Linnéuniversitetet
Kalmar Växjö

Master Thesis Project

Evaluating Stream Protocol for a Data Stream Center

Author: Mahdi Mohammadnezhad
Supervisor: Jonas Lundberg
Examiner: Welf Löwe
Reader: Morgan Ericsson
Semester: VT/HT 2016
Course Code: 4DV50E
Subject: Computer Science

Abstract

Linnaeus University is aiming at implementing a Data Stream Centre to provide streaming of accumulated data from the websites' newspapers and articles in order to help its scientists of University to have faster and easier access to the mentioned data. This mentioned project consists of multiple parts and the part we are responsible to research about is first nominating some text streaming protocols based on the criteria that are important for Linnaeus University and then evaluating them. Those protocols are responsible to transfer text stream from the robots (that read articles from the websites) to the data stream center and from them to the scientists. Some KPIs (Key Performance Indicators) are defined and the protocols are evaluated based on those KPIs. In this study we address evaluation of network streaming protocol by starting to read about the protocol's specifications and nominating four protocols including TCP, HTTP1.1, Server-Sent Events and Websocket. Then, fake robot and server are implemented by each protocol to simulate the functionality of real robots, servers and scientists in LNU data stream center project. Later, the evaluation is done in the mentioned simulated environment using RawCAP, Wireshark and Message Analyzer. The results of this study indicated that the best suited protocols for transferring text stream data from robot to data stream center and from data stream center to scientist are TCP and Server-Sent Events, respectively. In the concluding part, other protocols are also suggested in the order of priority.

Keywords: Network protocols, Streaming protocols, Evaluation, HTTP1.1, TCP, Server-Sent Events, Websocket.

Preface

First of all, I would like to sincerely thank my supervisor, Professor Jonas Lundberg, for his brilliant pieces of advice, his constructive suggestions and the fruitful discussions and meetings we had for this thesis.

I would like to express my appreciations to my course manager Professor Narges Khakpour and my examiner Professor Welf Löwe, for helping me with my problems in this course and guiding me through my thesis improvements.

I also express my gratitude to Professor Morgan Ericsson as the reader, who read my thesis and commented for better result.

Naturally, I would like to thank my wife who has always been helping and supporting me through my life, especially during the busy days of my studies.

Contents

1 Introduction

This chapter presents an introduction to this thesis. It starts with some background information in Section 1.1 and continues with motivation for this thesis in Section 1.2. Section 1.3 and 1.4 explain about problem statement and research questions, respectively. Contribution is discussed in Section 1.5 and an outline of the report structure is explained in Section 1.6.

1.1 Background

Currently, the amount of information available on websites in different formats is increasing constantly. Part of this vast amount of information can be accessed and fetched as text. Some texts on the websites are articles with different subjects which are of interest for a large number of people. Among them, some are restricted and available for internal use only. However, most of them are globally available and the interested people can access them easily. The exponential growth of the mentioned texts and data has caused increase in access request by interested users, scientist, organizations and companies. It is really important and sometimes vital for them to have live and fast access to the mentioned data to remain competitive through monitoring and analyzing the data. For example, a scientist can analyze the data to find interesting relationships between them. Also, in businesses, the companies try to maintain their reputation by analyzing their customers' data, finding out about customers' complaints and solving the users' problems. This way, they can keep their customers happy and satisfied [35]. On the other hand, reputable companies like Twitter are getting more popular and can help businesses to acquire big amount of information (tweets) about specific subjects in which they are interested, through fast streaming. During the last few years, the topic of streaming has become fascinating to the researchers as streaming is about transfer of big amount of data which is increasing day by day. In order to transfer the mentioned data, it is needed to engage the streaming technology. For the same purpose, an appropriate protocol is needed to make this live stream happen. In fact, to govern the manner of data communication and to determine how to handle the errors, how to authenticate and assess elements such as syntax of data and headers, there are rules called communication protocols.

This thesis is aiming to help text-streaming transmission for Linnaeus University data stream center by evaluating four network protocol candidates selected based on project criteria including Websocket, TCP, Server-Sent Events and HTTP1.1 to find the most appropriate one.

