DALARNA
UNIVERSITY

# Real-Time Synchronization of Multi-Window Web-Applications

## Combining SSE & XHR over HTTP/2 as an alternative to WebSockets

Anton Fladvad
Anders Khans

*Supervisor*: Serena Barakat
August Dellestrand (Triona)
*Examiner*: Vijay Pratap Paidi

*Thesis for Bachelor Degree in Informatics (GIK28T)*
*Degree Project for University Diploma in Informatics/Thesis for Degree of Bachelor of Science in Informatics*

*8 June 2021*

*Dalarna University, Sweden* – www.du.se

*Published in full-text as open access*

## ABSTRACT

Modern web-applications often need to be able to handle multi-window views that are dynamically synchronized depending on user input, as well as continuous and rapid data transfer between the client and the server. The WebSocket protocol has seen widespread industry use when it comes to bidirectional, real-time communication. However, its inherent security flaws make the prospect of other adequate alternatives highly relevant. SSE combined with XHR is a technology that has been overlooked by developers due to the shortcomings of the HTTP/1.1 protocol regarding efficiency. However, the HTTP/2 protocol expands upon and streamlines the core features of HTTP/1.1 while also providing full-duplex functionality. The introduction of the HTTP/2 protocol has given rise to speculations regarding a potential comeback for SSE as a viable contender for the WebSocket protocol.

The aim is to evaluate whether the combination of SSE and XHR over HTTP/2 could be an equally or more efficient alternative to the WebSocket protocol for real-time data synchronization between multiple web-application views. This is done through the design and creation of two proof-of-concepts supported by the theoretical foundation established by conducting a literature review.

The literature in this area indicates a lack of existing research concerning SSE over HTTP/2. The proof-of-concepts has produced empirical data, consisting of average data transmission times, that points to SSE/XHR performing as well as, if not better than the WebSocket counterpart.

The results confirm that a combination of XHR and SSE over HTTP/2 is an adequate alternative to WebSockets within the scope of this study.


*Keywords:* SSE, WebSocket, HTTP/2, “bidirectional communication”, “browser synchronization”, real-time

## INNEHÅLL | CONTENTS

## ABREVIATIONS AND DEFINITIONS

| | |
|---|---|
| POC | Proof-of-Concept |
| XHR | XmlHttpRequest |
| SSE | Server Sent Events |
| HTTP/1.1 | HyperText Transfer Protocol version 1.1 |
| HTTP/2 | HyperText Transfer Protocol version 2 |
| TLS | Transport Layer Security |
| HTTPS | An extension of HTTP using TLS for encryption |
| HTML | HyperText Markup Language |
| DOM | Document Object Model |
| TCP | Transmission Control Protocol |
| EventSource | HTML5 interface to enable server-push technologies |
| ws | The form of TCP connection utilized by WebSocket |
| wss | The encrypted form of the ws TCP connection |
| Selenium | A toolset for automating web browsers |
| NodeJS | A java-script runtime, created to be event-driven and asynchronous |
| KoaJS | A web framework for NodeJS |

## FIGURES AND TABLES

# 1 Introduction

## 1.1 Background

Many services are moving from desktop applications towards web-based applications. This has created a shift in how the service is being used. (Shimomura et al., 2009)

Traditional desktop applications, although reliable in delivering a consistent user experience, are limited to the device they are installed on, both in terms of operating system/hardware and the physical location of the device. Web-based applications remove those physical restrictions as well as the installation requirements and are accessible from practically any device with a web browser and a working internet connection. (Murley et al., 2021)

The rapid development in the area of cloud-computing has likely played a significant role in the migration and adoption of web-based applications among industry actors. The cloud diminishes the need for users, including businesses, to invest in its own hardware and software since it is provided as a flexible service hosted in the cloud that scales with user requirements. This provides businesses with the possibility of adaptive resource allocation and responsive expansion. (Rewatkar & Lanjewar, 2010)

Since web-based applications are based on HTML, often together with JavaScript and CSS, it is possible to design applications to be responsive and adapt to the medium on which it is accessed while still using the same code base. This enables versatile code reusability and thereby making the development quicker (Shahzad, 2017). Web-applications also enable the user to control the interface being used. The ability to simply open a new browser-window, or tab, to extend or expand the view of certain elements of the application creates new possibilities for the user to find a way to personalize the interface to fit the workflow of the user. (Rewatkar & Lanjewar, 2010)

Some organisations are therefore looking to migrate or extend existing desktop applications into a web-based variant. In order to do so without compromising usability and efficiency, the web-application may need to be able to handle multi-window views that are dynamically synchronized depending on user input. This becomes increasingly complex if the application requires continuous and rapid data transfer between the client and the server in both directions, which in turn requires a reliable, low-latency solution capable of bidirectional communication. (Aburajab & Salman, 2019)

There are several technologies that have been used to handle bidirectional communication between client and server, such as HTTP polling, long-polling and HTTP streaming. However, while these technologies manage to do this successfully, the data overhead that inevitably comes with every HTTP request impacts the latency to such a degree that it fails to meet the needs of more complex applications (Aburajab

& Salman 2019). These technologies were never intended for bidirectional communication in the first place. The only way that developers have been able implement bidirectionality is by manipulating the server, forcing it to use multiple TCP connections for each individual client. Since polling technologies rely on the client continuously sending requests to the server to check for any new data, regardless if there are any, these technologies put unnecessary strain on the server with poor scalability (Fette & Melnikov, 2011).

