

Міністерство освіти і науки України  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВО-МОГИЛЯНСЬКА АКАДЕМІЯ»  
Кафедра мережних технологій факультету інформатики

“Introducing real-time boundless data with websockets”

Текстова частина до курсової роботи  
за спеціальністю „Комп’ютерні науки та інформаційні технології”-122

Керівник курсової роботи:  
док.техн.наук, доц. Глибовець А.М.

(підпис)

“ \_ ” \_\_\_\_\_ 2020 року

Виконала: студентка КНІТ-4  
Діденко В. О.

(підпис)

“ \_ ” \_\_\_\_\_ 2020 року

Київ 2020

Ministry of Education and Science of Ukraine  
NATIONAL UNIVERSITY OF "KYIV-MOHYLA ACADEMY"  
Network Technologies Department of the Faculty of Informatics

APPROVED

Head of the Network Technologies Department  
associate professor, doctor of mathematics

\_\_\_\_\_ G.I. Malaschonok  
(signature)

“ \_\_\_\_ ” \_\_\_\_\_ 2019 year.

INDIVIDUAL TASK

For the course work

For 4-year Bachelor student of the Faculty of Informatics

TOPIC: Introducing real-time boundless data with websockets

Output data:

Text part content of coursework:

Individual task

Calendar plan

Abstract

Introduction

Section 1: Response time performance

Section 2: Choosing the data transfer method

Section 3: Displaying the large data set

Conclusion

References

Issue date “ \_\_\_\_ ” \_\_\_\_\_ 2019 year

Supervisor \_\_\_\_\_(signature)

Task received \_\_\_\_\_(signature)

## CALENDAR PLAN

**Theme:** Introducing real-time boundless data with websockets

#	Stage Name	Deadline	Note
1	Acquiring course work topic	01.10.2019	
2	Finding appropriate literature	10.11.2019	
3	Exploring smooth navigation techniques	10.12.2019	
4	Exploring Realtime	27.12.2019	
5	Exploring data transfer methods	27.01.2020	
6	Researching response time measurement	08.02.2020	
7	Researching WebSocket performance	15.02.2020	
8	Researching Long-polling performance	22.02.2020	
9	Researching SSE performance	29.02.2020	
10	Application implementation	27.03.2020	
11	Outline of the coursework	27.03.2020	
12	Coursework analysis with the Supervisor	28.03.2020	
13	Coursework main section and conclusion	29.03.2020	
14	Coursework analysis with the Supervisor	18.04.2020	
15	Coursework improvement	18.04.2020	
16	Creation of the presentation	18.04.2020	
17	Handing in the coursework	19.04.2020	

Student: Didenko V.O.

“ \_\_\_\_\_ ”

Supervisor: Glybovets A.M.

“ \_\_\_\_\_ ”

# TABLE OF CONTENTS

- TABLE OF FIGURES ..... 4
- ABSTRACT ..... 5
- INTRODUCTION ..... 6
- Section 1: Response time performance..... 8**
  - 1.1 Response time - what is the budget ..... 8**
  - 1.2 Meeting the time constraint ..... 11**
- Section 2: Choosing the data transfer method ..... 14**
  - 2.1 Long Polling..... 14**
  - 2.2 Server sent events ..... 16**
  - 2.3 WebSockets ..... 19**
  - 2.4 Implementation and performance comparison ..... 21**
- Section 3: Displaying the large data set ..... 45**
  - 3.1 Data preloading ..... 46**
  - 3.2 Load balancing ..... 47**
  - 3.3 Smooth rendering ..... 49**
- CONCLUSION ..... 51
- RESOURCES ..... 53

## TABLE OF FIGURES

Figure 1: Calculating invoked event and repaint time .....	12
Figure 2: Calculating affordable data transfer time .....	13
Figure 3: Long Polling general flow .....	15
Figure 4: Server Sent Events general flow .....	16
Figure 5: WebSockets general flow.....	20
Figure 6: Class diagram of the server-side for data transfer method testing.....	23
Figure 7: Diagram of the Program class.....	24
Figure 8: Diagram of the Startup class .....	27
Figure 9: SignalR Hub concept .....	30
Figure 10: Diagram of the TestsHub class .....	31
Figure 11: Setting the data transfer method test parameters.....	41
Figure 12: Running the test for all three data transfer methods.....	42
Figure 13: Performance time test results for 100 calls .....	43
Figure 14: Performance time test results for 1000 calls .....	43
Figure 15: Performance time test results for 10000 calls.....	44
Figure 16: Displaying the large data set on a data table.....	45
Figure 17: Preloading data depending on scroll direction.....	46
Figure 18: Executing the processes during idle browser time.....	48
Figure 19: Optimized number of draws by painting the result in the next frame.....	49
Figure 20: Rendering only the elements that are in view point .....	50
Figure 21: Updating only the results.....	50

## ABSTRACT

Loading and displaying a large data set with minimal delay has always been a challenging task. With the increase of data set size, the loading time before the data is displayed grows and the user experience suffers. In this research work the aim is to load and display a large data set within the time limit required for the user to perceive the response as instant and to provide smooth navigation and a pleasant user experience. Based on multiple research the required response time limit was determined to be 0.1 second. Based on this time constraint the time that can be spent for each process was calculated and after an empirical research the data transfer method for loading the data and keeping it real-time was chosen to be WebSockets. With WebSockets as the data transfer technology the large data set was loaded and displayed on a sample data table under 100 milliseconds and a smooth user experience was achieved.

## INTRODUCTION

When it comes to working with large data sets, providing a swift and pleasant user experience remains a challenge. While loading a large amount of data is one task, displaying it quickly to the user is yet another: with the growth of data set size, the total waiting time increases, therefore leading to a poor user experience.

In this work the aim is to provide a smooth and pleasant user experience while working with a large amount of real-time data: to quickly load and show the data with minimal delay to the user, providing the feeling that the user is working with the data in real-time. In order to solve the problem of poor user experience, it is important to define the response time budget, in other words the timeframe within which a user interaction should complete in order to give the sensation to users that they are directly manipulating elements in the UI and directly working with the data. This includes understanding the key stages that it takes to respond to a user's input, setting the main focus on the data transfer stage. This course work shows how the data transfer method that is chosen can help to achieve real time data experience in web-applications. This course work focuses on the performance factors of three data transfer technologies: Long Polling, Server sent events and WebSockets, and among the three, points out the approach which is best, from the performance and specification perspective, for the job at hand.

The comparison of the data transfer methods is done in three stages. At first an overview of each data transfer technology is given along with its

analysis in order to identify the performance factors. This is followed by an empirical research which implies implementing each data transfer method on a sample server to observe and compare their performance. The test results are then gathered and compared and a conclusion is drawn as to which technology is the most performance efficient.

The results of this work revealed that the choice of data transfer technology does matter when it comes to response time performance. WebSockets and Server sent events were measured to be the most performance efficient of the compared technologies in the experimental conditions used in this study, giving preference to WebSockets as the final choice. With WebSockets as the selected data transfer method and by balancing the CPU usage between data loading, calculations, and the rendering stages, it is then demonstrated on the example of a data table how the real-time effect is reached on a large data set by displaying the data to the user under a tenth of a second.



## **Section 1: Response time performance**

It is important to determine how fast the data needs to be displayed in order to provide a real-time effect. The answer lies in setting the appropriate response time limit. [1,2,3] Next, the established time frame needs to be distributed among the tasks that take place under the hood to provide a reaction to the user's action in time.

### **1.1 Response time - what is the budget**

According to multiple research and studies, when it comes to web-based applications or any application for that matter, there are three main time limits within which the application should respond to user input. These response time constraints are defined as follows: 0.1 second, 1 second, and 10 seconds. An interesting point to consider is that although these guidelines were introduced over 50 years ago, they remain accurate to this day and are unlikely to change since they are based on human perceptive abilities alone. [1,2,3]

The first time limit is 100 milliseconds. Based on how our eyes operate, it is determined that any response given in this time is about the limit required for a reaction to being perceived as instant. Remaining within this timeframe provides the user with the sensation of directly making things happen on the screen which is exactly the aim of real-time web-applications: users should feel as though they are directly manipulating the data, the system responds in a blink of an eye providing a smooth, without delay workflow. A practical application of such can be demonstrated on the example of a data table with multiple columns and rows. When a cell in the table is selected, the feedback

about its selection, such as highlighting, should be given under 100 milliseconds starting from the moment it was selected. This is also the response time that should be aimed at when sorting values of a column in ascending or descending order. Such a response time makes people feel that they are highlighting elements in the table and sorting it, not that they are ordering the computer to do it for them. [1,2,3]

The second limit is 1 second. In contrast to the fastest limit, with this response time, people feel that the application is reacting to what they did as opposed to feeling that they are making things happen. Although it is possible to freely use the application, the delay is noticeable and the instantaneous direct effect is lost. Returning to the data table example, if it is not possible to sort the table based on the selected column within 100 milliseconds, it definitely needs to be sorted in 1000 milliseconds. Otherwise, the user interface is perceived as being slow, taking away the flow sensation while navigating the system. For delays over this limit, feedback should be provided indicating that the process is in progress, for example displaying a preloader. This leads to the last response time limit. [1,2,3]

The third response time constraint is 10 seconds and it defines the limit until when the attention on the task can still be kept. This is how long people can wait and still stay in the flow of the interaction while remembering what they were in the process of doing. Nevertheless, it is without a doubt that once it takes more than 1 second for an application to respond to an input, people get brought down by the weight and feel they can't use the application freely. [1,2,3]

Bringing all the above together, it can be concluded that to provide the user with the best experience when working with a large data set, the aim should be to provide a response within a tenth of a second. In most situations when working with large data sets, not the whole data set is loaded fully at once into the browser, instead, the data is loaded by parts on demand during runtime. The reason for this recommendation is because first of all, for users to start working with the data, if there is no lazy loading implemented, they will be required to wait for quite some time until it is all loaded and ready to work with. Moreover, for some machines, it may be too much to handle a large data set at one go. This could lead to lagging or even cause the browser to crash because of the overload. On the other hand, when loading data by parts on-demand, it is desired to load and display the data within 100 milliseconds to ensure a smooth workflow and provide the real-time effect. [1,2,3,8,9,10,11,12]

## 1.2 Meeting the time constraint

As stated earlier, it is needed to meet the timeframe of 0.1 seconds. The processes that need to take place to respond to an event can be pointed out as the following: event handling, request composing, the handshake process and connection establishment, data sending, server query, response creation, receiving the response, displaying the result.