1.2 Motivation

LNU Data Stream Center project is an ongoing project by Linnaeus University. This project is aimed to implement a data stream center to provide streaming of accumulated data from the websites' newspapers and articles in order to help its scientists to have faster and easier access to the mentioned data. A datacenter is considered as a virtual or physical centralized repository for computation, management, storage and dissemination of information and data and normally includes computers, servers' racks, cooling system, switch/routers and other related equipment [47]. This project consists of multiple parts such as implementation of robots (website-robots), implementation of infrastructure and implementation and use of an appropriate network protocol in order to transfer the data as text-stream from robots to the data stream center and deliver them from data stream center to the scientists (as our clients). This study concerns the last part of the project which is implementation and use of appropriate network protocol.

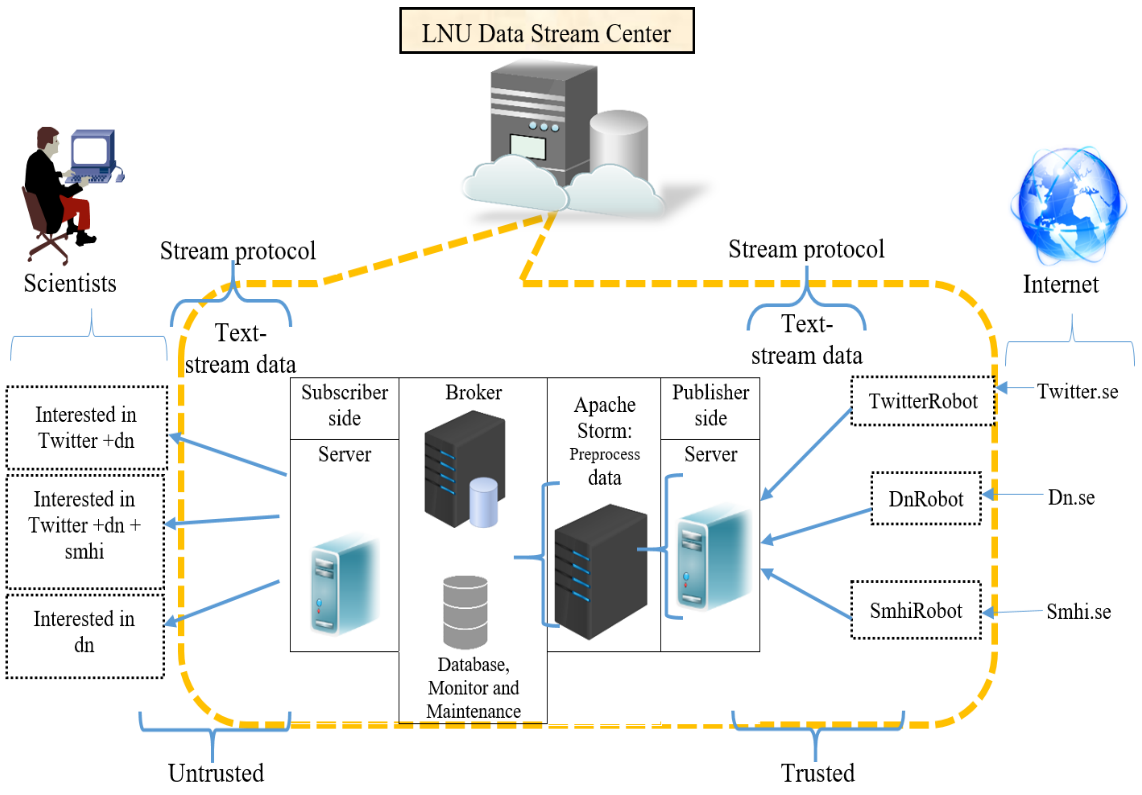


Figure 1: LNU Data Stream Center

Robot is responsible to monitor a website continuously and send the stream of data in text format or with JSON -JavaScript Object Notation- structure to the server (publisher side) through a channel (Figure 1). Before transferring the data from publisher to the broker, data preprocessing will be done by Apache storm [38] which includes sorting the data based on its language, filtering and adding extra information. After processing the data, it will be pushed to the broker and database. Broker will be implemented by Redis (Nosql database) [39] which enables us to categorize processed data that will be received from Apache storm. Also, the same processed data will permanently be stored in a database in order enable us to have future access to them. Scientist can subscribe to several different channels on broker to identify interesting correlations or can ask to have the oldest data from the database. But a problem may happen in this part which is called stall problem and happens when the destination side of text-streaming is slower than the source side. Handling stall problem is not part of this study so we will not further discuss it here.

