Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Computer Engineering
2020 | LIU-IDA/LITH-EX-G-20/025-SE

Performance study of JavaScript WebSocket frameworks

Prestandajämförelse av JavaScript WebSocket ramverk

Jakob Hansson

Supervisor : George Osipov
Examiner : Ola Leifler

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`.

© Jakob Hansson

Abstract

The requirements on software and applications are getting harder. If they are perceived as slow or power hungry, the customers will be looking for other solutions instead. The development of open source frameworks is rapid. Frameworks are being built and updated continuously, with different performance, functionality and complexity. This bachelor's thesis studies and compares WebSocket frameworks in JavaScript, with focus on performance and scalability.

A pre-study was made to find out which parameters are of interest when testing and measuring the performance of web servers. A test bench was then built and plain WebSocket, Socket.IO and SockJS were benchmarked.

The study shows that there exist significant differences in performance and trends that indicate that some frameworks are superior to others in high concurrency applications. Plain WebSocket can be up to 3.7 times as fast to receive a message from the server compared to Socket.IO and 1.7 times as fast compared to SockJS. Plain WebSocket scales well in terms of response time and memory requirement with higher concurrency levels. Socket.IO requires a lot of memory which is a potential problem. SockJS does not scale well at all and does not manage the same level of concurrency as the other two.

Acknowledgments

I would like to show my appreciation for the help I received during my thesis, my supervisor George Osipov and my examiner Ola Leifler from Linköping University. I would also like to thank André Andersson at Zenon AB for all the help given to me.

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Aim 2
- 1.3 Research question 2
- 1.4 Delimitations 2

2 Theory 3
- 2.1 The WebSocket Protocol 3
- 2.2 XHR and Ajax 4
- 2.3 Node.js 4
- 2.4 Related work 4
- 2.5 The frameworks 5

3 Method 6
- 3.1 Figuring out what to measure 6
- 3.2 Test bench 6
- 3.3 Measuring 7

4 Results 8
- 4.1 Time to establish a connection 8
- 4.2 Time to receive a message upon connection 9
- 4.3 Network traces 10
- 4.4 Memory usage on server 10

5 Discussion 11
- 5.1 Results 11
- 5.2 Method 12
- 5.3 The work in a wider context 13

6 Conclusion 14

Bibliography 15

List of Figures

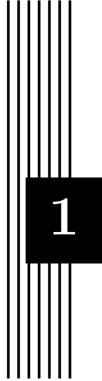
2.1 WebSocket connection establishment. 3

4.1 Time to establish a connection. 8
4.2 Average time to receive the message with different levels of clients. 9
4.3 Minimum, median and maximum times to receive the message with 7000 clients. . 9
4.4 Memory usage with different levels of clients. 10

List of Tables

3.1 The machines with associated specifications used in the test bench. 7

4.1 Network traces for one client receiving one message. 10



1 Introduction

Zenon AB released their application Blixtvakt[1] in 2015. It is a service to help the users get information about thunderstorms, available on the web, App Store and Google Play.

1.1 Motivation

As users expect more from applications the requirements becomes harder to fulfill, More functionality, faster response times and lower energy consumption. As the response time of applications increase, the user satisfaction decreases. This may lead to a decrease of customers as they turn to other solutions instead [1]. It is not uncommon to use open source frameworks when developing software to speed up the time to market. Ready-to-use functionality which already have been optimized, tested, debugged and reviewed. The development of open source frameworks is rapid, frameworks are being built and updated continuously. Different frameworks may give performance, functionality and complexity.

Zenon serves up to 100.000 concurrent users and have experienced performance issues during high surges of clients using Blixtvakt. Blixtvakt uses the SockJS[2] framework which in turn uses the WebSocket protocol to be able to communicate between the server and all connected clients. SockJS is an old framework that tries to solve compatability issues between different user agents, and is not designed with high concurrency or throughput in mind. Since Zenon has experienced problems with clients not receiving the data from their server I suspect that the clients timeout as a consequence of the server not being able to respond in time.

Socket.IO[3] is another framework which also uses the WebSocket protocol. While SockJS only had one version released under 2016 [2] Socket.IO released 15 versions under the same year [3] and may be more optimized and suitable for high concurrency applications. By using the alternative with highest performance and most functionality for the specific application, the following advantages can be achieved; faster and more responsive applications, possibly solving Zenon's performance issues. Lower development cost for Zenon and lower energy consumption on both client and server. This is the motivation to compare the performance of these frameworks and to study which solution is best suited to Zenon's requirements.