The processes that take place when the screen is updated and repainted by the browser are encapsulated in a frame and the rate with which each frame is generated is used to measure the interface responsiveness. To maintain a responsive user interface and provide smooth interface interactions, a high and steady frame rate must be kept. It is known that for a user interface to be perceived as responsive, the frame rate of 60 frames per second should be aimed for. In addition, the frames need to be kept equal in length to ensure a steady frame rate, which in return helps to steer clear of jaggy and rigid animations. [4,7,8,9,10,11,12]

Since the aim is to have a rate of 60 frames per second, it can be calculated that each frame needs to take up about 16.7 milliseconds:  $1000 \text{ (milliseconds)} / 60 \text{ (frames per second)} \approx 16.7 \text{ (milliseconds)}$ . [4,7,8,9,10,11,12] Taking into consideration that each frame can take up about this much time and the goal is to provide a response within 0.1 seconds, it needs to be calculated how much milliseconds can be afforded to spend on each process (that needs to take place to respond to the action) while remaining within the time constraint.

This is portrayed in the example of what takes place when the data table is scrolled. The invoked event takes up the first frame which corresponds to 16.7 milliseconds. The last part is the rendering stage when the result is shown to the user. Suppose this part takes up at most two frames which adds up to 16.7 (milliseconds) + 16.7 (milliseconds)  $\approx$  33.4 (milliseconds). The first frame is for recalculations and the second frame is requested for repainting the screen which will help to keep the frame rate steady. Therefore, in the first frame, the `requestAnimationFrame` function is called to draw the resulting elements in the next frame. [48,49,50,51] So, in total, the time spent is 16.7 (milliseconds) + 33.4 (milliseconds)  $\approx$  50 (milliseconds). This leaves us with the following time span: 100 (milliseconds) - 50 (milliseconds)  $\approx$  50 (milliseconds). This is illustrated in Figure 1: Calculating invoked event and repaint time.

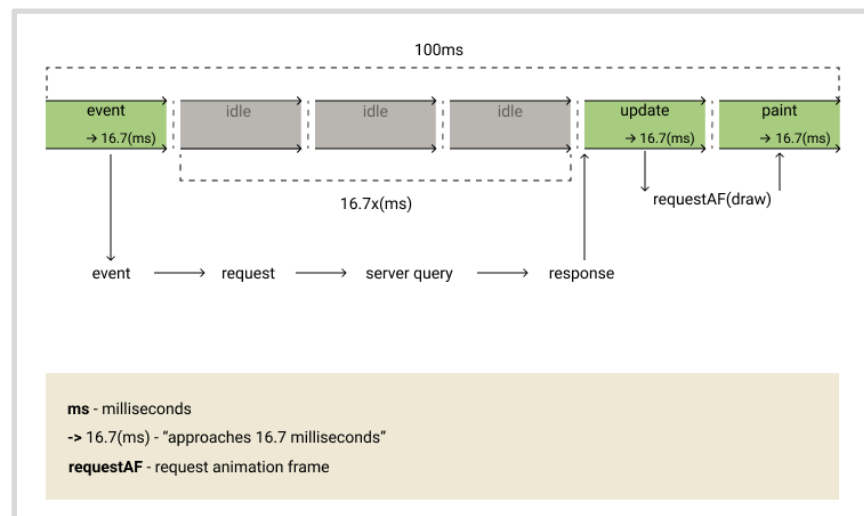


Figure 1: Calculating invoked event and repaint time

For the processes that happen between the event frame and the rendering stage, it can be afforded to spend 16.7x milliseconds, where x is the number of idle frames used. [1,2,3,4] Since the remaining time is 50

milliseconds, it means that  $x$  has to be less than or equal to 3 frames:  $16.7$  (milliseconds) \*  $3$  (frames)  $\approx 50$  (milliseconds). In other words, a connection needs to be established with the server, the server needs to respond to the request, and the received data needs to be sent back to the client within this time.

In this work, the main focus is set on the data transfer layer. Therefore, it is assumed that the server is a black box - it is known that there is no hold up from the server: the server responds instantly when data is requested from the data source and the total time spent is minimal and approaches 0 milliseconds. This leaves the connection and data transfer stages as the main point of focus (illustrated in Figure 2: Calculating affordable data transfer time). Therefore, it is vital to determine the data transfer method that is most suitable for the job at hand to minimize the waiting time.

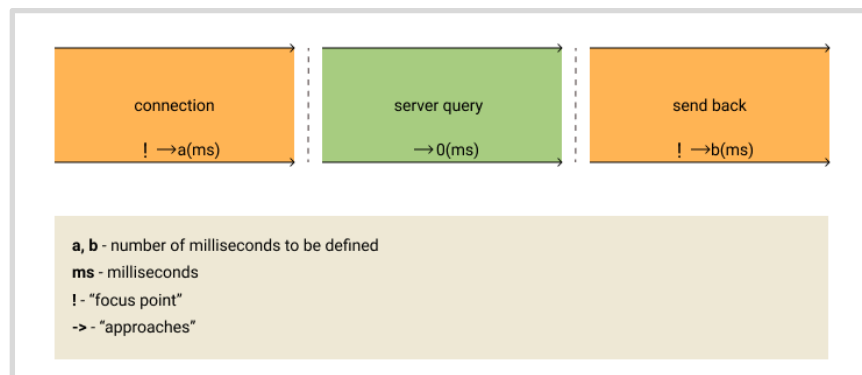


Figure 2: Calculating affordable data transfer time

## **Section 2: Choosing the data transfer method**

In this work, the following data transfer methods that solve the problem of real-time data updates from the server are selected as possible candidates: Long Polling, Web Sockets, and Server-Sent Events. Understanding the difference between them is essential in order to select the technology with which performance can be gained. The next section is dedicated to the analysis and comparison of the mentioned technologies.

Data transfer method presentation consists of: its definition and what it implies, illustration of the general flow, protocol handshake example, and portraying it on the task at hand. Assuming that the technology is being used, it is identified how it affects the response time performance.

### **2.1 Long Polling**

Long polling is considered to be one of the easiest ways to simulate a continual connection with the server. In Long Polling, a connection is established between the client and the server when the client sends a request to the server and the server doesn't close the connection until it has data to send back. When data is available, the server responds to the request by sending the data, which in return closes the connection, after which, the client opens a new request. [13,14,26,27,28,29,30,31,32,33,34] This flow is illustrated in Figure 3: Long Polling general flow.

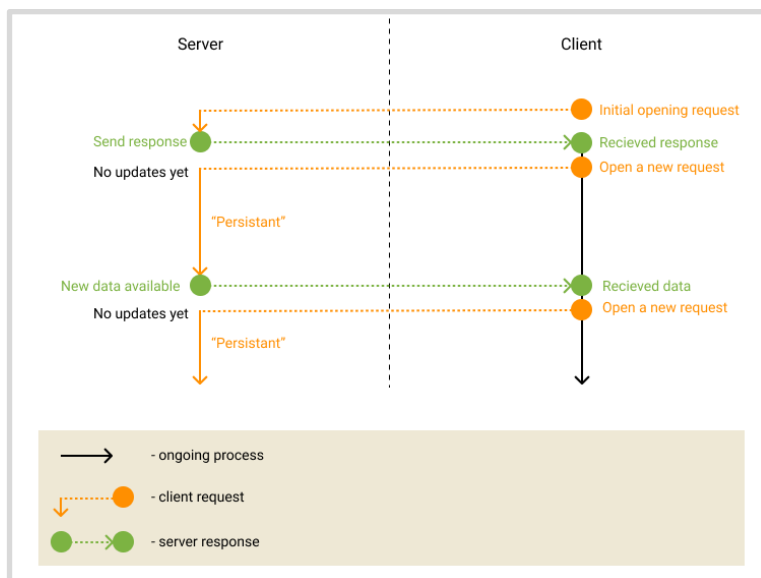


Figure 3: Long Polling general flow

Although Long Polling provides a real-time data update from the server, it is not a persistent connection, it only simulates it while in reality there are a lot of requests to the server. Moreover, since the connection is closed as soon as the server provides a data update, it will be required to establish a new connection with the server with every single new request. This affects the amount of traffic and message latency. In addition, with Long Polling there could be an occasional timeout right after which the client initiates a new request. It is also important to point out that Long Polling is one way: there is no real way that the server could tell the client that new data is available the moment it gets it, the server can only respond to an open request. With this data transfer technology, a server push can only be simulated: behind the scenes, there is always a request for data. [13,14,26,27,28,29,30,31,32,33,34,35,36,37] An example implementation of Long Polling usage on a sample server is provided in the [2.4 section](#) of this work.



## 2.2 Server sent events

Server sent events is a data transfer protocol that provides a way for the server to send updates to the client, meaning that, in contrast to long polling, it is no longer needed to open new requests to the server to check if new data is available every time a response is received. This data transfer method can be viewed as a subscription: the client subscribes to receive updates from the server and the connection remains open. It is still one way though: in contrast to long polling, where a client pull was possible, with Server Sent Events a server push is provided instead. The described flow of this data transfer method is illustrated on Figure 4: Server Sent Events general flow. [13,14,15,16,17,18,25,26,27,28,29,30,31,32,33,34,35,36,37]

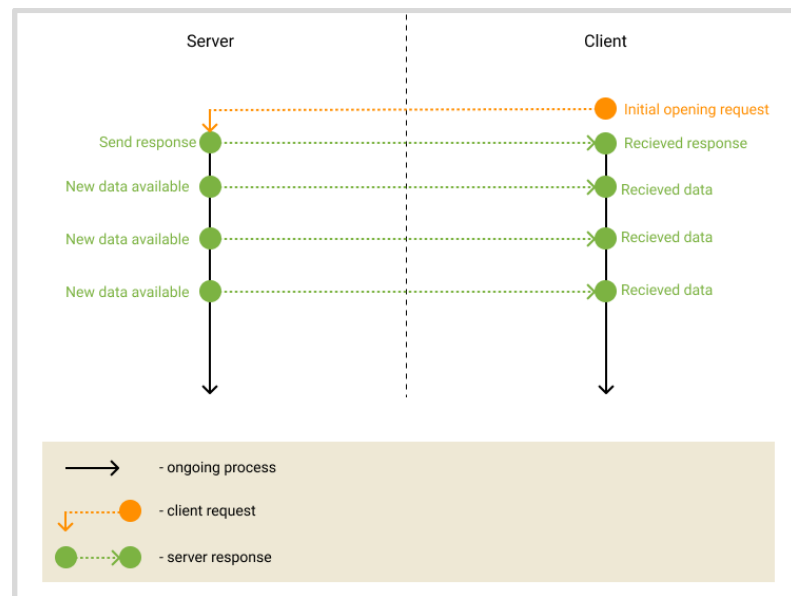


Figure 4: Server Sent Events general flow

This is provided by introducing two components: a new EventSource interface in the browser and an event-stream data format. The EventSource interface allows the client to receive real-time notifications from the server in

the form of DOM events and the event-stream data format's purpose is to deliver these updates. [14,15,16,17,18]

The Event Stream Protocol represents a streaming HTTP response where a regular HTTP request is initiated by the client and a response is sent from the server containing a custom "text/event-stream" content-type. Once the connection is established, the server can stream UTF-encoded text-based read-only event data to the client. [14,15,16,17,18] Below is an example of the described process:

```
1  <= Request:
2  GET /tests/?id=m62tuAPjwul7ptw0lVzu6w HTTP/1.1
3  Host: olap.flexmonster.com:5013
4  Connection: keep-alive
5  Accept: text/event-stream
6
7  => Response:
8  HTTP/1.1 200 OK
9  Content-Type: text/event-stream
10 Server: Kestrel
11 Transfer-Encoding: chunked
```

On line 2 the client connection is initiated via the EventSource interface. The server response is provided on line 8 with the content-type specified as "text/event-stream". [14,15,16,17,18]

So, in comparison to Long Polling, even though there is no client pull, the client automatically receives updates from the server and therefore time is saved by not having to reopen the connection to receive new data every time. [25,26,27,28,29,30,31,32,33,34,35,36,37] An example implementation

of Server Sent Events on a sample server is provided in the [2.4 section](#) of this work.