SSE, or Server-Sent-Events, is a technology that was introduced in 2006 as a way of allowing a server to push data to clients without the client explicitly requesting it. It does this by providing a type of subscription model where clients agree to receiving updates from the server whenever they are available. . SSE is currently used by e.g. Twitter and other live-feed subscription platforms requiring real-time updates, such as stock exchanges or news push-notifications. (Ably , n.d.). The technique was later adopted by the World Wide Web Consortium, or W3C, through the EventSource API, which currently is a standardized part of HTML5 (W3C, 2021a). However, while this technique seemingly solves the issue of needing constant client requests in order for the server to be able to push data to the client, it is still unidirectional in the sense that the client would still need to query the server. One method that can be used for this purpose is XML Http Requests (XHR), which is a JavaScript-based API providing client-side functionality for sending data to a server (WHATWG, 2021b), every time data needs to be transferred in the other direction.

The WebSocket protocol was officially introduced to the web at large by 2011 as a proposed solution to eliminate the need for 'abusing' the HTTP protocol through manipulation in order to get bidirectional communication. Instead of relying on the client to blindly request data from the server, WebSockets allow the server to independently provide clients with new data as it becomes available (Fette & Melnikov, 2011). It also strips away a significant part of the obligatory overhead otherwise related to HTTP requests, as well as allowing it to operate on a single TCP-port, making it more efficient when handling bidirectional real-time communication between server and client(s) (Fette & Melnikov, 2011) .

Although WebSockets solve the issue of the large request overheads otherwise associated with the HTTP/1.1. protocol, it poses some drawbacks such as lack of compatibility with certain browsers and protocols (Kucik, 2019). It is also missing a lot of the standardized features and security measures that come "out-of-the-box" with the HTTP protocol, such as being stateless which improves overall security and scalability, and the support for cross-origin resource sharing (CORS) (MDN Web Docs, 2021). However, judging by the overall level of industry adoption compared with SSE, WebSockets has seemingly still been favoured over SSE for real-time bidirectional web-applications such as online gaming or chatrooms despite its drawbacks, while the

interest for SSE has slowly but steadily been dwindling (Murley et al., 2021).

However, the introduction of the HTTP/2 protocol in 2015, which expands upon and streamlines the core features of HTTP/1.1 while also providing full-duplex functionality (Fietze, 2017), has given rise to speculations regarding a potential comeback for SSE as a viable contender for the WebSocket protocol (Burian, 2018).

## 1.2 Problem Description

The new possibilities of web-applications also introduce some concerns. There are numerous aspects to be considered when migrating from a desktop application to a web-based application, e.g., security, compatibility, information handling, and user availability (Murley et al., 2021). Another aspect is the possibility of performance problems caused by an unreasonably high number of connections.

Moreover, there are also concerns about the reliability of information between different browser windows. If the service in question needs to communicate large amounts of information to the user there is a risk that the user will receive outdated information (de Souza et al., 2018), meaning information that is being displayed on the client is not the same as the one in the data-source. This would be a significant drawback when working with time-sensitive tasks or complex calculations (de Souza et al., 2018) .

Many of the limitations that hindered SSE, such as the large overhead or being constrained to only 6 concurrent connections per browser, have been resolved by using HTTP/2 instead of the older HTTP/1.1. The lack of support that earlier haunted SSE has been reduced, and SSE is now supported by all major browsers (MDN Web Docs, 2021). Despite this, the overall global level of adoption among websites is still significantly lower when compared to the use of WebSockets (Murley et al., 2021).

Although the WebSocket protocol has seen broad use with acceptable contextual performance (Appelqvist & Örnmyr, 2017), it has foundational security vulnerabilities such as risks of DOS-attacks, insecure direct object references (IDOR) and lack of standardized security configurations (Taneja, 2021). This makes the prospect of potential equivalent alternatives most relevant.

This has given rise to the acknowledgement and definition of the following issue:

Is it possible to implement a combination of SSE/XHR over HTTP/2 that is comparable to the bidirectional abilities of the WebSocket protocol?

## 1.3 Objective

This study focuses on technologies for handling the synchronization of bidirectional real-time data in web-applications where one or more users have multiple browser windows displaying different views of the same service. The technologies that are examined are the WebSocket protocol and the combination of XHR and SSE over HTTP/2.

The aim is to evaluate whether the combination of SSE and XHR over HTTP/2 could be an equally or more efficient alternative to the WebSocket protocol for real-time data synchronization between multiple web-application views. This is done through the design and creation of two Proof-of-Concepts (POCs) supported by the theoretical foundation established by conducting a literature review.

We aim to evaluate the performance of the selected technologies by measuring the data transmission times, i.e. the time it takes for a message to be transmitted from one client to another (or in this case, from one browser window to another) of each respective technology. The average transmission time for each technology is then calculated and used for comparison.

The main knowledge contribution is an IT-artefact consisting of two working POCs demonstrating the technologies that have been researched, along with empirical data consisting of data transmission times collected from evaluating the POCs.

This study is carried out in collaboration with an IT-based business partner situated in Dalarna, Sweden, which has expressed an interest in the possibility of migrating one or several of their currently desktop-based applications to a real-time web-based solution.

## 1.4 Research Question

Is SSE over HTTP/2 an adequate alternative to WebSockets for real-time bidirectional data synchronization in multi-window web-applications in terms of performance?

