and WebSockets is not as trivial as comparing performance and efficiency. There is no single metric to measure for security (such as time for performance and amount of data for efficiency), so the comparison of security between server sent events and WebSockets will just be reviewed in the security section of results (section 3.3).

2.1 Performance

To compare the performance between server sent events and WebSockets, a simulation needs to be set up that can send data from both client to server and server to client. This setup needs to be able to send data via both the combination of HTTP post requests (from client to server) and server sent events (from server to client) to measure the speed of bidirectional communication using server sent events and with WebSockets to measure the speed of bidirectional communication using WebSockets.

To build this simulation, two clients and two servers need to be created. One client and server will communicate exclusively with each other via HTTP post requests (client to server) and server sent events (server to client) to perform bidirectional communication over HTTP with server sent events. The other client and server will communicate exclusively with each other via WebSockets (client to server and server to client) to perform bidirectional communication with WebSockets. This will provide the base structure to set up a simulation of data being sent back and forth that we can measure for speed of transfer.

To accurately measure the speed (performance) at which the client and servers can send data to and from each other, we need to perform enough individual transmissions of data that any significant outliers from fluctuations in network reliability don't significantly impact the average of the transmission speeds. To accomplish this, each client and server should perform 100 transmissions each to the corresponding other entity. This means that for both sets of client and server, the client should send 100 transmissions to the server and the server should send 100 transmissions to the client.

To actually measure the speed at which the data was sent, each transmission should contain a message with the current unix time in milliseconds (the time at which the message was created or sent). This will allow the entity receiving the message to compare the unix time in the message to the current time to determine the total time the message took to send.

Each client and server should be set up to have the simulation initiated by the client. When the client is started, it should start sequentially sending its 100 transmissions to the server (with enough time in between transmissions to allow for the transmission to be successfully received and processed). The server should be receiving and processing these transmissions and storing the total time each transmission took to be received. Once the server has received 100 transmissions, it should output the list of 100 transmission times (these will be the client to server transmission times). The server should then start sending its own 100 transmissions to the client in the same manner as the client did to the server. The client should receive and process these transmissions in the same manner that the server did, eventually outputting the list of 100 transmission times (these will be the server to client transmission times).

Each pair of client and server (one for server sent events bidirectional communication and one for WebHooks bidirectional communication) will then have the ability to simulate a group of transmissions from both client to server and server to client, and

output the list of transmission times for each transmission. The average of the transmission times for both server sent events and WebHooks can be compared for both client to server and server to client transmissions to compare the performance of the two methods of bidirectional communication.

A potential problem that may arise if attempting to transmit data from a client and server on the same local network (common in a test scenario such as this) is that the transmissions times may be so small that it is not useful to even compare them (such as if every transmission only takes 1 millisecond). If this is the case, the server code should be moved to a server that is physically farther away to simulate a more realistic transmission time.

2.2 Efficiency

To compare the efficiency, or amount of data transfer needed to send the same message, we can utilize the WireShark desktop application to analyze the packets being sent back and forth between each client and server.

WireShark can be used to record all network packets that are being sent to and from your computer. By recording with WireShark while the above performance simulations (outlined in section 2.1) are running, we can see the packets that are sent and received for each type of transmission (either server sent event or WebHook, both from server to client or client to server).

For a single transmission of data, multiple packets will be sent both to and from the computer running the simulation. It will be important to determine which group of packets are for the single transmission of data, so the amount of data required to send the same message can be determined for that type of transmission. This could prove challenging on a computer that has other active network activity, but the results can be narrowed down significantly by only looking at packets to and from the IP address of the server running the server side of the simulation. Additionally, if you know that the client is sending a transmission every 500 milliseconds (for example), and there is a distinct group of near identical packets being sent and received at that same interval, you have essentially isolated the group of packets needed to send that single transmission of data. This group of packets should be identified for each type of transmission, and the total number of bytes for these groups of packets can be calculated to compare the needed data to transfer the same message for each transmission type.

2.3 Security

As explained in the introduction to the methodology section, security can not be analyzed and measured like performance and efficiency can. There is no methodology or framework that needs to be set up to compare the security between server sent events and WebHooks, so the security of each method will simply be reviewed in the security section of results (section 3.3).