## 2.3 WebSockets

A websocket represents a persistent connection between the client and the server. In contrast to Server Sent Events and Long Polling which follow HTTP, this data transfer technology introduces its own protocol for data exchange known as the WebSocket protocol. WebSockets provide a channel for the client and the server to communicate, which operates over HTTP via a single TCP/IP socket connection, and is bidirectional as well as full-duplex, enabling a two-way simultaneous data exchange that means both the client and the server can send messages independently and at the same time. [19,20,21,22,23,24,26,33] Figure 5: WebSockets general flow illustrates the described method.

The WebSockets Protocol begins by the initial opening request - handshake, after which the client and the server can start exchanging messages. [19,20,21,22,23,24,26,33] Below is an example of how the connection is established:

```
1  <= Request:
2  GET ws://olap.flexmonster.com:5011/tests/?id=fDdS8wO9A-UZFsMm1EaKDA
3  HTTP/1.1
4  Host: olap.flexmonster.com:5011
5  Connection: Upgrade
6  Upgrade: websocket
7  Origin: http://cdn.flexmonster.com
8  Sec-WebSocket-Version: 13
9  Sec-WebSocket-Key: wLL3bL6eK/BOfLTgFGhGlA==
10 Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
11
12 => Response:
13 HTTP/1.1 101 Switching Protocols
```

```
14 | Connection: Upgrade
15 | Server: Kestrel
16 | Upgrade: websocket
17 | Sec-WebSocket-Accept: /ZY8F8M6BcswYnOkE4Bf15H01LU=
```

So, in comparison to Server Sent Events, with WebSockets not only is there a server push, but also the client can send messages to the server, without having to establish a new connection. Moreover, both the server and the client can send messages to each other at the same time. [25,26,27,28,29,30,31,32,33,34,35,36,37] An example implementation of WebSockets on a sample server is provided in the [2.4 section](#) of this work.

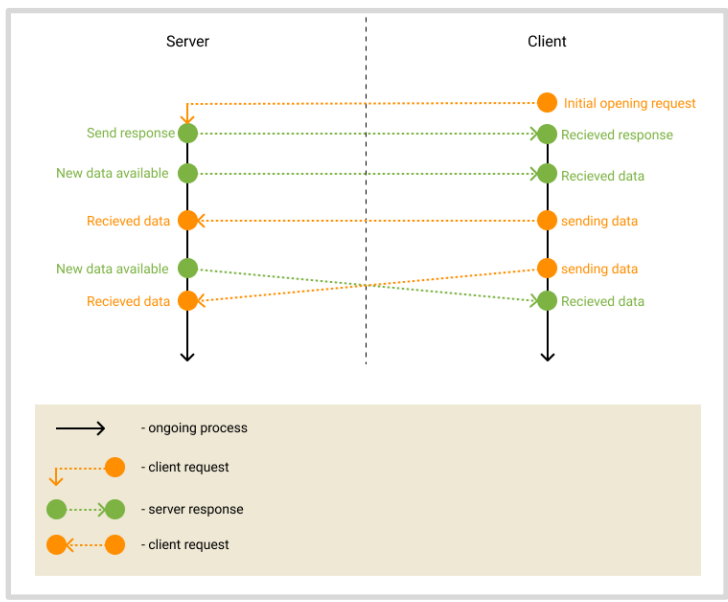


Figure 5: WebSockets general flow

## 2.4 Implementation and performance comparison

An application was created using .NET Core and SignalR for testing and visually comparing the performance of Long Polling, WebSockets, and Server Sent Events to discover which data transfer approach is the most performance efficient of the three. SignalR is an open-source library which is aimed at making real-time web functionality simpler to implement. [38,39]

To ensure clarity, transparency and validity in the way the data transfer technologies' performance time is compared, the following precautions were made. To start, on the implemented server three separate servers are started for each data transfer method, each running on a separate port. Moreover, each server uses the same implementation: the same controller and the same request processor. In return, the client streams equally sized requests and the application is written in such a way that it is either possible to stream the requests to all three ports simultaneously, allowing to compare the performance time for all data transfer methods at once, or to a specific one, enabling to test a certain data transfer technology alone. The user interface allows setting the size of the request and how many of them will be made for a particular test. The performance time for each data transfer technology is calculated starting from the command for establishing connection for the first request and ending when the last request is received. To ensure fairness during the performance test when the performance time for all three data transfer methods is compared, each data transfer method gets the same randomly generated data block of type string for the corresponding call, meaning the current data block that needs to be transferred during a specific call is the same for Server Sent Events, Long Polling, and WebSockets, if

the test is run for all three methods at once. In addition, no caching is used. Keeping this in mind, in the next paragraphs of this section, the application's structure and implementation is described in more detail.

The application for testing the data transfer methods has the following structure, where the client-side is provided in the client folder and the server-side is represented by the Hubs folder, Program.cs and Startup.cs files:

```

1 | .
2 | └─ client
3 |   └─ dist
4 |     └─ browser
5 |       └─ signalr.js
6 |         └─ signalr.min.js
7 |   └─ index.html
8 |   └─ style.css
9 |   └─ tests.js
10| └─ Hubs
11|   └─ TestRequest.cs
12|   └─ TestResponse.cs
13|   └─ TestsHub.cs
14| └─ Program.cs
15| └─ Startup.cs

```

Below is a class diagram of the server-side structure which consists of the Program, Startup, TestsHub, TestRequest, and TestResponse classes (see Figure 6: Class diagram of the server-side for data transfer method testing):

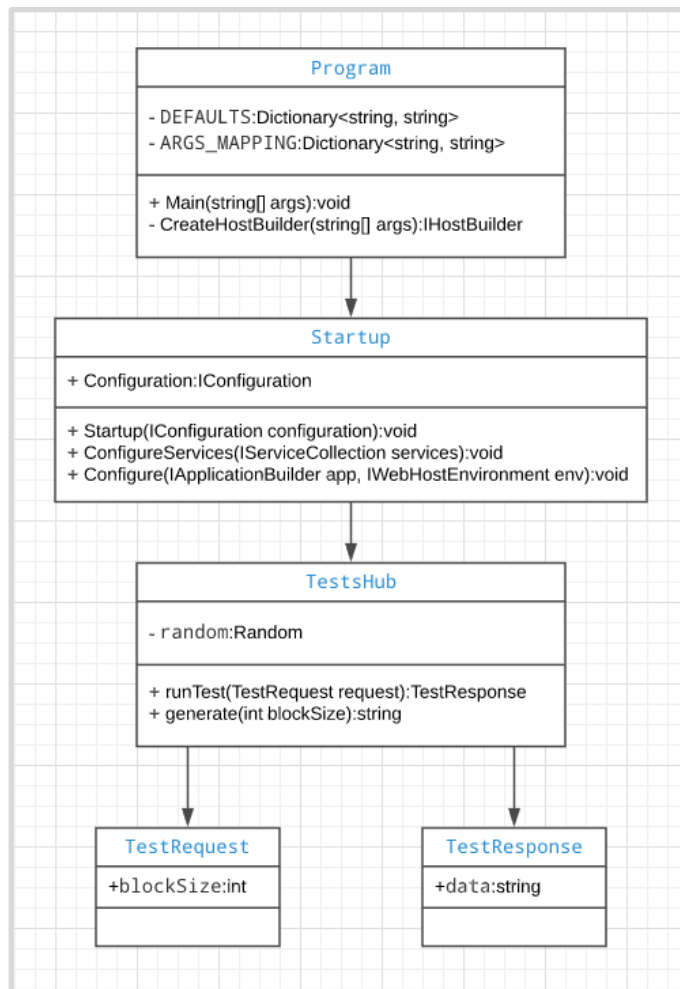


Figure 6: Class diagram of the server-side for data transfer method testing

The Program.cs file contains the Program class which is responsible for configuring the application's infrastructure and is the entry point of the application. A close up view of the Program class part of the diagram is illustrated in Figure 7: Diagram of the Program class. The Program class has the DEFAULTS and ARGS\_MAPPING private properties and contains the public Main and the private CreateHostBuilder methods.