Since we would like to preprocess the data, it is not possible to use message queue oriented protocols or connect the robots directly to the broker. We must have client-server relation in order to transfer the data from robots to the data stream center and some kind of channel to transfer the data from robot the server (Publisher side). In order to implement a channel, a protocol is needed to provide text-stream data between robot and server –the publisher-side. In the same way, an appropriate protocol is needed to be identified to stream data from server -subscriber side- to scientists. Data transmission between robots and data stream center (publisher-side) is trusted and we don't have any firewall in that side. But the communication between data stream center and the scientists (subscriber side) is untrusted and we are not able to open any certain port for data transmission. Another issue is that we do not have any information about the implementation type of client that will receive data from data stream center (on subscribers' side) and therefore we are not sure if it will be implemented by a browser or application. TCP as a transport layer protocol can not interact directly with a browser and most of the browsers are supporting application level protocols like HTTP. In fact, client implementation for scientists is not part of LNU Data Stream Center Project.

It is noteworthy that the criteria for selecting protocols are decided based on the requirements of the LNU project by the project manager and are to be: guarantee of packet delivery, compatibility with text-streaming in client-server implementation relation and using HTTP capabilities as an advantage (more explanation about the criteria and KPIs will be given in Subsection 3.2).

Communication network has significant impact on performance of data centers [46], [47] and in this thesis, we are primarily concerned with nominating some text streaming protocols to govern data communication with LNU data stream center. In the next step fake robot and fake server will be implemented by each protocol in our experimental environment which is a localhost machine, in order to simulate the functionality of actual ones. Finally, the selected protocols will be evaluated based on some KPIs such as: payload length, bandwidth usage, delta-time and interruption in data transmission. However, it is important to note that utilizing each approach to transfer text stream has some pros and cons and it is not true to say which protocol is the best and which one is the worst, since they are not fully alternative for each other.

We have emphasized on text format in this study since the data that is generated by robot in LNU data stream project is in text format with JSON structure. Therefore, other formats are not concern of this study.

As it explained earlier, we don't have access to the scientist's side and we don't have necessary information about client implementation in that side but we will do the same experiment for both, publisher and scientist sides since they are doing the same thing and text streaming should be transferred in both sides. The difference is in interpretation of experimental results that the lack of information has caused.

1.3 Problem Statement

In today's world of information and communication, the data on the internet increases extraordinarily. "On Wall Street and other global exchanges, electronic trading volumes are growing exponentially. Market data feeds can generate tens of thousands of messages per second" [36]. In order to provide fast access to the mentioned information in an organization, it is needed to engage streaming technology.

Moreover, in order to utilize streaming technology, we should find the most appropriate protocol to govern the data stream based on the organizational needs (e.g. in order to transfer the data event handling is needed or not), specifications (e.g. the client side is a browser or application) and criteria (e.g. how much the bandwidth usage of a specific protocol is important in order for text-streaming) to help the scientists in different sectors to have fast access to the data and information they need. By this thesis we aim to nominate some protocols that better meet the LNU project's defined criteria (in subsection 1.2) and are able to transfer text as stream. The criteria as defined by the project manager are: guarantee of packet delivery, compatibility with text-streaming in client-server implementation relation and using HTTP capabilities as an advantage. Then we evaluate the protocols based on some KPIs to identify the best suited one for our purpose. These KPIs include: payload length, bandwidth usage, delta-time and interruption in data transmission.

1.4 Research Questions

Having the mentioned aim, his thesis is going to find the answer to the following questions:

RQ1: Which protocols better fulfill the criteria of LNU Data Stream Center Project for text streaming?

RQ2: Among the selected protocols, which one is the best to be used for the text-streaming from the robots to the LNU data stream center?

RQ3: Among the selected protocols, which one is the best to be used for the text-streaming from the LNU data stream center to the scientist?

1.5 Contributions

In this thesis we first nominate and then evaluate Websocket, Server-Sent Events, Http1.1 and TCP protocols to understand their behavior and how they perform with regard to transfer of text-stream. Adding new technology is always challenging and the results of this thesis can help organizations which intend to utilize one of the mentioned protocols, to govern text-stream data communication. This would be especially helpful with Server-Sent Events and Websocket which both are new compared to HTTP1.1 and the traditional TCP protocol. Hence, we believe that the results of this thesis can be useful for network developers, who decide to transfer text-stream over the wired network with similar plan as that of Linnaeus University (from the robots to the data stream center and from the data stream center to the clients) or for any other client and server data communication.