[1] http://blixtvakt.se/
[2] http://sockjs.org/
[3] https://socket.io/

1.2 Aim

The aim of this thesis is to find out if there are meaningful performance differences between different JavaScript frameworks, utilizing the WebSocket protocol, in high concurrency applications. Specifically - are there trends that can be seen with high levels of concurrency. Which trends can be seen, why, and do they affect the ability to serve 100.000 concurrent users? Is it possible to solve Zenon's performance issues by switching to another framework?

1.3 Research question

The goal of this thesis is to be able to answer the following question:

- Are there meaningful performance differences in JavaScript WebSocket frameworks that affect the scalability and the ability to serve 100.000 concurrent users?

1.4 Delimitations

In this thesis SockJS, Socket.IO and plain WebSocket are studied. There exists more frameworks which also implements the WebSocket protocol and may have different functionality and performance.

2 Theory

The WebSocket protocol uses the Transmission Control Protocol (TCP) and HyperText Transfer Protocol (HTTP) between the client and server to establish a connection. The Transmission Control Protocol is the transport layer protocol mainly used on the Internet [4]. It enables reliable communication by using acknowledgement numbers, retransmissions and timers amongst other things. The protocol is defined in multiple *Request For Commments* (RFC). The WebSocket protocol makes use of the TCP protocol in the transport layer. The HyperText Transfer Protocol is the application layer protocol that WebSockets use to initiate the connection request. It defines how files are sent through the web. For example how the user may request an image to be downloaded from a web server via a GET request. The WebSocket protocol uses HTTP GET requests to request a connection.

2.1 The WebSocket Protocol

The WebSocket protocol is a standard for two-way communication between a client and server, intended for web applications. The protocol was standardized in 2011 by the *Internet Engineering Task Force* (IETF) as RFC 6455 [5]. The protocol consists of a handshake followed by messages. The motive was to be able to get two-way communication between a web browser and a web server without the need of opening multiple parallel HTTP connections. Since WebSocket sits on top of TCP, a TCP connection is required before a WebSocket connection can be established. A TCP connection is made according to the classic three-way-handshake with three messages: *SYN, SYN-ACK* and *ACK*. Then the client sends a *Upgrade Request* over an HTTP GET request. If/when the server responds with the *Upgrade Response* the client has successfully established a WebSocket connection with the server. Two-way communication is now possible between client and server in the form of messages until any party terminates the active connection.

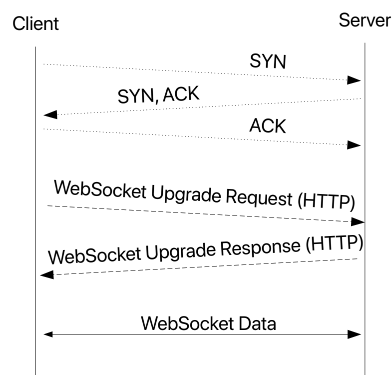


Figure 2.1: WebSocket connection establishment.

2.2 XHR and Ajax

Asynchronous JavaScript and XML (AJAX) is a set of techniques to modify web pages without the need to download a complete new web page for each update. Even though the name implies XML, multiple formats such as JSON, HTML and plain text can also be sent. The main object in Ajax is the XMLHttpRequest (XHR) which allows the browser to communicate with the web server via HTTP requests [6]. The main difference compared to WebSocket is that the client has to initiate each transmission with the server, the server has no control over when or with whom it is going to send data. Also by using WebSockets, the same connection can be used for multiple messages. While with XHR, a new HTTP request is required for each transmission.

2.3 Node.js

Node.js is an asynchronous event-driven runtime [7] built on Google's V8 engine. Google's V8 engine is an open source high-performance engine that compiles and executes JavaScript source code, used in the Google Chrome browser among others [8]. This enables execution of JavaScript code locally in a command prompt instead of in a browser. This in turn enables benchmarking and testing of JavaScript code and frameworks programatically.

2.4 Related work

There exist similar studies that investigate the performance in different frameworks and protocols. Wang et al. [9] investigated different frameworks made in Java and came to the conclusion that the frameworks *Netty* and *Undertow* are best in parallel applications while *Vert.x* and *Undertow* have the highest throughput . Wang et al. explain how they created a testbench to be able to measure the performance in the different frameworks. They implemented and configured the frameworks on their own machine, one at a time. Then they used Apache Bench on the client side to send requests and measured how the server responded. They did four measurements:

1. Requests per second with different levels of concurrency.
2. The time for each request to finish with 100 concurrent connections.
3. How the frameworks which used *HTTP keep-alive* and *HTTP close* differ in the amount of requests per second.
4. How much CPU and memory the different frameworks occupy with 100 concurrent connections.