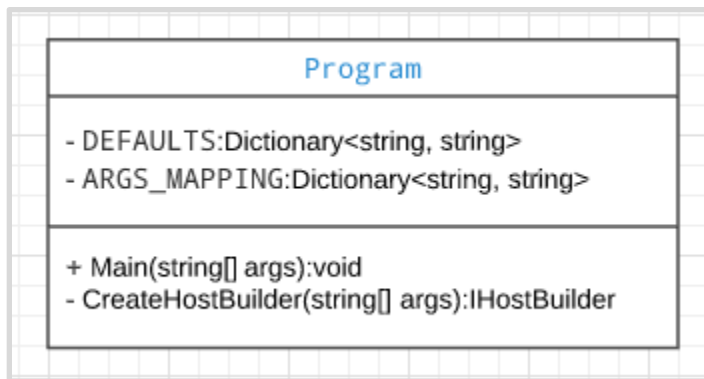


Figure 7: Diagram of the Program class

The DEFAULTS property defines the default input parameters for starting the server: if no parameters are specified in the start server command, by default the server will choose websockets as the data transfer method which will run on port 5000. Below is a code snippet of the described property:

```

1 private static readonly Dictionary<string, string> DEFAULTS =
2   new Dictionary<string, string>
3   {
4     { "transport", "websockets"},
5     { "port", "5000"}
6   };
7

```

The ARGS\_MAPPING property maps the server start command's input parameters to the corresponding variables. The -t command parameter stands for the data transfer type and the -port parameter represents the port on which the server using the specified data transfer protocol will be run. In the code snippet below it is shown how the ARGS\_MAPPING property is defined:

```
1 private static readonly Dictionary<string, string> ARGS_MAPPING =  
2 new Dictionary<string, string>  
3 {  
4     { "-t", "transport" },  
5     { "-port", "port" }  
6 };
```

The Main method is responsible for configuring, building and running the host, which is an object that encapsulates all of the app's resources. Organizing the app's interdependent resources in a single host object provides better application lifetime management. On line 3, in the code snippet of the Main method's implementation which is provided below, the CreateHostBuilder method is called which creates the host builder and provides the configurations to it, afterwards the Build method builds the host object based on the provided builder and configurations, and then the host is run by the Run method:

```
1 public static void Main(string[] args)  
2 {  
3     CreateHostBuilder(args).Build().Run();  
4     System.Threading.Thread.Sleep(-1);  
5 }
```

The next code snippet corresponds to the CreateHostBuilder method which entails the configuration set up and build (lines 3-6), the port set up (line 7), and the creation of the host object builder named hostBuilder (line 8) where the built configurations are applied (line 12) and the defined port is used (line 18). The startup type containing the startup methods of the application to be

used by the host is specified to be of the Startup class type and is shown on line 17. The created host object builder is then returned to be built and run in the Main method [38,39,40,41,42]:

```
1 public static IHostBuilder CreateHostBuilder(string[] args)
2 {
3     var configBuilder = new ConfigurationBuilder()
4         .AddInMemoryCollection(DEFAULTS)
5         .AddCommandLine(args, Program.ARGES_MAPPING);
6     var configuration = configBuilder.Build();
7     var port = configuration.GetValue<string>("port");
8     var hostBuilder = Host.CreateDefaultBuilder(args)
9         .ConfigureAppConfiguration(
10             (hostingContext, configBuilder)=>
11             {
12                 configBuilder.AddConfiguration(configuration);
13             }
14         )
15         .ConfigureWebHostDefaults(webBuilder =>
16             {
17                 webBuilder.UseStartup<Startup>()
18                     .UseUrls("http://0.0.0.0:" + port);
19             }
20         );
21     return hostBuilder;
22 }
```

The Startup class defined in the Startup.cs file contains the startup methods for the application. A close up view of the Startup class part of the diagram is illustrated in Figure 8: Diagram of the Startup class. The Startup class has the Configuration public property and Startup constructor and contains the following public methods: ConfigureServices and Configure.

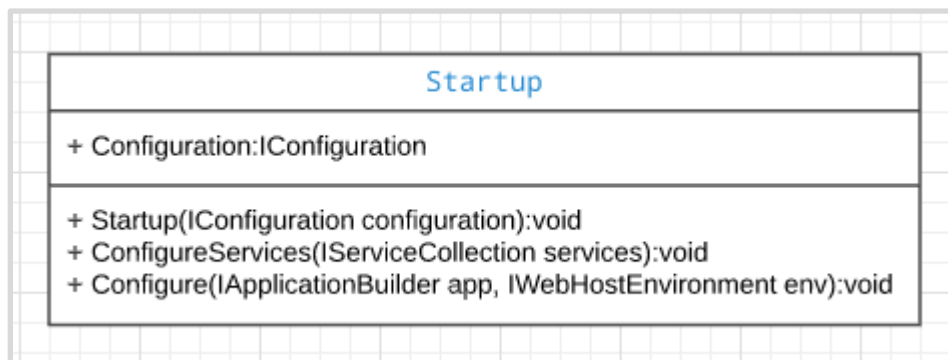


Figure 8: Diagram of the Startup class

Upon initialization, the configurations containing the data transfer type along with other application configurations are set via the defined constructor below:

```

1 | public Startup(IConfiguration configuration)
2 | {
3 |     Configuration = configuration;
4 | }
5 | public IConfiguration Configuration { get; }
  
```

The request pipeline which is responsible for handling all the incoming requests is configured in the ConfigureServices method shown in the code snippet below where cross-origin resource sharing is configured for the application:

```

1 | public void ConfigureServices(IServiceCollection services)
2 | {
3 |     services.AddCors(options =>{
4 |         options.AddPolicy("CorsPolicy",
5 |             builder => builder
  
```

```
6         .AllowAnyMethod()
7         .AllowAnyHeader()
8         .AllowCredentials()
9         .SetIsOriginAllowed((host) => true));
10    });
11    services.AddSignalR(hubOptions =>{
12        hubOptions.EnableDetailedErrors = true;
13    });
14 }
```

In the `Configure` method the services defined in `ConfigureServices` are applied. In addition, as shown in the `Configure` method's code snippet below, the data transport type is set based on the provided configurations. On lines 1-2 the transport type value is retrieved from the configurations. Since the application supports default input parameters if none were specified in the start command and the default transport type is set to `websockets`, on the third line the `transportOptions` variable is initialized as `WebSockets`. The `HttpTransportType` enumeration type provides several flag attributes that define which data transport type is used, those being the following: `LongPolling`, `None`, `ServerSentEvents`, and `WebSockets`. [38,39] On lines 4-18 the transport type that is in the configurations is checked via the switch case approach and the `transportOptions` variable's value is updated if needed.

```

1  string transportType =
2  this.Configuration.GetValue<string>("transport");
3  HttpTransportType transportOptions =HttpTransportType.WebSockets;
4  switch (transportType.ToLower())
5  {  case "websockets":
6      case "ws":
7          transportOptions = HttpTransportType.WebSockets;
8          break;
9      case "longpolling":
10     case "lp":
11         transportOptions = HttpTransportType.LongPolling;
12         break;
13     case "serversentevents":
14     case "sse":
15         transportOptions = HttpTransportType.ServerSentEvents;
16         break;
17 }

```

Next, the application's endpoints are configured via the `UseEndpoints` method. In the created application only one endpoint is needed - the page displaying the performance tests input parameters and results, and this endpoint is defined as `/tests`. This is demonstrated in the code snippet below:

```

1  app.UseEndpoints(endpoints =>{
2      endpoints.MapHub<TestsHub>("/tests", options =>
3          {options.Transports = transportOptions;});});

```

The SignalR library uses a high-level pipeline to enable the client and the server to call methods on each other and this pipeline is called a hub. [38,39,40,41,42] This concept is illustrated on the diagram in Figure 9: SignalR Hub concept below:

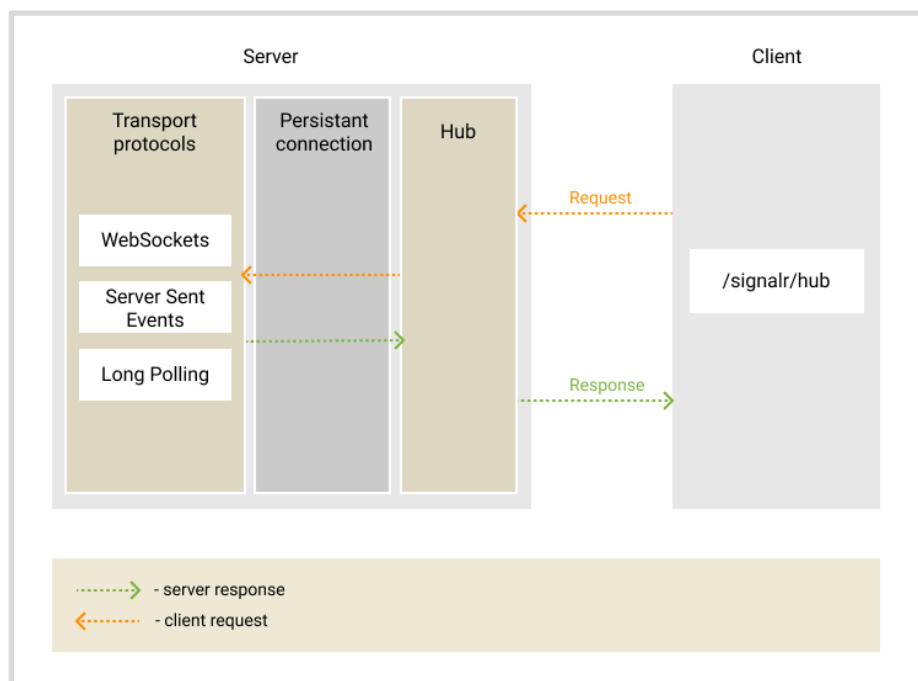


Figure 9: SignalR Hub concept

A hub is created by declaring a class which inherits from Hub - the base class for a SignalR hub. The created hub class can then have its own methods and the methods that are defined as public can be called by the client. [38,39,40,41,42] In the created application the hubs are kept in the Hubs folder. This folder contains the TestsHub.cs, TestRequest.cs, and TestResponse.cs files, with the TestsHub.cs being the one where the TestsHub class is defined inheriting from Hub. The TestsHub class is responsible for processing requests and contains the private random property together with the following public methods: runTest and generate. A close up view of the application's class diagram part that contains the TestsHub class is shown in Figure 10: Diagram of the TestsHub class below:

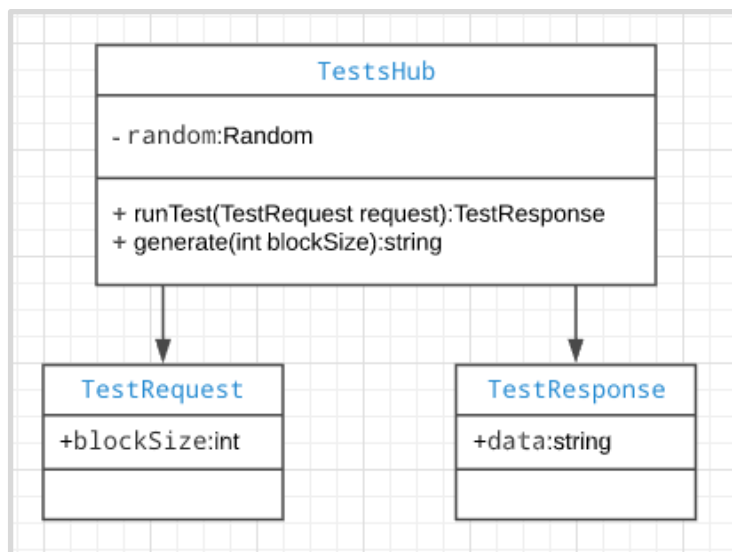


Figure 10: Diagram of the TestsHub class

The runTest method accepts the request from the client, based on the request's parameters generates the data block, creates the response and returns it back to the client. This method is invoked by the client during the performance tests. Below is a code snippet demonstrating the described method's implementation:

```

1 public TestResponse runTest(TestRequest request)
2 {
3     TestResponse response = new TestResponse();
4     response.data = this.generate(request.blockSize);
5     return response;
6 }
  
```

The request and response objects are defined as instances of the TestRequest and TestResponse classes respectively. These classes are defined in the TestRequest.cs and TestResponse.cs files and they are created for handling more complex requests easier and applying necessary



changes more swiftly. As shown in the code snippet below, the `TestResponse` class contains the `data` string property which corresponds to the string data block which is sent back to the client:

```
1 public class TestResponse
2 {
3     public string data { get; set; }
4 }
```

The string data is generated by the `TestsHub`'s `generate` method:

```
1 private static Random random = new Random();
2 public string generate(int blockSize)
3 {
4     const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
5     string result = new string(Enumerable.Repeat(
6         chars,
7         blockSize).Select(s => s[random.Next(s.Length)]).ToArray());
8     return result;
9 }
```

To ensure that each returned block size is the same, the `blockSize` property is passed to the `generate` method. This property is defined in the request sent by the client, which is represented by the `TestRequest` class. The implementation is provided in the code snippet below:

```
1 public class TestRequest
2 {
3     public int blockSize { get; set; }
4 }
```

The client folder contains the page structure (index.html) and style (style.css) as well as the logic for displaying and updating the performance test results (tests.js).