1.6 Report Structure

This thesis is organized as follows:

The thesis starts with chapter 1, the present chapter, which is an introduction of the research topic. This chapter starts with covering the background and motivation for this thesis. Also, the problem statement and research questions are explained and contribution of the research to the domain is included.

The second chapter concerns how and why the nominated protocols are identified. Moreover, each protocol is explained to provide a background for the topic under research.

Methodology and data collection methods and justification for the use of the chosen methods are discussed in chapter 3.

Chapter 4 is called implementation which starts with explaining about the tools that are used during the experiment. Later it continues with necessary explanation about the experimental environment and explanation about main the part of our implementation.

Chapter 5 is devoted to evaluation. Firstly, the evaluation metrics will be clarified and then the collected data will be visualized and analyzed. Later the result will be evaluated in two part. This chapter is divided into two experimental parts and a summary is included at the end of each part.

In Chapter 6 the final conclusion from this study is summarized and suggestion for further research is presented.

2 Background

In this chapter the necessary information about the candidate protocols will be presented. Section 2.1 explains about the protocol search and selection process. Section 2.2 reviews the necessary information about each candidate protocol and is divided into four subsections. This section is devoted to TCP as Subsection 2.2.1, HTTP1.1 as Subsection 2.2.2, Server-Sent Events as Subsection 2.2.3 and Websocket as Subsection 2.2.4.

2.1 Protocol Search

As the main goal of this thesis is evaluating text-stream protocols, we decided to find our candidates among those that are able to transfer text data as stream. In fact, there exists quite a large number of protocols over the internet that are involved in transmitting the data. The limitations of such a project does not provide us with the conditions to test and evaluate all the existing protocols. Therefore, we needed to limit our list of protocols to be evaluated in this project. For the same purpose we used LNU project requirements that define a set of criteria to select our candidate protocols. To find out which protocols among this large number of existing protocols meet our project criteria we did a literature review. Selection of a list of protocols as part of this study needed a review of different sources about the existing protocols to justify a list for our evaluation. In our literature review we looked for the protocols which are being more commonly used for similar purposes. We also searched about famous companies that have streaming APIs like Twitter, to understand the approach and the protocol they are using. Our review of literature indicated that HTTP is the protocol being used by Twitter for text streaming [50]. Similar studies have been done in the related fields for some protocols, but for different infrastructures and with different goals. For example, some researches have been done in order to understand about the performance of a specific protocol [3] and some others are devoted to a specific infrastructure [16].

It is an advantage for a network protocol to be able to send and receive data in two direction (bidirectional) and at the same time (full-duplex). However, there are some protocols that are not bidirectional but can be used as stream protocol, for instance HTTP is not bidirectional, but it is the basis of some other protocols. Some protocols are using HTTP for handshaking (like Websocket) and some other protocols have been created by changing the structure of HTTP (like Server-Sent Events). HTTP-based protocols (which can use HTTP both for data transmission or handshaking) are more effective on delivering streams since they can be used with the standard web servers.

After these review of the protocols specifications in the related literature, a list protocols are nominated based on the LNU Project criteria which are as follows:

- The text-stream protocol must be appropriate to be implemented in stand-alone client-server relation (explained in Section 1.2);
- The protocol must guarantee packet delivery;
- Being able to use HTTP protocol's capabilities during the communication is an advantage.