Their study did not test how the client performs, but how the server performs with different workload. This is an important point. I assumed that the framework that allows the highest throughput, should be the best framework for parallel applications aswell since it allows more work to be done in less time. Wang et al. demonstrated that that is not always true, and that different frameworks can be optimal depending on whether you require throughput or concurrency. I kept this in mind while doing my analysis.

Skvorc et al. [10] compared the WebSocket protocol to TCP, which it relies on, in terms of latency and amount of network traffic. They show that apart from the overhead in the initial connection estblishment, the WebSocket protocol does not generate more network traffic than plain TCP. Their explanation is that each WebSocket session first requires the TCP handshake, and the overhead is the roundtrip for the HTTP request including the time taken to allocate resources and initialize the WebSocket API. They also suspect the relative new implementation of the WebSocket protocol to be a factor. Opposed to TCP which have highly optimized libraries for almost all languages and have had decades of research and development.

Abdelzaher et al. [11] studied how it is possible to achieve dynamic overload protection and ensure response times of web servers during overload by using feedback control theory. By allowing the content delivered to be degraded the system will, based on the workload, serve a certain version or send a certain response to the client. (The degraded content could be a heavier compressed image, or information about thunderstorms within a smaller radius.) Thus enabling service and content to be delivered (albeit somewhat degraded) even under big surges of clients. They explained how most of today's web servers offer poor response time when the server is under heavy load and how clients may not be able to establish connections at all. Much like the problems experienced in Blixtvakt.

Smullen and Smullen [12] compared the performance of a regular HTML application with a version utilizing AJAX. They measured the response size and response time. By utilizing AJAX, the data sent was reduced by 56 %, reducing required bandwidth and the workload on the server. The mean response time was improved by approximately 16 % and thereby improving the user experience. Their work shows that it is possible to reduce the amount of data transmitted while maintaining the user experience. I believe a similar improvement is to be found in my study between plain WebSocket and the frameworks SockJS and Socket.IO.

2.5 The frameworks

Before testing and measuring I had to read the documentation of each framework to be able to setup the test benches. To get knowledge about the functionality, which function calls are available and how the complexity may differ. This was done to better understand the possible performance differences between the frameworks, as well as being able to build the test bench faster, without the need to debug as much as otherwise would have been needed.

2.5.1 Plain WebSocket

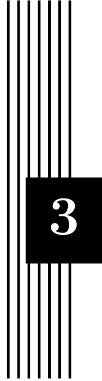
Plain WebSockets is available in almost all popular browsers [13]. It has no built-in functionality compared to Socket.IO or SockJS and the programmer is solely responsible for building the functionality wanted. For example to keep track of active connections or what to do if the protocol is not supported.

2.5.2 Socket.IO

Socket.IO is a framework which tries to utilize the WebSocket protocol under the hood. It first uses a XHR polling connection, and then tries to switch to the WebSocket protocol instead. It has a ping-pong mechanism implemented to detect disconnection and supports auto-reconnection. It supports multiplexing to create separation within applications (for example per module or permissions) with the help of namespaces. Within each namespace a room can be defined in which clients can join and leave as wanted/necessary. Socket.IO is only compatible with other instances of Socket.IO.

2.5.3 SockJS

SockJS is the current framework used in Blixtvakt. It tries to use the WebSocket protocol first. But if that does not succeed it can use other protocols and present them in a WebSocket-like abstraction. Supported protocols are, including but not limited to, *WebSocket, XHR-streaming, XHR-polling* and *EventSource*. SockJS is compatible with any client that communicates according to the WebSocket protocol.



3 Method

3.1 Figuring out what to measure

Performance testing and comparison is often made by creating a test bench to test the chosen functionality while measuring the appropriate parameters, e.g. throughput or memory usage. [9, 10, 14, 15, 12]. To be able to do a study that could help me answer my research question in section 1.3 and is useful to Zenon I had to gain knowledge about how performance of web servers can be measured. What have other done and what parameters are of interest. I did this by reading scientific articles and study the work of others.

It has been shown that some parameters are common to measure and some questions are common to answer when analyzing the performance and workload of different frameworks, protocols and network services. How many requests or messages per second is achievable (throughput) with different levels of concurrency [9, 14], the amount of data transmitted [10, 14, 12] and the response time [9, 15, 12] for example.