The tests.js logic is organized in the following sections. First the urls mapping for each of the data transfer types and the corresponding URL is defined as shown in the code snippet below:

```
1 | const urls = {  
2 |     "websockets": "http://olap.flexmonster.com:5011/tests/",  
3 |     "longpolling": "http://olap.flexmonster.com:5012/tests/",  
4 |     "serversentevents": "http://olap.flexmonster.com:5013/tests/"  
5 | }
```

Every test has its own index for identification and the first test is initially given the value zero:

```
1 | var testIndex = 0;
```

The performance test itself is represented by the Test class, this means that with each new test a new instance of the Test class is created and then the performance test is started. The `window.requestIdleCallback()` method ensures that the start function, responsible for starting the test, is called during the browser's idle periods, which in return helps to prevent impacting latency-critical events such as animation and input response. [43,44,45,46,47]

```
1  var test = new Test( testIndex++,  
2                        transport,  
3                        numberOfCalls,  
4                        blockSize  
5  );  
6  window.requestIdleCallback(() => {test.start();})
```

The Test class constructor's implementation is provided in the next code snippet and it sets the following properties: the index represents the identification of the corresponding test; the transport specifies the data transfer type; the blockSize defines the size of the data block that will be transferred during the performance test; the numberOfCalls stands for the amount of requests that will be streamed by the client during the test; the responseSize property displays the number of bytes loaded during the test; the callNum property is used for keeping track of how many requests have already been made during the test; the timeSpent property represents the total time spent during the performance test; the connection property is responsible for creating and starting the connection with the hub to call the needed hub methods from the client. In this case the hub that is connected to is the one defined by the TestsHub class described earlier in this section. The above description is provided in the code snippet below:

```

1  constructor(index, transport, numberOfCalls, blockSize) {
2      this.index = index;
3      this.transport = transport;
4      this.blockSize = blockSize;
5      this.numberOfCalls = numberOfCalls;
6      this.responseSize = 0;
7      this.callNum = 0;
8      this.timeSpent = 0;
9      this.connection = new signalR.HubConnectionBuilder()
10         .withUrl(urls[this.transport]).build();
11 }

```

The start method is responsible for starting the performance test. On line 4 the time calculation is started and on lines 5-9 the corresponding hub method is called by the `_invoke` function:

```

1  start() {
2      document.getElementById("btnRun").disabled = true;
3      this._addNewLine();
4      this.startTime = Date.now();
5      this.connection.start().then(() => {
6          this._invoke();
7      }).catch(error => {
8          this.addError(error);
9      });
10 }

```

On line 2 the `runTest` public method, which was defined in the `TestsHub` class on the server-side, is called from the client. The `invoke` method accepts the following parameters: the name of the hub method to be called and the required arguments that are defined in the corresponding hub method.

[38,39,40,41,42] In this case, the runTest hub method is called and the needed block size is passed to the hub method to specify the size of the data block that needs to be generated on the server and returned to the client.

```
1  _invoke() {
2      this.connection.invoke("runTest", {
3          blockSize: parseInt(this.blockSize)
4      }).then(response => {
5          this.addResponse(response);
6          this.draw();
7          if (this.callNum < this.numberOfCalls)
8              this._invoke();
9          else
10             this.stop();
11     }).catch(error => {
12         this.addError(error);
13     });
14 }
```

When the response is received, the number of loaded bytes is added to the responseSize variable via the addResponse method (line 5), the result is rendered in the draw method (line 6), and the hub method is called again if the number of currently made requests is below the total one specified in the input parameters of the test. If the set number of calls is reached, the method invocation is stopped and this means the performance test is finished. The addResponse method's implementation is provided below:

```

1  addResponse(response) {
2      this.responseSize += JSON.stringify(response).length;
3      this.timeSpent = Date.now() - this.startTime;
4      this.callNum++;
5  }

```

On line 3, in the `addResponse` method's implementation, the total spent time is updated and on line 4 the number of made calls is incremented. When the performance test is completed, the button to start a new performance test is enabled allowing the user to run another test:

```

1  stop() {
2      document.getElementById("btnRun").disabled = false;
3  }

```

Returning to the `start` method, when the test is started the test object is displayed to the user with the test parameter values via the `_addNewLine` method. The implementation is provided in the code snippet below:

```

1  _addNewLine() {
2      let rowHtml = `
3          <div class="output" id="output${this.index}">
4              <div class="label">${this.transport}</div>
5              <div class="label">${this.blockSize}</div>
6              <div class="label">${this.callNum}</div>
7              <div class="label">${this.responseSize}</div>
8              <div class="label">${this.timeSpent} ms</div>
9              <div class="label"></div>
10         </div>
11         <div class="delimiter"></div>`;
12     window.requestAnimationFrame(

```

```

13     () => {
14         document.getElementById("output").innerHTML += rowHtml;
15     }
16 };
17 }

```

The test parameters' display is updated via the draw method shown in the code snippet below:

```

1  draw() {
2      const percent = Math.round(
3          10000 * this.callNum / this.numberOfCalls) / 100;
4      let testHtml = `
5          <div class="label">${this.transport}</div>
6          <div class="label">${this.blockSize}</div>
7          <div class="label">${this.callNum}</div>
8          <div class="label">${this.responseSize}</div>
9          <div class="label">${this.timeSpent} ms</div>
10         <div class="label">
11             <div class="progress"
12                 style="width:${percent * 3}px"/>${percent}%
13             </div>`;
14         window.requestAnimationFrame(
15             () => {
16                 document.getElementById("output" + this.index)
17                     .innerHTML = testHtml;
18             }
19         );
20     }

```

In case of an error, the error object will be displayed to the user notifying that an error occurred and the performance test will be terminated. This is demonstrated in the code snippet below:

```
1  addError(error) {
2      this.error = error;
3      this.drawError();
4      this.stop();
5  }
6  drawError() {
7      let errorHtml = `
8          <div class="label error">error</div>
9          <div class="label error">error</div>
10         <div class="label error">error</div>
11         <div class="label error">error</div>
12         <div class="label error">error</div>
13         <div class="label"></div>`;
14     window.requestAnimationFrame(
15         () => {
16             document.getElementById("output" + this.index)
17                 .innerHTML = errorHtml;
18         }
19     );
20 }
```

The dist/browser folder contains the signalr.js file together with its corresponding minified version signalr.min.js, which represents the ASP.NET Core SignalR JavaScript client library enabling the server-side hub code to be called. The SignalR client is available as an npm package or through CDN. [41] For the application the npm package way was chosen. Below is the sequence of commands how to get the SignalR client library:

```
1  npm init -y
2  npm install @microsoft/signalr
```



At first a default package.json file is created on line 1, and on line 2 the SignalR client library package is installed. For the application only the signalr.js file was used out of @microsoft/signalr contents. According to the SignalR official documentation, the signalr.js file can be copied to the needed location in the application. [38,41] Then the signalr.js file is included in the page as shown in the code snippet below:

```
1 | <script src="./dist/browser/signalr.js"></script>
```

Each data transfer method is run on a separate port and in consequence the server for testing all of the mentioned data transfer approaches is started by running each of the commands below:

```
1 | dotnet run -t ws -port 5011
2 | dotnet run -t lp -port 5012
3 | dotnet run -t sse -port 5013
```

The first command is responsible for starting the server with WebSockets on port 5011, the second command starts the server with Long Polling on port 5012, and the third command runs the server with Server Sent Events on port 5013.

The created application is deployed and available by the following link: <http://cdn.flexmonster.com/vera/tests/index.html>. In Figure 11: Setting the data transfer method test parameters and Figure 12: Running the test for all data transfer methods the resulting performance tester application is shown. Upon initial visit the user is greeted by a simple interface providing several

input fields for setting the parameters' values for running the performance comparison tests. As shown on the screenshot in Figure 11: Setting the data transfer method test parameters, the input fields are initially displayed with default values for convenience but they can be adjusted if desired.

INPUT DATA						
Transport type	<input type="text" value="websockets"/>	Block size	<input type="text" value="1000"/>	Number of calls	<input type="text" value="100"/>	<input type="button" value="RUN TEST"/>
RESULTS						
Transport	Block size	Calls	Bytes loaded	Time	Progress	

*Figure 11: Setting the data transfer method test parameters*

The performance tests can be run for WebSockets, Long Polling, and Server Sent Events individually, or run for all three methods at the same time for a better visual comparison. In Figure 12: Running the test for all data transfer methods it is shown how the performance tests are run for all three technologies at one go. As a result, the input data is displayed for each data transfer method along with the number of bytes loaded and the time it took to complete the task. While the tests are running, a progress bar is shown both helping to estimate the waiting time as well as providing a visual comparison of the methods' performance.