We exclude those protocols which are used for video streaming or audio streaming. Using TCP in video and audio streaming causes some delay due to routing selection and error control of this protocol [44]. For the same reason audio and video streaming protocols are generally using UDP [40,45] to transfer the data and this may lead to packet loss since there is no mechanism in UDP to guarantee to deliver the packets. Moreover, these days RTP (Real-Time transport Protocol) and RTMP (Real Time Message Protocol) are two basic popular protocols that are being used for media streaming [48]. RTP is designed for audio streaming and video streaming like video teleconference which is based on UDP protocol [49]. RTMP (made by Adobe) is another streaming protocol but does not use caching mechanism like HTTP that causes increase both in data transmission speed and in compatibility with most of the existing firewalls. Also RTMP is designed for streaming data between a Flash player and a server [48]. Consequently, we exclude RTP and RTMP. Also for the same reason we exclude some other protocols like WebRTC (Web Real Time Communications) and UDT (UDP-based data transfer protocol). This is because WebRTC is made mainly for video and voice streaming rather than text streaming and both protocols are UDP based and cannot guarantee packet delivery [51] [52]. Message queue protocols are also excluded since based on the criteria and because of data preprocessing in data stream center we need to implement protocol in client and server relation. Therefore, it is not possible to directly transfer data from robots to the broker (Figure 1), while message queue protocols can be used to transfer data from robot to broker directly without need to any client server relation implementation.

Based on the aforementioned criteria, and according to what explained in Subsection 1.2 about trusted and untrusted side, our qualified protocol for trusted side (robot to data stream center) are TCP, HTTP1.1 and Websocket. Also our qualified protocols for untrusted side (data stream center to scientist) are HTTP1.1, Server-Sent Events and Websocket. Therefore, all our qualified protocols are listed as below:

- *Websocket*: It is bidirectional and full duplex protocol, and has HTTP level handshaking and guarantees the packet delivery.
- *TCP*: It is bidirectional and full duplex protocol which guarantees the packet delivery.
- *HTTP1.1*: It is a new version of HTTP protocol which can be used as text-stream with delivery guarantee.
- *Server-Sent Events*: It is a unidirectional protocol from server to the client only. HTTP based which can guarantee delivery of text data as stream.

2.2 Protocols

As discussed earlier in this chapter, we decided and chose our protocols based on some criteria. In this section we are going to discuss the candidate protocols in more details. All the four chosen protocols are explained and the required information about them are provided under the four sub-sections below.

2.2.1 TCP

It is really important to understand the Transmission Control Protocol (TCP) since it is the main protocol in the internet protocol suites [15]. TCP is working with Internet Protocol (IP) and is making a bi-directional connection which means the client and server can send and receive data simultaneously. The TCP connection is maintaining a communication until client or server sends the message in order to finish transmission. For the same reason TCP is called a connection-oriented protocol. TCP is a transport layer protocol which ensures packet delivery over the internet and makes reliable connection between client and the server over an unreliable network [17].

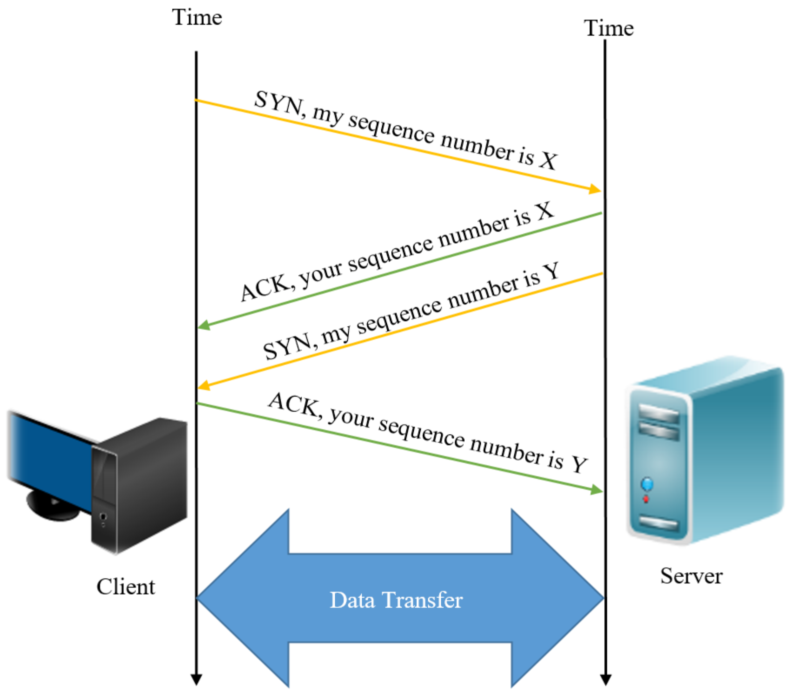
There is a conceptual and logical network model that divides the network communication to seven layers in OSI model or four layers in TCP/IP model (Figure 2.1). Each layer builds upon the previous layer and performs a specific task to complete the communication. To transmit the data via TCP, data encapsulation is needed and for the same purpose the data have to pass to TCP from upper layer protocol (Figure 2.1) [18]. After encapsulation of the data to a segment, it is time to encapsulate the given segment to datagram and send it via point-to-point transmission over IP in physical layer.