## 1.5 Scope and limitations

The main scope of this thesis will be confined to web-applications using multiple browser windows on the same client, with a focus on the communication intended for manipulation of DOM elements.

The thesis will not consider hardware or the differences in frameworks. Rather, the focus will be on the technologies themselves, regardless of what underlying technologies they run on. To achieve this, the POCs produced will be built on a unified configuration (detailed in section 4.2), both to limit the scope of the thesis and to create comparable data.

Server Push is one of the new features introduced with HTTP/2 with the aim of reducing load times between server and client (Grigorik & Surma, n.d.). Despite the HTTP/2 protocol being in-scope for this

research, Server Push will not be taken into consideration. This is due to the reason that Server Push mainly focuses on streamlining the initial requests and responses of multiple static resources, e.g., external script files or stylesheets, between the client and the server, as opposed to continuous and dynamic real-time data transmission.

# 2 Theoretical Frame of Reference

The following chapter aims to provide a contextual summary of the web 'ecosystem' of which the technologies that are examined relate to.

## 2.1 What Is Server-Push Technology?

The web as we know it is based on the HTTP protocol (de Souza et al., 2018). HTTP is a request/response technology and is by nature not able to send information to the client without first receiving a request. (Shuang et al., 2013; de Souza Soares, et al., 2018) This means that the connection between server and client is lost after the request has been answered, and the client needs to make a new request if it wants to check if and what changes have been made.

To enable servers to push information several technologies have been developed. Among the most commonly used are HTTP -Polling and its further development Long-Polling, WebSockets, and to a lesser extent Server Sent Events (SSE). (Murley et al., 2021)

HTTP -Polling and Long-polling work by periodically asking, or "polling", the server to see whether there is any new information available (de Souza Soares et al., 2018). WebSockets and SSE work by keeping the connection open, enabling the server to push the information only when it is needed.

## 2.2 Unidirectional vs. Bidirectional

One of the biggest differences between technologies is whether they communicate in a bidirectional or unidirectional manner; in other words, whether they are able to send data in both directions. (de Souza Soares et al., 2018)

In addition to whether a connection is unidirectional or bidirectional, there is the difference in duplex. In contrast to the directional capabilities, duplex refers to the connection's ability to send data in both directions at the same time, without having to queue requests or wait for a response. (Appelqvist & Örnmyr, 2017)

## 2.3 WebSockets

While WebSockets uses the HTTP protocol to perform an introductory handshake which establishes the connection between the client and the server, it immediately "upgrades" the connection from HTTP to its own pure TCP connection type. This is the only relation that the WebSocket protocol shares with HTTP, but it is enough to make it compatible with browsers using HTTP. (de Souza Soares et al., 2018)

By upgrading the connection to TCP, WebSockets allows for full-duplex communication for multiple connections in both directions while keeping the amount of overhead and metadata to a minimum in order to increase efficiency. The WebSocket protocol maintains a persistent connection between client and server once the opening handshake has been properly reciprocated, which requires the connection to be

properly closed when finishing the session in order to mitigate security risks. (W3C, 2021b)

WebSockets can be used either using an unencrypted connection (ws://) or an encrypted connection (wss://), which can be compared with HTTP and HTTPS respectively. (W3C, 2021b)

## 2.4 HTTP/2

HTTP/2 was published in 2015 (Ramli et. al., 2018) and has since been adopted by an estimated 45% of the one million websites examined by Murley et. al (2021)

To enable full duplex communication HTTP/2 uses streams and binary framing within the application layer. Each stream has a unique identifier and priority number, and the protocol splits the message into frames marked with which stream each frame belongs to. (Ramli et. al., 2018)

HTTP/2 includes server push, a feature designed to enable data to be sent to the client without the client requesting it (Murley et. al). This feature is not designed for real-time communication but to limit the amount of GET requests by serving static content such as CSS and JavaScript without the client asking for it (Grigorik & Surma, n.d.).

Even if HTTP/2 itself does not deliver encryption, most browsers only accept handshakes over HTTP/2 if the connection utilizes TLS. (Grigorik & Surma, n.d.)

## 2.5 SSE

The Server Sent Events protocol (SSE) is by itself unidirectional, meaning that after the stream has been established it can only send data to the client. Communication from the client to the server needs to be made using some other channel, for example XHR. (WHATWG, 2021a, 2021b)

SSE uses the underlying protocol for communication, meaning that if the server utilizes http/1.1, so does the SSE Stream. When using SSE over HTTP/1.1, there is a limit of 6 open connections per browser. This limit is negotiated between the server and the client over HTTP/2, with a default value of 100 connections per browser (MDN Web Docs, 2021) .

# 3 Literature Review

By conducting a literature review, both SSE and WebSockets are examined more closely. By highlighting certain aspects that will not be covered by the POCs, the aim is to provide a more complete picture of both technologies and their respective advantages and disadvantages. In order to provide a more concrete overview of the summary of the literature review, it is divided into subchapters focusing on the following aspects:

*Compatibility*: Does the technology have the functionality required and how well will it work with other techniques?

*Scalability*: The technology's ability to perform as the service evolves and expands.

*Security*: The technology's ability to keep information secure and protect the service againt external threats

*Level of Adoption*: Is the technology recognized by the development community and established as a valid option and will there be continuous support? .