3. Results

I used web technologies to implement the simulation needed to get the data to compare performance and efficiency between server sent events and WebHooks. Each client was built as a simple HTML web page that uses its respective technology (HTTP post requests for server sent events and WebSockets for WebSockets) to start sending its 100 transmissions to its corresponding server. Both servers were written in JavaScript with NodeJS [12] and use publicly available packages to easily listen and send the correct transmissions

(server sent events and WebSockets). The code for the clients and servers along with the raw data that was measured is available in a public GitHub repository (see reference) [13].

3.1 Performance

Running the clients and servers on the same local network did have the potential problem as described in the performance section of methodology (section 2.1), the transmission times were too small to compare. To solve this, I rented a virtual server from DigitalOcean (a cloud computing company) and ran the server code on a machine in New York. This resulted in transmission times that were more realistic to compare.

The below table shows the average transmission times from client to server and server to client for both server sent events and WebHooks. Note that the client to server time should only be compared to the other client to server time and the server to client time should only be compared to the other server to client time. This is because there are inherent differences in the way the data is being sent from the client (my computer) and from the server (a virtual server somewhere in New York) that are not accounted for in this comparison.

	Server sent events	WebHooks
Client to server	25.33 ms	22.56 ms
Server to client	70.86 ms	84.63 ms

The transmission times are comparable within each category (client to server and server to client), with WebHooks offering a slightly faster client to server transmission time and server sent events offering a fairly faster server to client transmission time.

3.2 Efficiency

Using WireShark as laid out in the efficiency section of methodology (section 2.2) proved to be successful in determining the total amount of data transferred for each transmission type. The following is an outline of the packets sent (and their sizes) for each single transmission of data (from both client to server and server to client for server sent events and WebHooks).

Server sent events - client to server

- HTTP post packet (data) client to server: 463 bytes
- SSH packet server to client: 174 bytes
- TCP packet (ack) client to server: 66 bytes
- SSH packet server to client: 102 bytes
- TCP packet (ack) client to server: 66 bytes
- HTTP response packet server to client: 304 bytes
- TCP packet (ack) client to server: 66 bytes
- Total: 1,241 bytes

Server sent events - server to client

- SSH packet server to client: 150 bytes
- TCP packet (ack) client to server: 66 bytes
- SSH packet server to client: 102 bytes
- TCP packet (ack) client to server: 66 bytes
- TCP packet server to client: 101 bytes
- TCP packet (ack) client to server: 66 bytes
- Total: 551 bytes

WebHooks - client to server

- WebSocket packet (data) client to server: 93 bytes
- TCP packet (ack) server to client: 66 bytes
- SSH packet server to client: 158 bytes
- TCP packet (ack) client to server: 66 bytes
- SSH packet server to client: 102 bytes
- TCP packet (ack) client to server: 66 bytes
- Total: 551 bytes

WebHooks - server to client

- SSH packet server to client: 158 bytes
- TCP packet (ack) client to server: 66 bytes
- SSH packet server to client: 102 bytes
- TCP packet (ack) client to server: 66 bytes
- Web socket packet (data) server to client: 89 bytes
- TCP packet (ack) client to server: 66 bytes
- Total: 547 bytes

The following table compares the total data transferred for each transmission type.

	Server sent events	WebHooks
Client to server	1241 bytes	551 bytes
Server to client	551 bytes	547 bytes

The total data transferred is fairly consistent across all transmission types except for client to server transmission for server sent events, which are significantly higher (more than double). The larger amount of data transferred is explained by the fact that this data transmission is an HTTP post request, which has much higher overhead in terms of data transfer than a direct data transfer over a TCP connection such as with WebSockets (either direction) or server sent events (server to client).

3.3 Security

All of the transmissions that are being compared support TLS/SSL (transport layer security/secure sockets layer) which puts all of these transmission types on the same even playing field in terms of base technical security. Transmissions that are setup properly with TLS/SSL will be safe from any type of man in the middle attack. This still leaves both technologies vulnerable to general security issues when it comes to actually implementing a server that utilizes that type of data transmission.

By default, WebSockets do not have CORS (cross origin resource sharing) enabled [14]. This means that someone setting up a server that accepts WebSocket connections will allow connections from other origins by default. This could potentially be dangerous if the server is not set up to authenticate connections properly. A server set up to accept HTTP requests and send server sent events would potentially be subject to the same dangers.