Based on the work of others, the following questions were made to be able to answer the research question in section 1.3.

1. How does the time to establish a connection change with different levels of clients?
2. How does the time to receive a message (upon established connection) change with different levels of clients?
3. How does the memory usage on the server change with different levels of clients?

3.2 Test bench

The test bench consisted of a workstation which spawned the selected number of clients and a server that managed the incoming requests and sent data back to the clients. The specifications are seen in Table 3.1. Three versions of the code which ran on the server and workstation was created, one for each framework to study. The three versions are identical apart from the different framework-specific function calls and events needed to perform the described scenario. The server and workstation are connected to a local area network via gigabit ethernet. All code running on the server and workstation are written in JavaScript and executed in the Node.js environment.

Machine	CPU	Memory	Operating System
Workstation	Intel Core i5-4690K	12 GB of DDR3 1333 MHz	Ubuntu 19.04
Server	Intel Core i3-8300	8 GB of DDR4 2400 MHz	Ubuntu 18.04 LTS

Table 3.1: The machines with associated specifications used in the test bench.

The clients sends connection requests to the server which responds and a connection is established. The server then sends a message to the client and measurements are being taken on the server and the clients. When the client receives the message it closes the connection. The messages sent are of the same length and format as the messages currently being sent in Blixtvakt to mimic the real application as best as possible The content of the message is a JSON string with *key:value* pairs, including but not limited to the timestamp and coordinates of the thunderstorm, as well as the distance to it.

3.3 Measuring

The questions determined in the pre-study are tested and measured in different ways, explained here.

3.3.1 Parameter 1 - Time to establish a connection

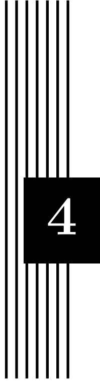
The question *"How does the time to establish a connection change with different levels of active clients?"* is measured in the Node.js environment, by using the built-in function *process.hrtime()* which returns the current high-resolution real time in nanosecond precision. The time returned by the function call is relative, not prone to clock drift and thus suited for benchmarks [16]. The test is done by issuing a number of requests in parallel. All of which takes a relative timestamp just before requesting the connection. Upon connection established a second timestamp is taken. The difference between the two timestamps is the time to establish a connection, which then is saved along the issued request. This is done for all parallel requests. The *average* time is then calculated and printed out.

3.3.2 Parameter 2 - Time to receive a message upon connection

The question *"How does the time to receive a message (upon established connection) change with different levels of active clients?"* is setup in the same way as in subsection 3.3.1. The difference is that the first timestamp is taken upon connection established. And the second timestamp is taken when it has received the message from the server. Selected statistics such as the *minimum, maximum* and *median* time as well as the *standard deviation* is then calculated and printed out.

3.3.3 Parameter 3 - Memory usage on server

The question *"How does the memory usage on the server change with different levels of active clients?"* is measured in the Node.js environment, by using the built-in function *process.memoryUsage()* which returns detailed data about the memory currently being used by the process. For example the memory usage of the V8 engine, but also the Resident Set Size (henceforth called RSS). The RSS is the amount of space occupied by the process in the main memory. Including all JavaScript objects and code. RSS is the data that is used when measuring since it is the actual memory required to run the process. The *maximum* RSS used so far is updated 10 times a second but only printed out once every second.



4 Results

In this chapter the results and data to the questions in section 3.3 are presented.

4.1 Time to establish a connection

With only 1 client, plain WebSockets managed to establish a connection in 8 milliseconds. Socket.IO was roughly 3 times slower and SockJS roughly 4 times slower. Plain WebSockets continue to be roughly 3 times faster than Socket.IO and SockJS. all the way up to 1000 clients. With 10000 concurrent clients plain WebSockets required 411 ms and Socket.IO required 519 ms. SockJS did not manage but topped out with 7000 clients at 470 ms. As seen in Figure 4.1.