The summary is followed by an overall reflection discussing observations made during the review process.

An estimated 15-20 publications were initially selected and examined further, particularly focusing on their respective abstracts, results and conclusions. This process led to a final selection of the following publications that were to be the subject for the literature review.

## 3.1 Summary

The following part details our findings from the literature review regarding WebSockets and the combination of HTTP/2 with SSE, divided into subcategories that were deemed relevant for our research, based on the research objectives.

The aim of the literature review is to establish an adequate theoretical foundation that will be used in order to answer the research questions previously stated.

### 3.1.1 Compatibility

According to Murley et al. (2021), WebSockets are, since 2013, supported by all major and modern browsers with the exception of older versions of Internet Explorer. This provides the possibility of broad code reuse across both browsers and devices. (Murley et al., 2021)

The support for SSE has not been as straightforward as with WebSockets, with support for Microsoft Edge being added as late as 2019 and is, at the time of writing, still not supported by Internet Explorer (Murley et al., 2021). Moreover, Server-Push, which is a feature included in HTTP/2 allowing asynchronous data transfer from the

server to the client without the need for specific requests, also lacks support by Internet Explorer as well as the mobile browser iOS Safari (Murley et al., 2021) .

Although WebSockets operate under its own protocol, the initial connection request consists of an HTTP handshake thus making it compatible with browsers using the HTTP/1.1 protocol (Ogundeyi & Yinka-Banjo, 2019).

### 3.1.2 Scalability

Since the WebSocket protocol is able to establish a full-duplex bidirectional communication channel between the client and the server based on a single TCP-connection, more of the server's hardware is available for processing the requests (de Souza et al., 2018). According to Ogundeyi and Yinka-Banjo (2019), by using the WebSocket protocol instead of HTTP when working with real-time data transfer also eliminates a significant amount of excess data (consisting of e g., HTTP headers). This improves the scalability of web-based applications.

HTTP/2 enables multiplex communication by utilizing binary framing and streams (Ramli & Jarin, 2018) and thereby limiting the number of TCP connections to one per browser.

Appelqvist and Örnmyr (2017) note that when WebSockets is used, the overhead is in relation to the payload. While the overhead over SSE is only in relation to the number of used fields in the message. Meaning that SSE should have an advantage if the payloads are bigger.

The WebSocket protocol, in contrast to other technologies such as Polling, does not rely on client requests in order for the server to send updates to the client, thus reducing a lot of traffic that would otherwise occur. This could greatly affect the performance and server load of larger scale web-applications (de Souza et al., 2018). This is supported by Appelqvist and Örnmyr (2017) who found that both WebSockets and SSE outperforms Polling and Long-polling. In the Study by Appelqvist and Örnmyr the overall difference between WebSockets and SSE was considered too small to be measurable under the test condition (2017).

Meanwhile, the results from the tests run by Słodziak and Nowak (2016) on bidirectional communication show that the combination of SSE and XHR has an approximately 15% longer transmission compared to WebSockets. The study does however fail to specify whether the tests are using version 1.1 or version 2 of the HTTP protocol.

Murley et al. (2021) attempt to map out the extent of current WebSocket usage across the web. This is done by studying websites on the Tranco Top 1 million list; a compiled collection of ranked websites that can be used for research and analysis (Pochat et al., 2019). During their evaluation of a website hosting a web-based game they observed an extremely high frequency of messages being sent from the client to the server. While they refer to the phenomenon as a possible configuration

error, the authors also point out that the fact that the website still manages to function speaks to the level of scalability and persistence of the WebSocket protocol. (Murley et al., 2021)

### 3.1.3 Security

Modern browsers require the initial handshake to utilize TLS to allow HTTP/2, making the communication encrypted without the need for further modification. WebSockets can also utilize TLS but to accomplish this it needs to be enabled, otherwise an unencrypted connection is used (Ramli & Jarin, 2018)

Murley et. al. (2021) finds that 14.1% of WebSocket servers they surveyed allowed unencrypted connections. They do note that the unencrypted services they examined further were mostly trackers and other types of insensitive services, but that it cannot be ruled out that sensitive information is being sent in plain text.

There is also the issue of misconfiguration regarding origin headers when setting up the WebSocket connection. By examining the HTTP origin header, the WebSocket protocol is able to identify and authenticate the origin of the incoming upgrade request. Although several of the commonly used libraries provide functionality to expedite and simplify this process, many of them allow for any origin to connect to the WebSocket server by default. While this can be mitigated through the built-in functions, it paves the way for developers to unknowingly leave the door unlocked. (Murley et al., 2021)

In accordance with the official WebSocket documentation from Fette and Melnikov (2011), de Souza Soares et al. (2018) also recognize the persistent connections utilized by the WebSocket protocol as potentially constituting a major security risk. This mechanism may be a target for DOS-attacks since the connection needs to be explicitly closed by either the server or the client. This does not include any built-in default failsafe measures.

### 3.1.4 Level of Adoption

As previously mentioned, Murley et al. (2021) focuses their research on surveying the web ecosystem of bidirectional real-time data synchronization. While the paper explicitly states the emphasis on the WebSocket protocol they also analyse the adoption of SSE.

Murley et al. (2021) estimate that roughly 6% of websites currently use WebSockets which, compared to earlier studies from 2018, indicate an increase by approximately 4 percentage points in overall website adoption over the last 3 years. In contrast, SSE was found to be used in only 0.05% of the top 1 million websites ranked by Tranco. Murley et al. (2021) speculate that this is a consequence of SSE not being supported by all major browsers. This would give developers a highly justifiable reason to opt for other technologies when implementing real-time functionality to their applications.