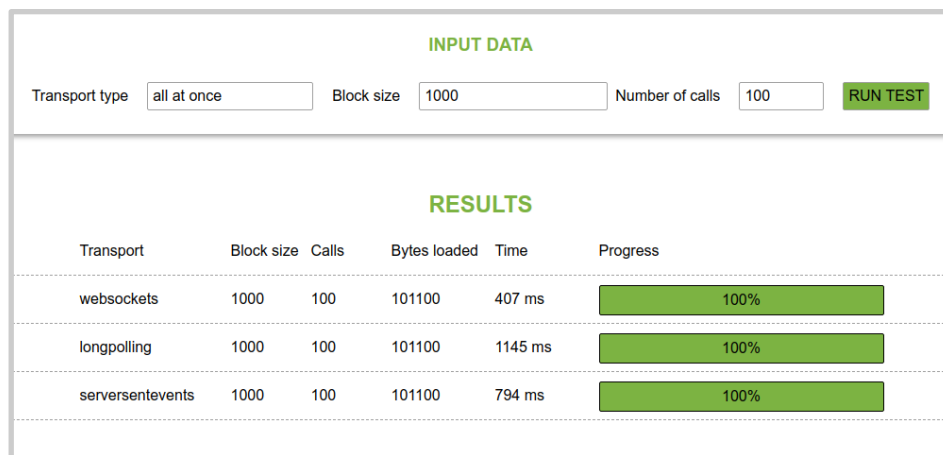


Figure 12: Running the test for all three data transfer methods

After running several tests for all of the methods at once, while increasing the number of calls each time, the test results were gathered in JSON format and fed to a chart library. On Figure 13: Performance time test results for 100 calls the comparison of the test results when 100 calls were made is illustrated on a bar chart type. It is clearly seen that of all the data transfer methods, WebSockets came forth as the most performance efficient. Long Polling took last place after Server Sent Events. Next, to these results, the outcome when 1000 calls were made is added. Again, WebSockets took first place, followed by Server Sent Events, leaving Long Polling far behind (see Figure 14: Performance time test results for 1000 calls). For 10000 calls the test results have the same outcome: first WebSockets, then Server Sent Events, and Long Polling taking the longest to complete (see Figure 15: Performance time test results for 10000 calls). The reasons why Long Polling's performance is drastically different is given in [section 2.1](#) of this course work.

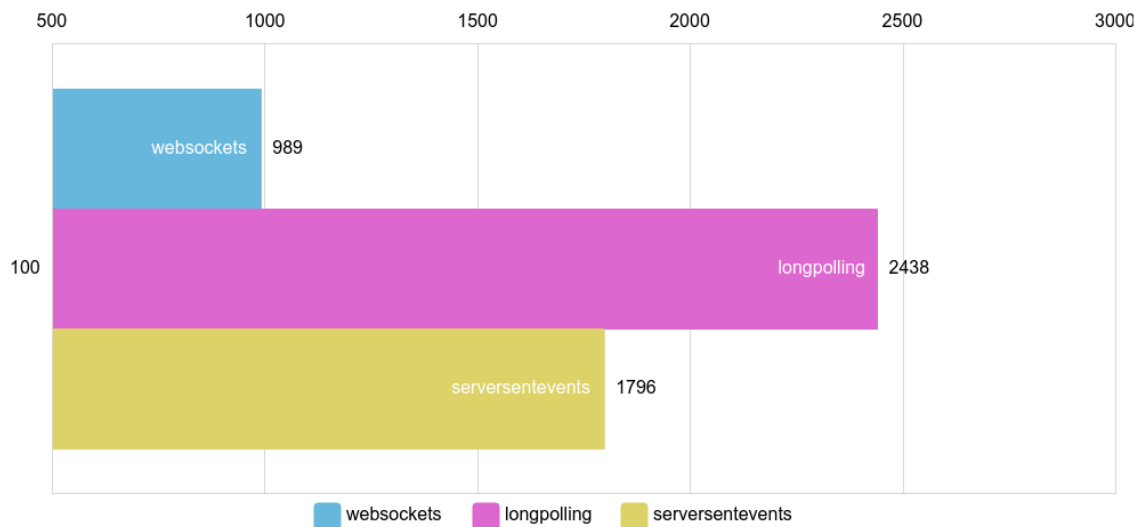


Figure 13: Performance time test results for 100 calls

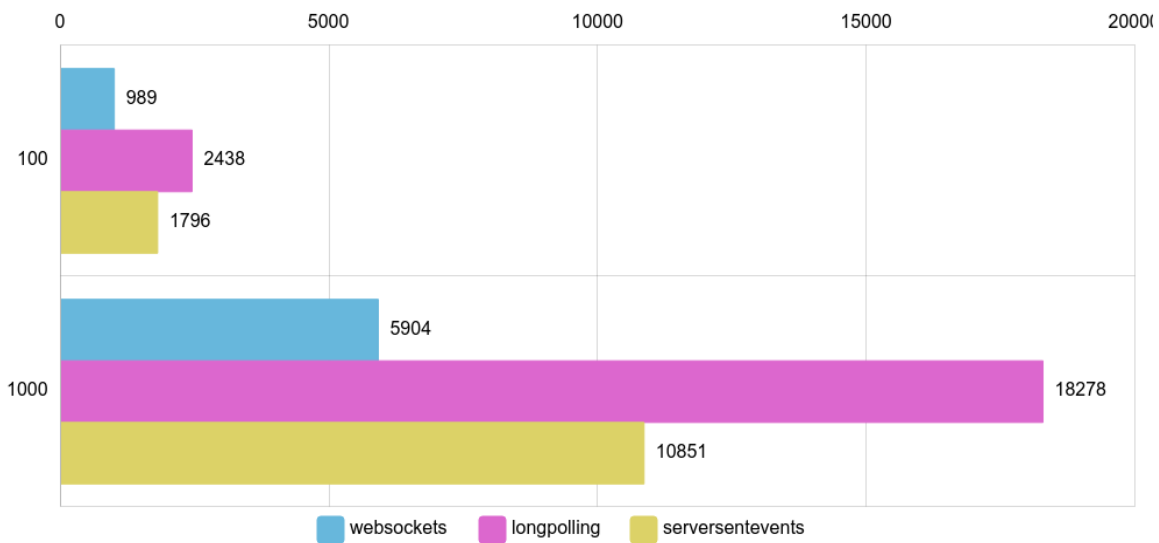


Figure 14: Performance time test results for 1000 calls

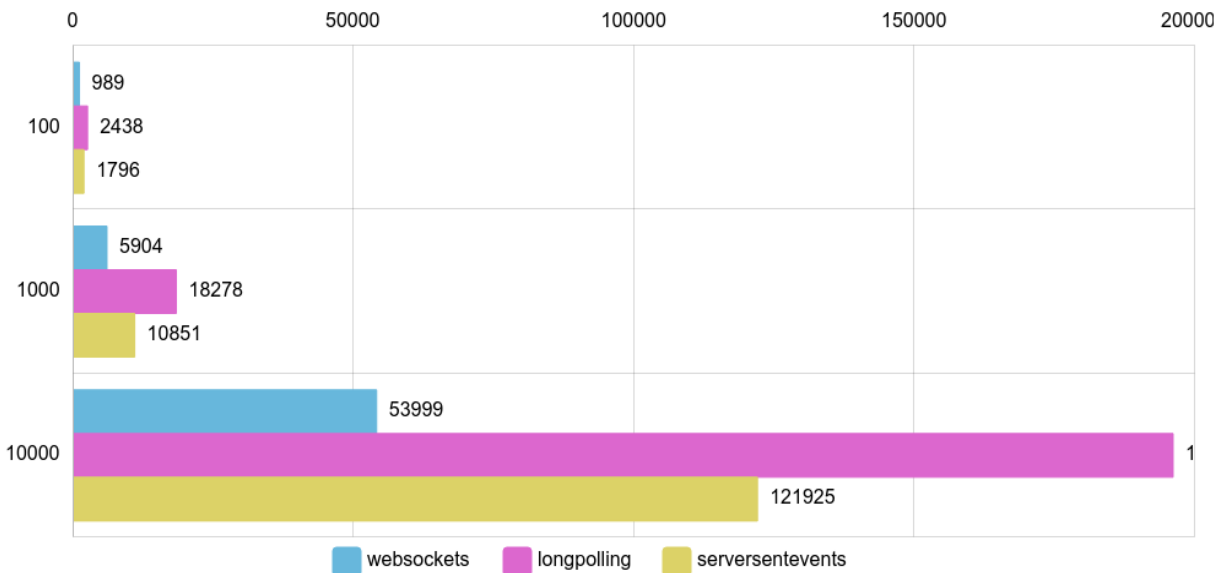
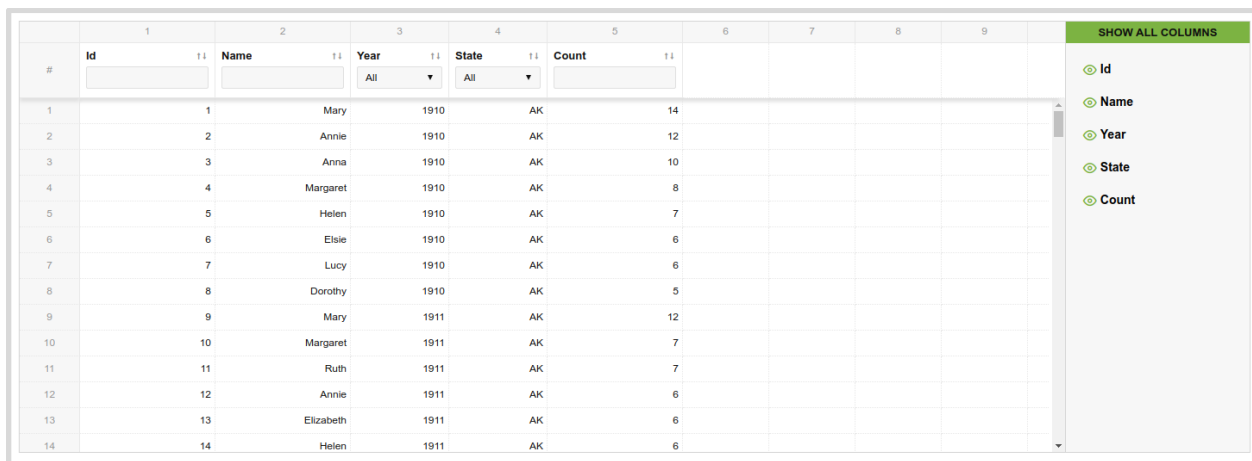


Figure 15: Performance time test results for 10000 calls

From the experiment results, the conclusion can be derived that WebSockets work best for the task at hand, coming forth as the most performance efficient data transfer technology in the simulated scenarios.