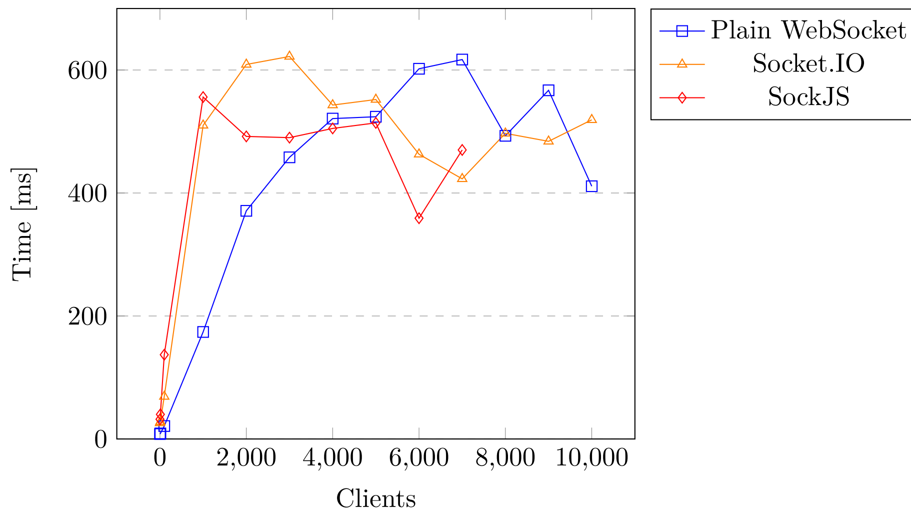


Figure 4.1: Time to establish a connection.

4.2 Time to receive a message upon connection

With only 1 client, plain WebSockets required only 1 ms to receive the message from the server, while Socket.IO required 21 ms and SockJS required 3 ms. With 1000 concurrent clients, plain WebSockets required 73 ms while Socket.IO required 525 ms and SockJS required 47 ms. With 10000 clients plain WebSockets required 466 ms and Socket.IO required 483 ms. SockJS topped out with 7000 clients at 481 ms. See Figure 4.2. In addition to the average time,

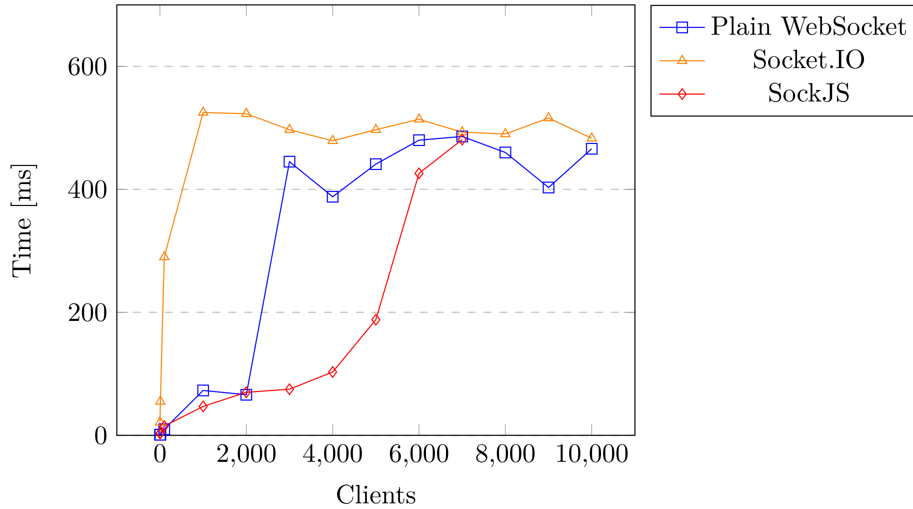


Figure 4.2: Average time to receive the message with different levels of clients.

the minimum, maximum and median time was also looked at. These measurements were taken with 7000 clients since that was the maximum amounts of clients that SockJS managed. As seen in Figure 4.3 the *minimum* time to receive the message for plain WebSockets was