On the other hand, the adoption of HTTP/2 by the same websites was estimated by Murley et al. (2021) to be about 45% in total. Despite the protocol being introduced relatively recently in 2015, the paper does not say anything about the prevalence of combining SSE and HTTP/2.

## 3.2 Reflection

Both SSE and WebSockets have been examined in earlier research, but this research has mainly been conducted over HTTP/1.1. Of the literature in this study only Ramli et al. (2018) examines the impact of implementing HTTP/2 specifically. Other studies mention HTTP/2 as a subject for future studies and the potential it brings (Murley et al., 2021).

The results found using HTTP/1.1 shows that WebSockets performs better under these conditions when it comes to response time (Słodziak et al., 2016). But as noted by Appelqvist and Örnmyr (2017), the load on the server-side is comparable indicating that the difference in response time is caused by the communication and not due to limitations of the server itself.

One of the reasons why SSE seemingly has been overlooked by developers over the years may be due to shortcomings in terms of compatibility. Since the technology has not been prioritized and adopted by all browsers (Murley et al., 2021). However, with Microsoft Edge adding late support in 2019, all major browsers, with the exception of Internet Explorer which consequently is being officially dropped by Microsoft (2021) in August of 2021. Whether this will affect the level of industry adoption regarding SSE is unknown. But it does eliminate the lack of support as a reason for developers to exclude SSE from consideration when deciding which technology to implement.

The literature mentions several security risks and considerations regarding WebSockets, some linked to misconfigurations during implementation (Murley et al., 2021).Others seemingly linked to the technology design itself (de Souza Soares et al., 2018). Meanwhile, the security aspect of SSE does not appear to be discussed in detail in any of the publications that were examined.

Despite the security concerns that are brought up in the literature, WebSockets remains as one of the most used protocols for bidirectional communication. The reason for this is not addressed by the literature, but a combination of ease-of-use and good performance likely plays a part, as well as the lack of equivalent or better alternatives..

# 4 Methodology

The following part details the processes, methods and main concepts behind the proof-of-concepts (POCs) that were created specifically for the purpose of this research. The aim is to provide transparency and legitimacy to the work, as well as to lay the foundation for potential future research.

The design and creation of this proof-of-concept aims to clarify the potential pitfalls that might occur during development using these technologies.

## 4.1 Proof of Concept

Two POCs were created. One using the XHR/SSE combination over HTTP/2 and the other using the WebSocket protocol. The POCs emulate a text editor for collaboration where each change made in one editor should be mirrored in real-time on editors shown on different clients. This is similar to the concept of other real-time collaboration tools such as Google Docs.

To create comparable data, both POCs are built on the same base application, meaning that the foundational code structure of both POCs is identical to each other, only differing in how the communication between client and server is handled.

### 4.1.1 Common Ground

On the server side, NodeJS is being used to provide server functionality with the KoaJS framework on top. KoaJS was chosen because of its light-weight nature. It also comes with minimal bundled functionality, only providing the essential components for setting up a basic server. All additional functionality must be imported as separate modules. This approach lowers the risk of the acquired data being tainted by unnecessary middleware before being collected for analysis.

The server acts only as a relay and does not contain any form of data storage. Messages from the client are not stored; they are instead redistributed to the other clients.

The client consists of a simple ReactJS app, with a single component, as shown in figure 1. This component features an editor provided by the react-medium-editor (npm, 2016) package. The editor is used to provide better functionality and eliminate bugs that occurred using the HTML5 Textfield. The component holds the data to be displayed as a state, which enables the different POCs to manipulate the editor by setting this state. The editor's onChange event is used to trigger the real-time communication from the client to the server.

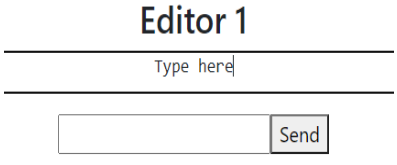



*Fig. 1 - Screenshot of the POC UI*

### 4.1.2 WebSocket Implementation

To enable the WebSocket protocol, a separate server needs to be set up that handles all communication that concerns WebSocket. Since the KoaJS-based server in this case only serves the React application as a static build-folder, a module called "koa-websocket" (Cremin, 2019) was used in order to wrap the KoaJS server instance and add support for the WebSocket protocol. Additional middleware was implemented to handle the initial HTTP upgrade request as well as specified functionality for handling the connection and bidirectional messaging.

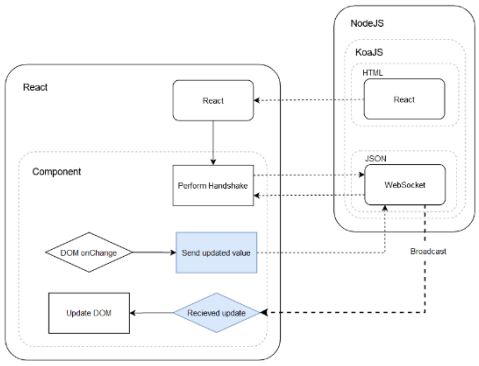



*Fig. 2 - Diagram of PoC using WebSockets*

The client implementation of the WebSocket protocol is straightforward. It consists of instantiating the WebSocket object and setting the provided events with handlers. Since WebSocket provides bi directional communication no other functionality is needed.