Application	Application
Presentation	
Session	
Transport	Transport
Network	Network
Data Link	
	Physical
Physical	
a) OSI model	b) TCP/IP Mode

Figure 2.1: (a) OSI model and (b) TCP/IP model [18]

To Establish a TCP connection from client to the server, which is called three-way handshaking, first client sends SYN (Sync request) to the server and lets the server know about its sequence number (for example X). Then the server responds with ACK (Acknowledge response) that means it has received the sequence number X and will send SYN and its sequence number (for example Y) to the client [18]. Finally, the client will send its ACK to the server [18] and the data transmission will begin (Figure 2.2).

On the other hand, in terminating a connection, the scenario is slightly different from establishing the connection. One point can start the terminating scenario with sending the FIN (Finish) to another point. To be more specific, assume that client sends FIN to the server, then server will respond with ACK, that it has received the message, and FIN to terminate the connection. In the final step, the client will send the ACK to the server to say that it has received the server's message and the connection will be close as soon as the mentioned message is received by the server [18] (Figure 2.2).


Time
Time
SYN, my sequence number is X
ACK, your sequence number is X
SYN, my sequence number is Y
ACK, your sequence number is Y
Client
Data Transfer
Server

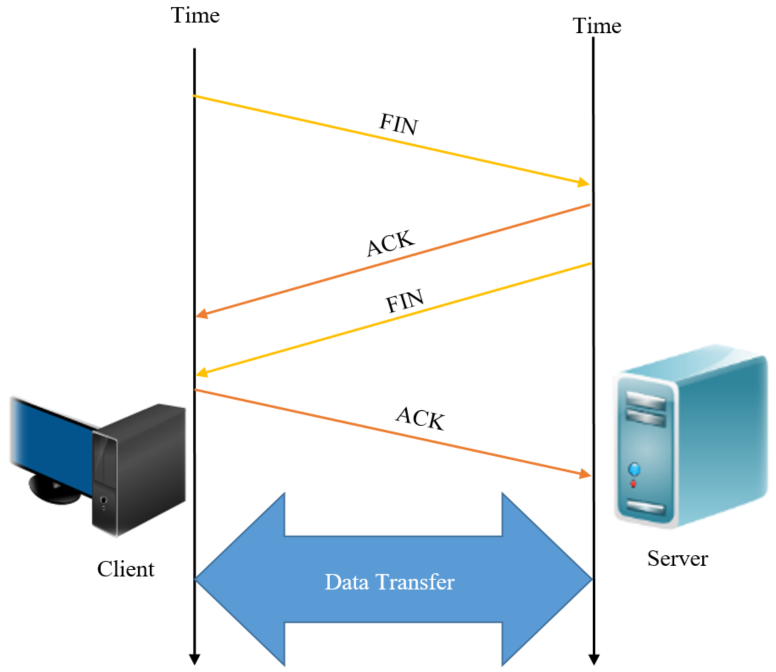


Figure 2.2: Establishing a TCP connection (up) and terminating a TCP connection (down)

These days it is hard to pass through the internet via TCP port since most of the firewalls block TCP ports and HTTP (80) and HTTPS (443) ports became the standard port to communicate over the internet. Also all modern browsers are using HTTP1.1 persistence connection [41] and its plain request and response. Explanation about this type of connection will be presented in the next subsection.

2.2.2 HTTP1.1

This traditional communication protocol for the World Wide Web has been announced in early 1990[8]. HTTP uses a synchronous way to send and receive the data between web client and web server as the client sends the request to the server and waits for the response from the server [2]. This process is sufficient for some of the web clients which need to load the webpages and post the request to the server and then wait for the results [2].

This scenario which is used by HTTP1.0 protocol, is used in about 75% of internet backbone traffic and has been improved by adding and updating its features in HTTP 1.1 [1]. In fact, HTTP1.1 is an enhanced version on HTTP1.0 that transfers the data in a more secure way [4]; but one of the biggest differences between HTTP1.0 and HTTP1.1 is in their implementation that causes different packet transmission structures through the network.