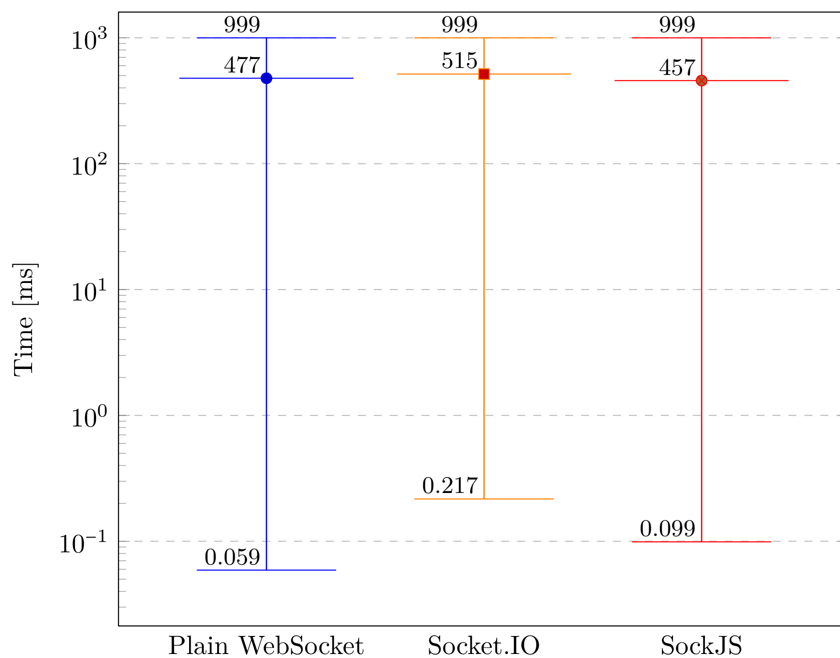


Figure 4.3: Minimum, median and maximum times to receive the message with 7000 clients.

59 µs, while Socket.IO never transmitted the message faster than 217 µs (3.7 times as slow) and SockJS never faster than 99 µs (1.7 times as slow). The *maximum* time to receive the message was just under 1 second for all frameworks. The *median* time was almost the same for all frameworks with times between 457 and 515 milliseconds. In addition to the previously presented parameters, the *standard deviation* was also looked at to get an indication of how the times are spread out from the average, the *standard deviation* was between 288 and 293 for the three frameworks.

4.3 Network traces

To understand why and how the framework may differ, the network traces for 1 client receiving 1 message was recorded and saved with the network analyzer Wireshark [17]. The amount of data needed for the message to be sent differs. As seen in Table 4.1 the total amount of *Reassembled TCP Segments* for plain WebSockets is 75963 bytes. While for Socket.IO it is 96262 bytes and for SockJS 96012 bytes. Also the number of transmissions is an indication of the amount of overhead in the frameworks, plain WebSocket uses 5 transmissions, while Socket.IO requires 13 and SockJS requires 8.

Framework	Reassembled TCP Segments	Number of transmissions
Plain WebSocket	75963	5
Socket.IO	96262	13
SockJS	96012	8

Table 4.1: Network traces for one client receiving one message.

4.4 Memory usage on server

With only 1 client, the server required roughly 40 MB of memory no matter the framework. With 1000 clients, plain WebSockets required roughly 80 MB while Socket.IO required close to 200 MB. SockJS required 94 MB. With 10000 clients plain WebSockets required 210 MB of memory while Socket.IO required almost 2 GB. SockJS topped out with 7000 clients requiring 241 MB of memory. See Figure 4.4.

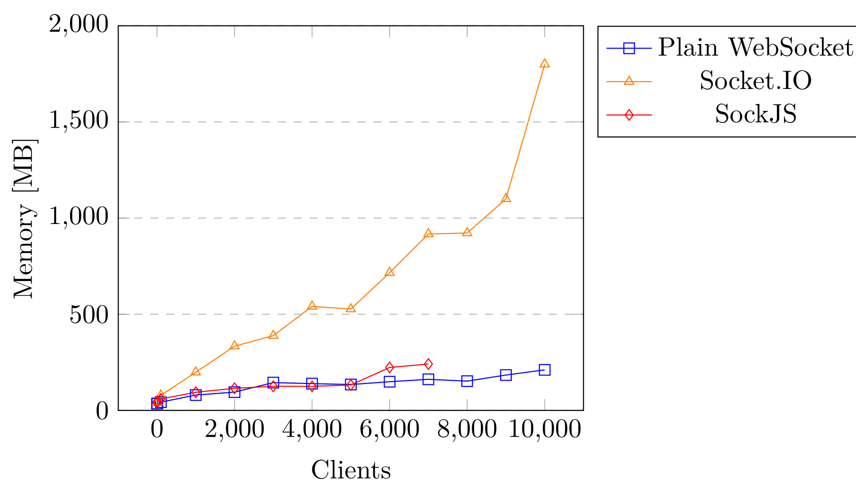
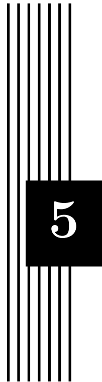


Figure 4.4: Memory usage with different levels of clients.



5 Discussion

5.1 Results

In general the results have been as expected, with differences between all frameworks, but I did not except that big of a difference.

5.1.1 Time to establish a connection

As expected plain WebSockets are faster than the frameworks Socket.IO and SockJS, up to 3 times faster. This is expected as plain WebSocket follow the protocol specification while the frameworks Socket.IO and SockJS have additional overhead in the form of sending multiple requests as well as messages just to establish the connection.