### 4.1.3 SSE Implementation

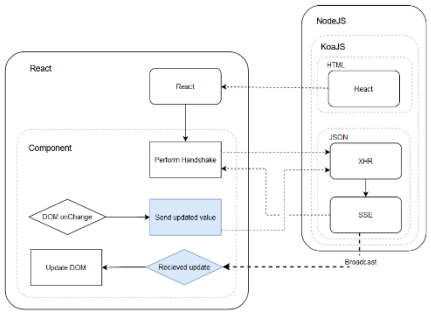



*Fig. 3 - Diagram of the POC using XHR/SSE*

NodeJS provides functionality to enable HTTP/2 without additional modules. But it does require SSL certificates. This is because modern browsers only accept HTTP/2 over HTTPS. In this case the keys were created using OpenSSL and passed as parameters to the HTTP/2 server instance.

The server needs endpoints to be set up both for initialization and to enable the client to send data to the server. This means that this POC required additional modules to be imported in order to create the functionality needed.

The client uses XHR requests as described in figure 3. This is because SSE on its own is inherently unidirectional: only transmitting data from the server to the client. In order to achieve bidirectionality, the module AXIOS (npm, 2020) was implemented to add support for client-to-server transmission using XHR. SSE uses EventSource to receive data from the server, but since ReactJS handles this as any other object, making it behave as WebSocket.

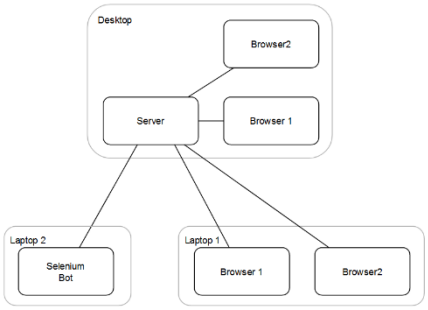
## 4.2 Evaluation

To evaluate the developed POCs, the transmission time i.e the time it takes for a change to be registered to the response being received from the server, was recorded. In the case of SSE, the response to the change is received through a different technology (SSE) than the one being sent (XHR), the 200 response to the XHR request is not usable to measure the transmission time, and a different way of measuring was needed.

This was done by giving every outgoing message from the client a unique ID and recording this ID along with the timestamp when it was sent into memory. The same ID is then sent along with the message to the server to be included in the following broadcast. The broadcast is then recorded by the clients on arrival along with the timestamp. The data collected were then processed to match the IDs from the different clients to get an average transmission time of the messages.

To automate the evaluations, Selenium was used. Selenium is a framework for automated testing, but in this case, it is only used for automation. To run the same scenarios on the different POCs. Selenium enabled one of the clients to be automated and emulate keystrokes in the editor component. This way identical scenarios were run, creating comparable data for the evaluation.

The scenario emulated 200 keystrokes, each time adding a single character to the payload being sent and received. This scenario was chosen to create small requests sent in sequence.




Three different computers were used for the test scenario: one desktop computer to act as the server and two laptops to act as clients. The desktop handles all server functions as well as running two browser windows displaying the site, making it act as both server and client, described in figure 4. One of the laptops displayed the site in two different browser windows and the other was used to run the Selenium bot. The evaluations were executed over a local area network.

Although it is possible to run the application on a single machine, the motivation behind using several computers were twofold; to increase the amount of collected data in order to get a more solid evaluation foundation, and to simulate an environment closer resembling a real-life context by both conceptually and physically separating the server from the clients.

The POC in this particular case is simplified in the way that the client opens the exact same application view in multiple browser tabs. The

data transmission technology used by the prototype would still be applicable to a design featuring several different views, since it would just be a matter of redirecting the server's responses to a different ReactJS component.

The hardware of the desktop acting as the server were as follows:

CPU: AMD Ryzen 5 1600 3200Mhz

memory: 16GB

operating system: Windows 10

In addition to the two developed POCs, the POCs using SSE were also evaluated over HTTP/1.1 to clarify the effects of HTTP/2.

In order to exclude the risk of getting inaccurate transmission time measurements, both laptops were programmed to sync their internal clocks to the one used by the desktop computer, i.e., the server.

# 5 Results and Discussion

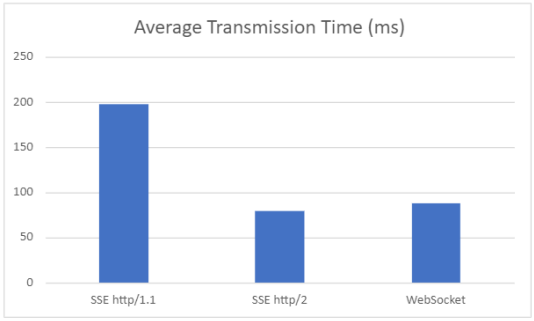



*Fig. 4 - Chart displaying average transmission times*

The data collected from running the POCs, both of which consist of the same application with interchangeable communication protocols, suggest that SSE/XHR over HTTP/2 is able to match, and possibly even outperform, the WebSocket protocol in this particular context. Performance-wise, the HTTP/2 protocol seems to improve the bidirectional capabilities of combining SSE and XHR significantly compared to HTTP/1.1. With results averaging close to three times faster transmission times, as illustrated by Figure 5 and Table 1. The PoC using WebSocket only uses HTTP to serve the ReactJS components, and for the initial handshake. For all other communication it uses the TCP connection provided by WebSocket. Because of this, only the PoC using the SSE/XHR combination was evaluated over HTTP/1.1

| **Protocol** | SSE HTTP/1.1 | SSE HTTP/2 | WebSocket |
|---|---|---|---|
| **Avg. transmission time (ms)** | 198.45 | 79.889 | 88.171 |

*Table 1 - Table of average transmission times (n=200)*

The literature review indicates that SSE has earlier had drawbacks that hindered the technology from taking a more prominent role in the web-application landscape. But it also tells us that the use of HTTP/2 is not very well researched, even if it is mentioned as something worth looking into.