### Section 3: Displaying the large data set

As stated throughout this course work, the goal is to load and display a large data set to the user with minimal delay. As for the visualization part, the aim is to achieve smooth navigation on a large dataset, so that the display is not choppy, jerky or uneven and without sudden jolts. The data transfer protocol that is chosen is WebSockets as they came out to be the most performance efficient based on theory and backed up by the performance test results provided in [2.4 section](#) of this work. For the visualization part, a simple data table was created. The screenshot of the result is provided below (see Figure 16: Displaying the large data set on a data table):



#	Id	Name	Year	State	Count
1	1	Mary	1910	AK	14
2	2	Annie	1910	AK	12
3	3	Anna	1910	AK	10
4	4	Margaret	1910	AK	8
5	5	Helen	1910	AK	7
6	6	Elsie	1910	AK	6
7	7	Lucy	1910	AK	6
8	8	Dorothy	1910	AK	5
9	9	Mary	1911	AK	12
10	10	Margaret	1911	AK	7
11	11	Ruth	1911	AK	7
12	12	Annie	1911	AK	6
13	13	Elizabeth	1911	AK	6
14	14	Helen	1911	AK	6

Figure 16: Displaying the large data set on a data table

The resulting project is deployed to a server and is available by the following link: <http://cdn.flexmonster.com/vera/demo/index.html>. As a result, the smooth navigation effect was reached using the following techniques: data preloading, load balancing, and smooth rendering, in other words balancing the CPU usage.

### 3.1 Data preloading

When working with large datasets, the acceptable approach is to load the data by parts: in most cases it is not affordable to load the whole dataset into the browser at once as most user machines won't handle such an overload, causing the user interface to lag or freeze, leading to a potential browser crash. Therefore, in the data table, the data is loaded by parts. Aside from initially loading the data that is currently in the view point, data preloading is implemented while the user interacts with the data table depending on the scroll direction, thus forming a data "cushion" (please see Figure 17: Preloading data depending on scroll direction). The data "cushion" moves together with the scroll depending on the scroll direction making sure that data is always available to be shown within a certain range of the current view point. Hence, on scroll, the data loading process is not noticeable or visible to the user because data is preloaded on the go in advance. With this technique, on scroll, the user should not encounter empty rows, where the data portion has not been loaded yet, thus providing a more pleasant user experience. [52,53,54,55,56,57,58]

The diagram shows a central table with columns: id, Name, Year, State, Count. The table is surrounded by orange shaded areas labeled 'preloaded data'. A legend at the bottom right shows a dashed box labeled '- the data "cushion"'. The table content is as follows:

id	Name	Year	State	Count
400000	Alexandra	1984	CA	
400001	Archie	1984	CA	
400002	Archie	1984	CA	
400003	Archie	1984	CA	
400004	Archie	1984	CA	
400005	Archie	1984	CA	
400006	Archie	1984	CA	
400007	Archie	1984	CA	
400008	Archie	1984	CA	
400009	Archie	1984	CA	
400010	Archie	1984	CA	
400011	Archie	1984	CA	
400012	Archie	1984	CA	
400013	Archie	1984	CA	
400014	Archie	1984	CA	
400015	Archie	1984	CA	
400016	Archie	1984	CA	
400017	Archie	1984	CA	
400018	Archie	1984	CA	
400019	Archie	1984	CA	
400020	Archie	1984	CA	
400021	Archie	1984	CA	
400022	Archie	1984	CA	
400023	Archie	1984	CA	
400024	Archie	1984	CA	
400025	Archie	1984	CA	
400026	Archie	1984	CA	
400027	Archie	1984	CA	
400028	Archie	1984	CA	
400029	Archie	1984	CA	
400030	Archie	1984	CA	
400031	Archie	1984	CA	
400032	Archie	1984	CA	
400033	Archie	1984	CA	
400034	Archie	1984	CA	
400035	Archie	1984	CA	
400036	Archie	1984	CA	
400037	Archie	1984	CA	
400038	Archie	1984	CA	
400039	Archie	1984	CA	
400040	Archie	1984	CA	
400041	Archie	1984	CA	
400042	Archie	1984	CA	
400043	Archie	1984	CA	
400044	Archie	1984	CA	
400045	Archie	1984	CA	
400046	Archie	1984	CA	
400047	Archie	1984	CA	
400048	Archie	1984	CA	
400049	Archie	1984	CA	
400050	Archie	1984	CA	
400051	Archie	1984	CA	
400052	Archie	1984	CA	
400053	Archie	1984	CA	
400054	Archie	1984	CA	
400055	Archie	1984	CA	
400056	Archie	1984	CA	
400057	Archie	1984	CA	
400058	Archie	1984	CA	
400059	Archie	1984	CA	
400060	Archie	1984	CA	
400061	Archie	1984	CA	
400062	Archie	1984	CA	
400063	Archie	1984	CA	
400064	Archie	1984	CA	
400065	Archie	1984	CA	
400066	Archie	1984	CA	
400067	Archie	1984	CA	
400068	Archie	1984	CA	
400069	Archie	1984	CA	
400070	Archie	1984	CA	
400071	Archie	1984	CA	
400072	Archie	1984	CA	
400073	Archie	1984	CA	
400074	Archie	1984	CA	
400075	Archie	1984	CA	
400076	Archie	1984	CA	
400077	Archie	1984	CA	
400078	Archie	1984	CA	
400079	Archie	1984	CA	
400080	Archie	1984	CA	
400081	Archie	1984	CA	
400082	Archie	1984	CA	
400083	Archie	1984	CA	
400084	Archie	1984	CA	
400085	Archie	1984	CA	
400086	Archie	1984	CA	
400087	Archie	1984	CA	
400088	Archie	1984	CA	
400089	Archie	1984	CA	
400090	Archie	1984	CA	
400091	Archie	1984	CA	
400092	Archie	1984	CA	
400093	Archie	1984	CA	
400094	Archie	1984	CA	
400095	Archie	1984	CA	
400096	Archie	1984	CA	
400097	Archie	1984	CA	
400098	Archie	1984	CA	
400099	Archie	1984	CA	
400100	Archie	1984	CA	

Figure 17: Preloading data depending on scroll direction

### 3.2 Load balancing

In addition to data preloading depending on scroll, load balancing is a must. Avoiding big chunks when the data is being transferred to the client helps to achieve high throughput as well as improve the network bandwidth utilization. Invoking the data transfer and loading background processes during the browser's idle states is necessary in order to avoid impacting the visual representation such as animation and input response, which are latency-critical events that have a direct influence on the overall user experience. [43,44,45,46,47] During the data preloading process, large data blocks must be divided into a sequence of chunks to transfer the needed data portions in a non-blocking manner. It is also important to implement prioritized loading in order to ensure that the data that is about to be displayed to the user is loaded in time. [52,53,54,55,56,57,58]

Taking these three factors into consideration: avoiding big chunks, prioritized loading, and idle browser time usage, in the designed data table for displaying the large dataset to the user, in data transfer implementation a maximum data size block constraint is set to define the maximum data size portion that can be transferred at a time. In consequence, on scroll when the data needs to be preloaded in order to maintain the appropriate data "cushion" length, the data is transferred in blocks with the size under the maximum size limit. The data preloading is prioritized, meaning that the data that is closest to the user's view point is given the highest priority and hence loaded first. To invoke the data loading processes during the browser idle state, the browser idle time is detected using the `requestIdleCallback` method. The `requestIdleCallback` method is an elegant way to identify the



moment most appropriate for executing under the hood tasks, as well as predict how much time is left until the idle stage changes, therefore making it possible to estimate whether a certain process can be afforded to be completed at the given moment thence carrying out the procedures explicitly during the idle times. [43,44,45,46,47,48,49,50,51] Below the idle callback usage is illustrated on Figure 18: Executing the processes during idle browser time:

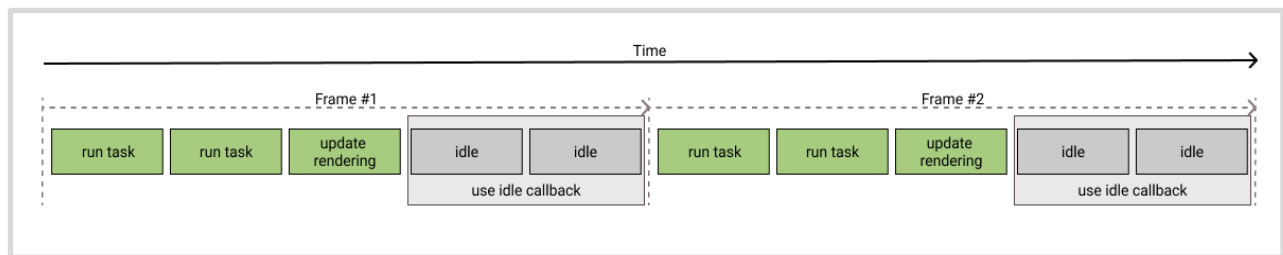


Figure 18: Executing the processes during idle browser time

### 3.3 Smooth rendering

In addition to data preloading and load balancing during the data transfer process, the `requestAnimationFrame` method was used for rendering the result to ensure one drawing per frame. [48,49,50,51] For reaching the smooth navigation, such as scrolling, effect the result needs to be rendered in a smooth and even fashion and as explained in [section 1.2](#) of this course work, for the display to be graceful a steady frame rate must be kept, therefore requiring to maintain each frame's length alike. [3,4,5,6] So, in the designed data table the `requestIdleCallback` method was used for executing background tasks such as data loading during idle browser times and the `requestAnimationFrame` method was used to execute the rendering code on the next available screen repaint to avoid possible reflows caused by recalculation of previously rendered elements. [52,53,54,55,56,57,58] This means the drawing process is executed when changes are ready to be made, thus providing a more efficient repaint, resulting in a smoother and more pleasant user interaction (see Figure 19: Optimized number of draws by painting the result in the next frame. [48,49,50,51]

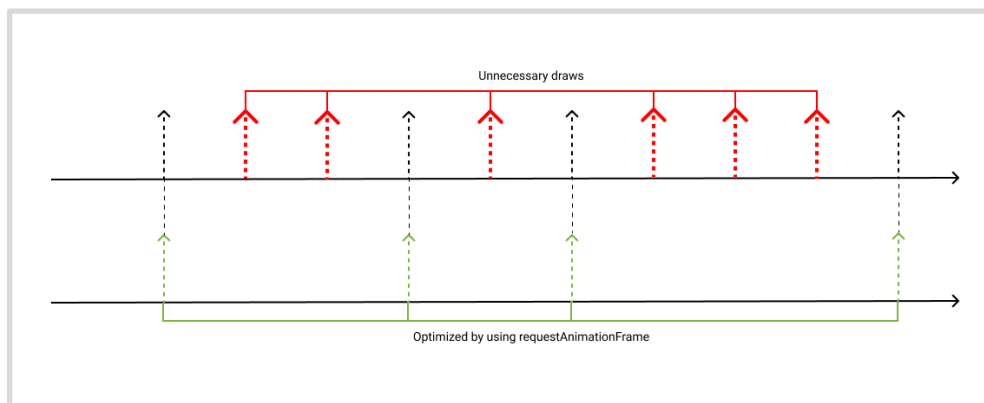
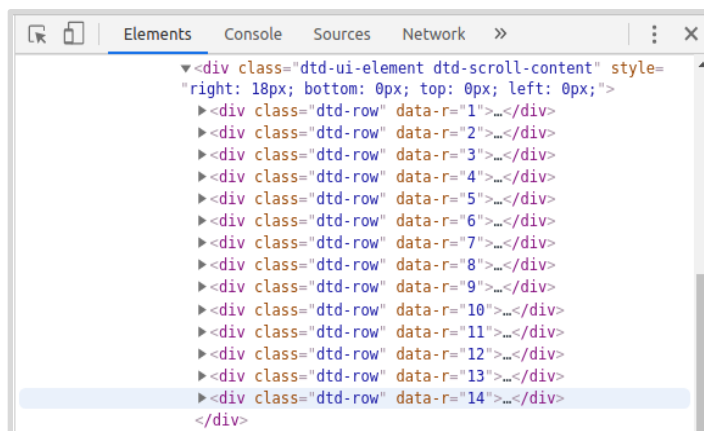