Socket.IO uses three GET requests over HTTP and two messages over WebSocket when establishing a connection. First it establishes a XHR polling connection and then it tries to establish a WebSocket connection which gets verified. The overhead is seen as more transmissions required in Table 4.1.

SockJS first sends a regular HTTP GET request and the server responds with a JSON object. Then the WebSocket request is sent along with data in the URL and the server responds. At last, a message is sent from the server to the client. Again the overhead is seen in Table 4.1. All of this is done to only establish a connection and without the server sending the actual Blixtvakt message.

With up to 2000 parallel clients there are significant performance gains to be made by using plain WebSockets. With 3000 and more clients all frameworks are almost identical. I believe this to be because of the gigabit network the test bench was connected to. The message sent from the server to the clients is large (76 KB), and the amount of traffic received on the workstation showed around 110 MiB/s which amounts to just under 1 Gbit/s. It seems like the gigabit ethernet limits the server and clients from establishing connections any faster while the server already sends messages to the connected clients.

A thing to note is that SockJS did not manage more than 7000 connecting clients at all. I did not manage to find out why. Apart from the fact that the network utilization was reduced to zero after a few seconds and the workstation and server application stopped responding without throwing any visible errors.

5.1.2 Time to receive a message upon connection

Even with only 1 client connecting significant performance differences are seen. Socket.IO being 21 times (!) slower and SockJS being 3 times slower than plain WebSockets. With 3000 clients and more, plain WebSockets and Socket.IO seems to be comparable and limited by the network as discussed in subsection 5.1.1.

A very interesting thing is that SockJS seems to outperform both plain WebSockets as well as Socket.IO with up to 7000 clients, but also that it does not manage more than 7000 clients at all. The curve indicates that SockJS does not scale well. This supports the idea that SockJS may be better with low concurrency applications. But also that it does not scale well and is not a reasonable framework in high concurrency applications, such as Zenon's Blixtvakt. This is similar to the conclusion made by Wang et al. [9] where they concluded that some frameworks may have better throughput while other frameworks can be better with parallel workload, that is throughput or response time is not equal to good performance with high levels of concurrency.

With 7000 clients, the median is between 457 to 515 milliseconds for the frameworks, while the average is between 481 and 493 milliseconds. It would seem that the time taken to receive the message with 7000 clients is nearly always around 450 to 500 milliseconds.

The standard deviation with values of 288 to 293 for all three frameworks indicates that the time to receive the message is highly uncertain and differs a lot with each message received.

The difference in minimum time proves that plain WebSocket can be (and sometimes is) much faster than the other two frameworks.

5.1.3 Memory usage on the server

As seen in Figure 4.4 there are significant differences in the amount of memory required to serve all clients connecting. SockJS being comparable with plain WebSockets with a low memory requirement that scales well. The memory requirement for Socket.IO is a great deal higher and does not look like it scales well with high concurrency. With 5.000 clients Socket.IO requires just over 500 MB of memory, while at 10000 clients, the requirement jumps up to almost 2 GB.

5.2 Method

My setup may have been not that representative of the configuration that Zenon has. I was running the tests on on my private local network while Zenon's server is running on the Amazon Web Services platform. Also in my test I requested the connections in parallel, while Zenon may have users connecting in a period of a couple of seconds or even minutes. Thus my workload was heavier than the actual workload at Blixtvakt.

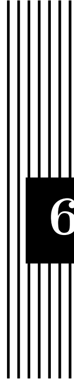
It seemed that the gigabit ethernet that the workstation and server was connected to was limiting the throughput in the network. And thus possibly the results achieved. By having access to a higher bandwidth network, the network would not have been limiting the server from responding to the clients. And the results may have been more accurate and representative of the situation with Blixtvakt. Although I do believe that the trends would still be the same.

To get more accurate results, I could have configured a test bench and network to allow for higher throughput. Unfortunately I did not have access to equipment that suffice 10 Gbit/s. Although it made me realize that the problems Zenon are having may be due to the infrastructure and/or network configuration. That it in fact, may not have anything to do with the current communication framework that they use. This is something that should be considered and further researched.

5.3 The work in a wider context

By choosing a framework with appropriate amount of abstraction two main benefits can be achieved. Less development time, and thus also cost. As well as optimized applications that are faster, more stable and have a lower power consumption. In addition - being able to manage the workload with less resources, for example one server. Contrary to requiring multiple servers to be able to cope with the occasional high surge of users, while most of the time running at idle-level workload, which has been shown to be highly inefficient and still consume a lot of power [18].

This study is useful for Zenon in their continued work to improve Blixtvakt and solving the performance issues they have.