The review did reveal that HTTP/2 is already widespread and implemented on a large number of webservers. Since our POCs indicate that HTTP/2 significantly improves performance, it is difficult to draw conclusions from earlier research since it focuses on HTTP/1.1.

Regarding scalability, one potentially major drawback of using SSE over HTTP/1.1 is the limit of concurrent connections. Since this has been mitigated with HTTP/2, SSE over HTTP/2 appears as at least equivalent to the WebSocket protocol in terms of scalability.

When it comes to protocol security, a lot of the issues detailed by Fette and Melnikov (2011) in the official WebSocket documentation seem to still be present in 2021. There are measures available that can increase the level of security, but these require explicit implementation by developers and can often be overlooked. There also seems to be a lack of

best-practice standards when implementing WebSockets. The findings from the literature review does not reveal much regarding the security of SSE. This is because it relies completely on the HTTP and HTTPS protocols to do most of the heavy lifting, many of its related security features are already in place.

Even though the EventSource is a part of HTML5 standard, SSE has been suffering from lack of adoption that WebSocket has not. SSE was not supported by Microsoft Edge until 2019 and Internet Explorer never supported it. But at the time of writing all major browsers support SSE, thereby removing this obstacle.

## 5.1 Considerations

During earlier stages of the design-and-creation process, the initial base application prototype design was significantly more complex compared to the end result. Originally, the application contained several tabs integrated into a single page, and the data that was put in by the user in one tab was consequently manipulated by the server. O ften performing semi-advanced mathematical calculations before being passed on to a different tab in the same application.

While this arguably was a more appealing design, at least in terms of presentation and demonstration, one might question the validity of the measurement data. Every addition of components that handle the data in any way might introduce e.g. unnecessary latency, resulting in inaccurate measurements.

There were also some considerations made regarding the method used for time synchronization, specifically how to establish a proper collective benchmark for all devices involved in the evaluation. During the initial POC test runs, it became apparent that the devices own internal clocks differed, which in turn led to the overall measurement data being skewed. After doing some research on the subject and trying out a few different options, the method of using one of the devices as the “master clock”. The remaining devices used the master as a reference point for setting their own internal clock, leading to significantly more consistent and leveled results.

# 6 Conclusion

This thesis aimed to answer the following question:

Is SSE over HTTP/2 an adequate alternative to WebSockets for real-time bidirectional data synchronization in multi-window web-applications in terms of performance?

Although the extent of the POCs created may be considered relatively limited, it nevertheless indicates that it is possible to implement a combination of SSE/XHR over HTTP/2 that possesses real-time data synchronization abilities similar to the WebSocket protocol, albeit utilizing slightly different mechanisms.

The findings of this thesis confirm that a combination of XHR and SSE over HTTP/2 is a viable alternative to WebSockets in the context that has been examined. The scope of this research is not extensive enough to draw any conclusions regarding larger scale applications and/or the transmission of more complex data. But it does confirm that the foundational mechanisms for enabling real-time bidirectional data synchronization with acceptable performance using XHR and SSE over HTTP/2 are in place.

# 7 Future Work

The literature review conducted shows that there is a lack of research published that investigates the impact of HTTP/2 in server-push scenarios. There is a possibility that the technology has been overlooked as a subject of research because of SSEs earlier shortcomings, such as browser-support and its unidirectional nature.

The evaluations of the POCs in this thesis are not sufficient enough to draw any precise conclusions outside of its scope. They do indicate that XHR/SSE over HTTP/2 seem to be at least comparable to WebSocket B ut tests on a larger scale need to be conducted to provide more reliable data.

The hardware used is not representative of hardware being used in real-life scenarios, something that should be mended in future studies. There are also metrics that are not collected, such as CPU load and memory usage on the server. Earlier research indicates that both technologies use roughly the same amount of server resources, but this is something that needs to be examined further. The limited number of clients during the evaluations is another thing that limits the results. Future research with larger numbers of clients could reveal new information leading to different results.