Another common pitfall with WebSockets specifically is not authenticating users until after they have already established a WebSocket connection [14]. It seems like it would make sense to allow anyone to connect to the WebSocket server and then simply authenticate them when they have connected, but this could allow an attacker to send malicious data to the server. Any user connecting to the server should be authenticated before being allowed to establish a WebSocket connection to prevent this.

Another area to look into is denial of service attacks. Bidirectional communication with WebSockets utilize a single TCP connection that remains open, while bidirectional communication with server sent events utilize a new TCP connection for every transmission from client to server. This larger amount of connections could more easily lead to denial of service attacks [8].

4. Conclusion

While server sent events and WebHooks both solve similar problems, we can see that each works slightly differently and may be more appropriate for different use cases. If your application is mostly reliant on server to client transmissions then using HTTP server sent events over WebSockets may be an appealing option. HTTP server sent events had much faster transmission times for server to client than WebSockets, while transmitting roughly the same amount of data. If your application is heavily reliant on client to server transmissions, then WebSockets are likely a more appealing option due to the high amount of data transfer needed for client to server HTTP requests. If no specific requirements are needed, WebSockets would be easier to work with to have a consistent method of transmitting data, both for client to server and server to client.

References

[1] “Hypertext Transfer Protocol -- HTTP/1.1,” *IETF*, Jun-1999. [Online]. Available: https://tools.ietf.org/html/rfc2616. [Accessed: 13-Feb-2020].

[2] “When to use a HTTP call instead of a WebSocket (or HTTP 2.0),” *Windows Developer Blog*, 11-Mar-2016. [Online]. Available: https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/. [Accessed: 13-Feb-2020].

[3] “The WebSocket Protocol,” *IETF*, Dec-2011. [Online]. Available: https://tools.ietf.org/html/rfc6455. [Accessed: 13-Feb-2020].

[4] “Using server-sent events,” *MDN Web Docs*, Dec-2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events. [Accessed: 13-Feb-2020].

[5] M. Chaov, “Using SSE Instead Of WebSockets For Unidirectional Data Flow Over HTTP/2,” *Smashing Magazine*, 12-Feb-2018. [Online]. Available: https://www.smashingmagazine.com/2018/02/sse-websockets-data-flow-http2/. [Accessed: 13-Feb-2020].

[6] E. Bidelman, “Stream Updates with Server-Sent Events - HTML5 Rocks,” *HTML5 Rocks - A resource for open web HTML5 developers*, Nov-2010. [Online]. Available: https://www.html5rocks.com/en/tutorials/eventsource/basics/. [Accessed: 13-Feb-2020].

[7] “Server Sent Events,” *HTML Standard*, 12-Feb-2020. [Online]. Available: https://html.spec.whatwg.org/multipage/server-sent-events.html. [Accessed: 13-Feb-2020].

[8] “Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP,” *IETF*, Apr-2011. [Online]. Available: rfc-editor.org/rfc/pdfrfc/rfc6202.txt.pdf. [Accessed: 12-Feb-2020].

[9] A. Zlatkov, “How JavaScript works: Deep dive into WebSockets and HTTP/2 with SSE how to pick the right path,” *Medium*, 12-Dec-2018. [Online]. Available: https://blog.sessionstack.com/how-javascript-works-deep-dive-into-websockets-and-http-2-with-sse-how-to-pick-the-right-path-584e6b8e3bf7. [Accessed: 13-Feb-2020].

[10] “The WebSocket API (WebSockets),” *MDN Web Docs*, Jan-2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Accessed: 13-Feb-2020].

[11] “Writing WebSocket servers,” *MDN Web Docs*, Apr-2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers. [Accessed: 13-Feb-2020].

[12] “About,” Node.js. [Online]. Available: https://nodejs.org/en/about/. [Accessed: 02-Apr-2020].

[13] P. Richeal, “Network security project,” GitHub. [Online]. Available: https://github.com/pricheal/network-security-project. [Accessed: 02-Apr-2020].

[14] M. Mohan, “How to secure your WebSocket connections,” freeCodeCamp.org, 03-Sep-2019. [Online]. Available: https://www.freecodecamp.org/news/how-to-secure-your-websocket-connections-d0be0996c556/. [Accessed: 02-Apr-2020].