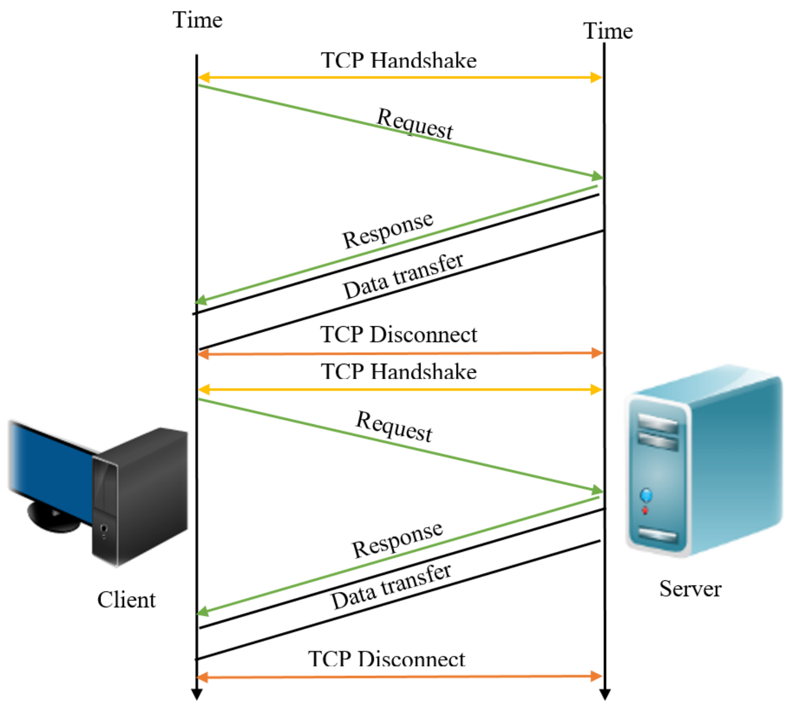


Figure 2.3: Request and response steps in HTTP 1.0

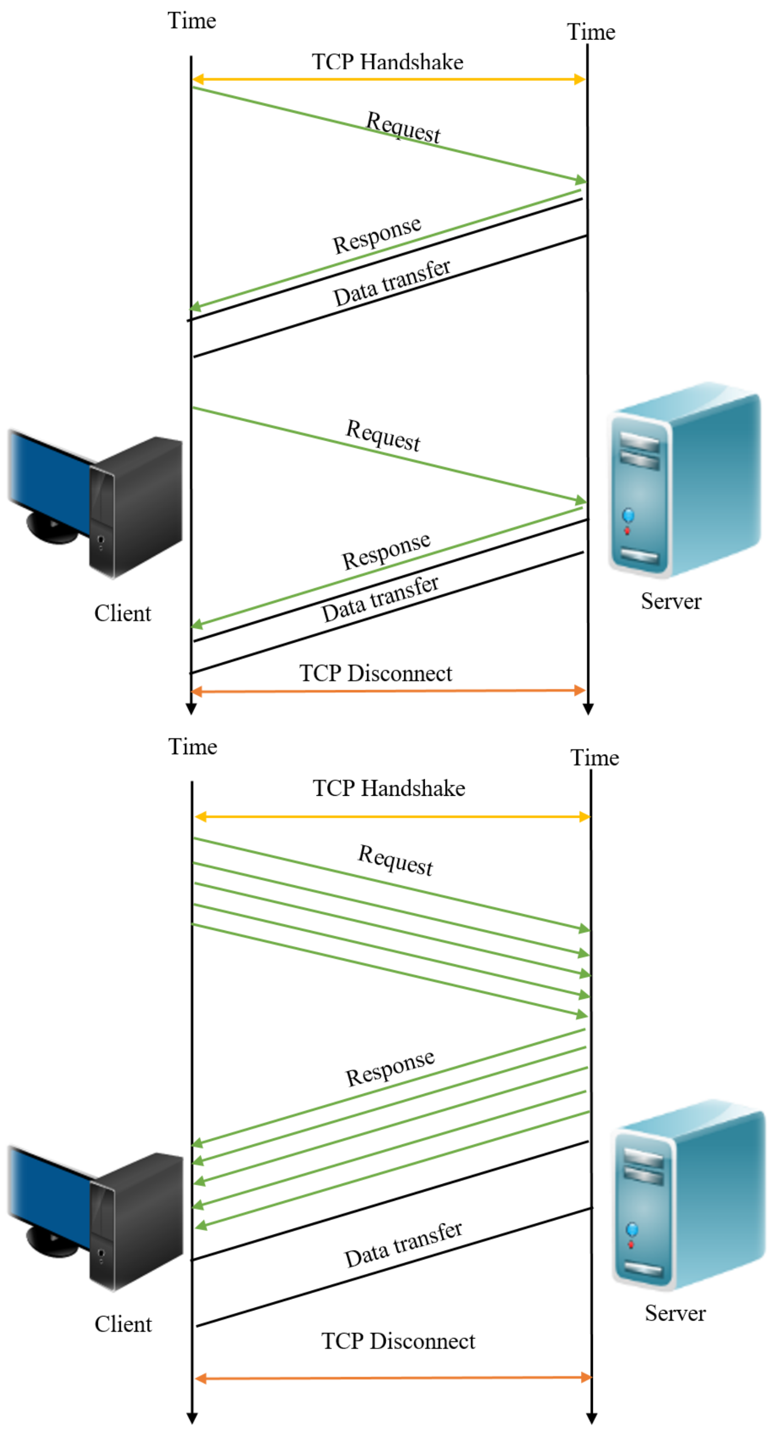


Figure 2.3.1: Request and response steps in HTTP 1.1(up) persistent connection, (down) pipeline in persistent connection

As Figure 2.3 indicates in HTTP1.0, the client has to send request per each data transfer and after the data transmission, the connection will be closed [1]. The same scenario must be implemented for each request and response. This is while HTTP1.1 uses a persistent connection [6] to have more than one request/response per each HTTP connection and due to this feature, HTTP1.1 can be used to fetch data as a stream [1]. HTTP1.1 runs on top of TCP (like HTTP1.0) and allows multiple request to be pipelined (Figure 2.3.1) in buffer in order to transfer the data faster and consequently reduce the web traffic. It should be mentioned that in some cases of implementation of HTTP1.0, using the keep-alive header can make a connection similar to persistent connection in HTTP1.1, but it makes some proxy-server issues. This is while HTTP1.1 and its persistent connection [11] provide better request handling via proxy servers. Persistent connection increases the performance through many ways such as reducing the network overhead, reducing the IO calls and as a result reducing the system resource usage [3]. Lack of persistence-connection feature by itself is convincing enough to continue with HTTP1.1 and omit HTTP1.0 from the candidate protocols list, as it is not possible to transmit the data as stream without persistence-connection. From now on, we will continue with HTTP1.1 and will not get in more detail about the older HTTP1.0 protocol.

It should also be mentioned that HTTP1.1 uses both polling and long polling [5] techniques to receive data (or notification/event) from server. In polling, the client has to send request for each notification (even if there is not any notification ready) while in long polling the server will keep the connection alive until an event becomes available to be sent to the client [12].