Figure 19: Optimized number of draws by painting the result in the next frame

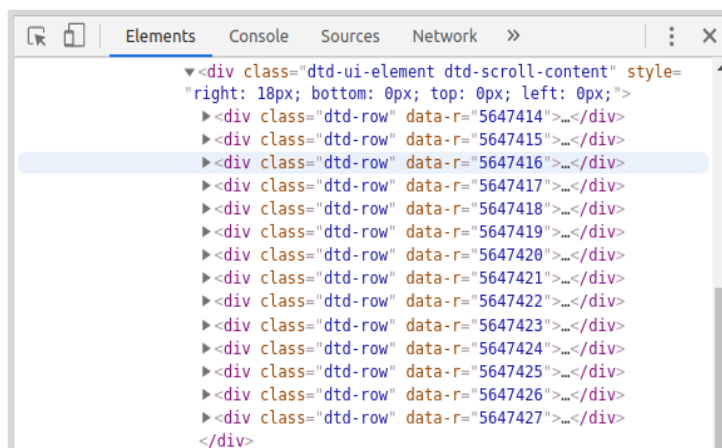
Furthermore, in the resulting data table the scroll content is dynamic - not all elements are rendered at once, only the ones that are in the viewpoint. On Figure it is shown how at first load only the data table rows which are in the viewpoint are rendered:



```
<div class="dtd-ui-element dtd-scroll-content" style="right: 18px; bottom: 0px; top: 0px; left: 0px;">
  <div class="dtd-row" data-r="1">...</div>
  <div class="dtd-row" data-r="2">...</div>
  <div class="dtd-row" data-r="3">...</div>
  <div class="dtd-row" data-r="4">...</div>
  <div class="dtd-row" data-r="5">...</div>
  <div class="dtd-row" data-r="6">...</div>
  <div class="dtd-row" data-r="7">...</div>
  <div class="dtd-row" data-r="8">...</div>
  <div class="dtd-row" data-r="9">...</div>
  <div class="dtd-row" data-r="10">...</div>
  <div class="dtd-row" data-r="11">...</div>
  <div class="dtd-row" data-r="12">...</div>
  <div class="dtd-row" data-r="13">...</div>
  <div class="dtd-row" data-r="14">...</div>
</div>
```

Figure 20: Rendering only the elements that are in view point

After scrolling further down in the data table and inspecting the page the amount of row elements didn't increase, instead the content changed to update the view. This is shown on Figure 21: Updating only the results below:



```
<div class="dtd-ui-element dtd-scroll-content" style="right: 18px; bottom: 0px; top: 0px; left: 0px;">
  <div class="dtd-row" data-r="5647414">...</div>
  <div class="dtd-row" data-r="5647415">...</div>
  <div class="dtd-row" data-r="5647416">...</div>
  <div class="dtd-row" data-r="5647417">...</div>
  <div class="dtd-row" data-r="5647418">...</div>
  <div class="dtd-row" data-r="5647419">...</div>
  <div class="dtd-row" data-r="5647420">...</div>
  <div class="dtd-row" data-r="5647421">...</div>
  <div class="dtd-row" data-r="5647422">...</div>
  <div class="dtd-row" data-r="5647423">...</div>
  <div class="dtd-row" data-r="5647424">...</div>
  <div class="dtd-row" data-r="5647425">...</div>
  <div class="dtd-row" data-r="5647426">...</div>
  <div class="dtd-row" data-r="5647427">...</div>
</div>
```

Figure 21: Updating only the results

## CONCLUSION

The goal of this research work was inspired by the desire to improve the user's experience when working with large data sets. To provide the best experience it was determined that the response time when the user interacts with the interface should be not more than a tenth of a second. Such a time limit was derived from the way the human brain works and how our eyes perceive information. Based on the 100 milliseconds constraint it was calculated how much time is affordable to spend for each process and the appropriate data transfer method was chosen to be WebSockets for loading the data and keeping it up to date. As a result of this work the goal was reached: a large data set was loaded and displayed in a datatable within a tenth of a second, and on scroll the display is even, smooth, without any choppiness or sudden jerks.

The smooth user experience was achieved by correctly balancing the CPU usage between data loading, calculations, and the rendering stages. By preloading data within the set range of the view point depending on scroll direction, as well as implementing prioritized data loading, it was ensured that data will always be available in time when it is needed to be displayed to the user, therefore, providing the instantaneous effect. The overall load was balanced by setting the maximum data portion size that can be transferred at a time, therefore avoiding big data chunks which in return also improved the throughput and the network bandwidth utilization. In addition, the processes were executed during idle browser times by using the `requestIdleCallback` method, thus preventing the user interaction to be blocked by background ongoing tasks. The rendering stage was organized

to have one drawing per frame when changes are ready to be made, thus avoiding unnecessary repaints, keeping the frame length alike, and maintaining a steady frame rate.

## RESOURCES

1. <https://developers.google.com/web/fundamentals/performance/rail>
2. <https://www.nngroup.com/articles/response-times-3-important-limits/>
3. <http://news.mit.edu/2014/in-the-blink-of-an-eye-0116>
4. [https://developer.mozilla.org/en-US/docs/Tools/Performance/Frame\\_rate](https://developer.mozilla.org/en-US/docs/Tools/Performance/Frame_rate)
5. [https://developer.mozilla.org/en-US/docs/Web/Performance/Animation\\_performance\\_and\\_frame\\_rate](https://developer.mozilla.org/en-US/docs/Web/Performance/Animation_performance_and_frame_rate)
6. <http://designingforperformance.com/basics-of-page-speed/>
7. <https://www.linkedin.com/pulse/20141119181300-142790335-site-speed-mobile-usability-and-seo/>
8. <https://aerotwist.com/blog/the-anatomy-of-a-frame/>
9. <https://medium.com/platform-engineer/web-api-design-35df8167460>
10. <https://wolfcrow.com/notes-by-dr-optoglass-motion-and-the-frame-rate-of-the-human-eye/>
11. <http://amo.net/NT/02-21-01FPS.html>
12. [https://en.wikipedia.org/wiki/Flicker\\_fusion\\_threshold](https://en.wikipedia.org/wiki/Flicker_fusion_threshold)
13. <https://learn.javascript.ru/long-polling>
14. <https://learn.javascript.ru/server-sent-events>
15. <https://www.filamentgroup.com/lab/weight-wait.html>
16. <https://hpbn.co/server-sent-events-sse/#eventsource-api>
17. <https://hpbn.co/server-sent-events-sse/#event-stream-protocol>
18. <https://hpbn.co/server-sent-events-sse/#sse-use-cases-and-performance>
19. <https://stackoverflow.blog/2019/12/18/websockets-for-fun-and-profit/>
20. <https://hpbn.co/websocket/#websocket-api>

21. <https://hpbn.co/websocket/#websocket-use-cases-and-performance>
22. <https://hpbn.co/websocket/#performance-checklist>
23. <https://pusher.com/websockets>
24. <https://hpbn.co/websocket/#websocket-protocol>
25. <https://www.internet-technologies.ru/articles/ispolzovanie-sse-vmesto-websockets.html>
26. <https://sookocheff.com/post/networking/how-do-websockets-work/>
27. <https://www.smashingmagazine.com/2018/02/sse-websockets-data-flow-http2/>
28. <https://medium.com/dailyjs/a-comparison-between-websockets-server-sent-events-and-polling-7a27c98cb1e3>
29. <https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77>
30. <https://codeburst.io/polling-vs-sse-vs-websocket-how-to-choose-the-right-one-1859e4e13bd9>
31. <https://aquil.io/articles/a-comparison-between-websockets-server-sent-events-and-polling>
32. <https://dzone.com/articles/websockets-vs-long-polling>
33. <https://www.ably.io/concepts/websockets>
34. <https://browsee.io/blog/websocket-vs-http-calls-performance-study/>
35. <https://medium.com/system-design-blog/long-polling-vs-websockets-vs-server-sent-events-c43ba96df7c1>
36. <https://www.ably.io/blog/websockets-vs-long-polling/>
37. <https://www.smashingmagazine.com/2018/02/sse-websockets-data-flow-http2/>
38. <https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr>

39. <https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1>
40. <https://docs.microsoft.com/en-us/aspnet/core/signalr/hubs?view=aspnetcore-3.1>
41. <https://docs.microsoft.com/en-us/aspnet/core/signalr/javascript-client?view=aspnetcore-3.1>
42. <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-3.1>
43. <https://developers.google.com/web/updates/2015/08/using-requestidlecallback>
44. <https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback>
45. <https://www.dnsstuff.com/network-throughput-bandwidth>
46. <https://www.w3.org/TR/requestidlecallback/>
47. <https://developers.google.com/web/updates/2015/08/using-requestidlecallback>
48. <http://www.javascriptkit.com/javatutors/requestanimationframe.shtml>
49. <https://dev.opera.com/articles/better-performance-with-requestanimationframe/>
50. <https://developers.google.com/web/fundamentals/performance/rendering>
51. <https://developers.google.com/web/tools/chrome-devtools/evaluate-performance>
52. <https://medium.com/javascript-in-plain-english/better-understanding-of-timers-in-javascript-settimeout-vs-requestanimationframe-bf7f99b9ff9b>
53. <http://creativejs.com/resources/requestanimationframe/index.html>



54. <https://www.udacity.com/course/browser-rendering-optimization--ud860>
55. <http://shop.oreilly.com/product/0636920028048.do>
56. <http://scienceadvantage.net/wp-content/uploads/2020/03/JavaScript-High-Performance-Build-Faster-Web-APPs-2020-Year-.pdf>
57. <https://dev.to/canastro/javascript-long-running-tasks-use-cpu-s-idle-periods-58g2>
58. <https://www.luigicolella.it/blog/how-to-use-js-background-tasks-api>