6 Conclusion

I have shown that there exists significant performance differences in JavaScript WebSocket frameworks. Mainly due to the overhead and abstraction that the programmer may or may not want. The following points are the main differences I have discovered in the three solutions researched:

- Plain WebSocket - linear response time and memory required on server.
- Socket.IO - exponential memory requirement on server.
- SockJS - exponential time to receive message.

I have shown that Zenon's current framework have issues with high concurrency levels and that there exists alternatives that are more scalable and may solve their problems regarding high levels of concurrency. Plain WebSockets seems to be the most linear, scalable and most performant solution by far, as guessed. If Zenon does need or want the advantages of the Socket.IO framework they should be prepared to equip their server with a lot of memory.

Even if there exists substantial performance and scalability differences among communication frameworks. It is difficult to say if the choice of frameworks really does matter when dealing with up to 100.000 concurrent users. I suspect that other factors such as the maximum network throughput may the the bottleneck instead. This should be considered and further research into server load and network utilization is needed. Since Zenon deal with large messages one could question if it is possible to reach higher concurrency levels by splitting the message into smaller parts, or if compression is an alternative?

Bibliography

[1] J. A. Hoxmeier and C. DiCesare. "System Response Time and User Satisfaction: An Experimental Study of Browser-based Applications." In: *Proceedings of the Association of Information Systems Americas Conference* (Jan. 2000).

[2] *SockJS GitHub Repository.* URL: `https://github.com/sockjs/sockjs-client/releases`. (accessed: 07.04.2020).

[3] *Socket.IO GitHub Repository.* URL: `https://github.com/socketio/socket.io/releases?after=2.0.0`. (accessed: 30.03.2020).

[4] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach, 7th Edition.* Pearson, 2017, pp. 126, 261. ISBN: 978-1-292-15359-9.

[5] I. Fette and A. Melnikov. *The WebSocket Protocol.* 2011. URL: `https://tools.ietf.org/html/rfc6455`. (accessed: 02.04.2020).

[6] *AJAX.* URL: `https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX`. (accessed: 09.06.2020).

[7] *Node.js.* URL: `https://nodejs.org/en/`. (accessed: 09.04.2020).

[8] *V8 JavaScipt Engine.* URL: `https://v8.dev/`. (accessed: 14.04.2020).

[9] Y. Wang, L. Huang, X. Liu, T. Sun, and K. Lei. "Performance Comparison and Evaluation of WebSocket Frameworks: Netty, Undertow, Vert.x, Grizzly and Jetty." In: *1st IEEE International Conference on Hot Information-Centric Networking* (2018), pp. 13–17. DOI: `10.1109/HOTICN.2018.8605989`.

[10] D. Skvorc, M. Horvat, and S. Srbljic. "Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams." In: *37th International Convention on Information and Communication Technology, Electronics and Microelectronics* (2014), pp. 1003–1008. DOI: `10.1109/MIPRO.2014.6859715`.

[11] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach." In: *IEEE Transactions On Parallel And Distributed Systems* 13 (2002), pp. 80–96.

[12] C. W. Smullen and S. A. Smullen. "AJAX Application Server Performance." In: *Proceedings 2007 IEEE SoutheastCon* (2007), pp. 154–158. DOI: `10.1109/SECON.2007.342873`.

[13] *Can I use - WebSockets.* URL: `https://caniuse.com/#feat=websockets`. (accessed: 29.04.2020).

[14] A. Rahmatulloh, I. Darmawan, and R. Gunawan. "Performance Analysis of Data Transmission on WebSocket for Real-time Communication." In: *16th International Conference on Quality in Research (QIR): International Symposium on Electrical and Computer Engineering* (2019), pp. 1–5. DOI: 10.1109/QIR.2019.8898135.

[15] WK. Chang and SK. Hon. *Evaluating the Performance of a Web Site via Queuing Theory*. Springer Berlin Heidelberg, 2002, pp. 63–72. ISBN: 978-3-540-47984-0. DOI: 10.1007/3-540-47984-8_10.

[16] *process.hrtime*. URL: https://nodejs.org/api/process.html#process_process_hrtime_time. (accessed: 14.04.2020).

[17] *Wireshark - A Network Protocol Analyzer*. URL: https://www.wireshark.org/. (accessed: 14.04.2020).

[18] L. A. Barroso and U. Hölzle. "The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition." In: *Morgan & Claypool Publishers* (2018), pp. 107–108. DOI: 10.2200/S00874ED3V01Y201809CAC046.