## REFERENCES

*Ably. (n.d.). Server-Sent Events (SSE): A Conceptual Deep Dive. Retrieved May 27, 2021, from https://ably.com/topic/server-sent-events*

*Aburajab, A., & Salman, A. (2019, April). Interactive Blackboard for Web-based Real-time Tutoring System. In 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT) (pp. 63-68). IEEE.*

*Appelqvist, R., & Örnmyr, O. (2017). Performance comparison of XHR polling, Long polling, Server sent events and Websockets., Faculty of Computing, Blekinge Institute of Technology*

*Burian, A. (2018, June 12). WebSockets, HTTP/2, and SSE - Axiom Zen Team. Medium. Retrieved 2021-05-10 from https://medium.com/axiomzenteam/websockets-http-2-and-sse-5c24ae4d9d96*

*Chen, B., & Xu, Z. (2011, July). A framework for browser-based Multiplayer Online Games using WebGL and WebSocket. In 2011 International Conference on Multimedia Technology (pp. 471-474). IEEE.*

*Cremin, J. (2019). npm: koa-websocket. Npm. Retrieved 2021-05-13, from https://www.npmjs.com/package/koa-websocket*

*Fette, I., & Melnikov, A. (2011). RFC 6455: The WebSocket Protocol. RFC6455. Retrieved 2021-05-01, from https://www.rfc-editor.org/rfc/rfc6455.html*

*Fietze, M. (2017). HTTP/2 Streams: Is the Future of WebSockets decided? Faculty of Computer Science, TU Dresden. https://www.rn.inf.tu-dresden.de/hara/arbeiten/SCC_HS_SS2017_Fietze_-_HTTP-2%20Streams_-_Is_the_Future_of_WebSockets_decided.pdf*

*Grigorik, I., & Surma. (n.d.). Introduction to HTTP/2 | Web Fundamentals |. Google Developers. Retrieved 2021-05-13, from https://developers.google.com/web/fundamentals/performance/http2*

*IETF HTTP Working Group. HTTP/2. Retrieved 2021-04-19 from, https://httpwg.org/specs/rfc7540.html*

*Kucik, Ł. (2019, June 12). WebSockets — friend or foe? How to achieve real-time experience in your web application. Medium. https://medium.com/nexocode/websockets-friend-or-foe-how-to-achieve-real-time-experience-in-your-web-application-9ad5f1fad012*

*MDN Web Docs (2021, March 16). HTTP | MDN. Retrieved 2021-05-27 from https://developer.mozilla.org/en-US/docs/Web/HTTP*

*MDN Web Docs. (2021, April 4). Using server-sent events - Web APIs | MDN. Retrieved 2021-05-09 from https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events*

*Microsoft (2021, May 11). Microsoft 365 apps say farewell to Internet Explorer 11 and Windows 10 sunsets Microsoft Edge Legacy. TECHCOMMUNITY.MICROSOFT.COM. Retrieved 2021-05-02 from https://techcommunity.microsoft.com/t5/microsoft-365-blog/microsoft-365-apps-say-farewell-to-internet-explorer-11-and/ba-p/1591666*

*Murley, P., Ma, Z., Mason, J., Bailey, M., & Kharraz, A. (2021). WebSocket Adoption and the Landscape of the Real-Time Web*

*npm. (2020, December 22). npm: axios. Retrieved 2021-05-13, from https://www.npmjs.com/package/axios*

*npm. (2016, August 25). npm: react-medium-editor. 2021-05-13, from https://www.npmjs.com/package/react-medium-edito*

*Ogundeyi, K. E., & Yinka-Banjo, C. (2019). WebSocket in real time application. Nigerian Journal of Technology, 38(4), 1010-1020.*

*Pochat, V.L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M. and Joosen, W. (2019). Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019). Retrieved 2021-05-03 from https://doi.org/10.14722/ndss.2019.23386*

*Ramli, K., & Jarin, A. (2018). A real-time application framework for speech recognition using HTTP/2 and SSE. Indonesian Journal of Electrical Engineering and Computer Science, 12(3), 1230-1238.*

*Rewatkar, L. R., & Lanjewar, U. L. (2010). Implementation of cloud computing on web application. International Journal of Computer Applications, 2(8), 28-32.*

*Shahzad, Farrukh. (2017). Modern and Responsive Mobile-enabled Web Applications. Procedia Computer Science. 110. 410-415. 10.1016/j.procs.2017.06.105.*

*Shimomura, T., Ikeda, K., & Takahashi, M. (2009, August). Synchronization of multi-window requests for server-side regression test of web applications. In 2009 Ninth International Conference on Quality Software (pp. 129-134). IEEE.*

*Shuang, K., & Feng, K. (2013). Research on server push methods in web browser based instant messaging applications. Journal of Software, 8(10), 2644-2651.*

*Słodziak, W., & Nowak, Z. (2016). Performance analysis of web systems based on XMLHttpRequest, server-sent events and websocket. In Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology–ISAT 2015–Part II (pp. 71-83). Springer, Cham.*

*de Souza Soares, E. F., Thiago, R. M., Azevedo, L. G., de Bayser, M., da Silva, V. T., & Cerqueira, R. F. D. G. (2018, March). Evaluation of Server Push Technologies for Scalable Client-Server Communication. In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE) (pp. 1-10). IEEE.*

*Taneja, S. (2021, April 28). Everything You Need to Know About Web Socket Pentesting. Appknox. Retrieved 2021-05-10 from https://www.appknox.com/blog/everything-you-need-to-know-about-web-socket-pentesting#three*

*W3C. (2021a). Server-Sent Events. W3.Org. Retrieved 2021-04-19, from https://www.w3.org/TR/eventsource/*

*W3C. (2021b). The WebSocket API. W3C.Org. Retrieved 2021-04-19, from https://www.w3.org/TR/websockets/*

*WHATWG. (2021a) HTML Living Standard. Retrieved 2021-04-19, from https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events*

*WHATWG. (2021b). XMLHttpRequest Standard. Retrieved 2021-05-11 from https://xhr.spec.whatwg.org/*