Moreover, HTTP1.1 uses *caching* mechanism that can improve distribution efficiency and quality of the service by reducing the redundant information-request that causes decrease in both CPU load and network load [3]. It should be noted that only static resources should be cached and caching the dynamic resource will prevent the user from receiving up-to-date data, which is not recommended [3].

2.2.3 Server-Sent Events

As mentioned in HTTP section, there are some approaches to receive event from the server such as polling and long polling. However, polling will use more network resources since it needs to send request (GET) to the server for each notification [11]. The mentioned notification may not be ready in the same time, so the client has to send the request again [11]. For the long polling, it is true that server keeps the connection open and will not reply to the request immediately, but server will close the connection once the new message becomes ready and be sent to the client [11].

Another approach to transfer data between the client and server is Server-Sent Events which originally has been developed by Opera (2006) [21]. In fact, Server-Sent Events is an API which keeps open an HTTP connection for a long time, to receive push notification from the server in a special form [22]. This form can be registered on different event handlers by programming languages [22].

However, Server-Sent Events is a JavaScript API with support within all HTML5-complaint browsers [23]. Server-Sent Events is similar to HTTP streaming connection and uses regular HTTP technique (like push technique). It is an advantage to make unlimited (in time) persistent connection between the client and the server to send the events to the client while the mentioned event (messages) can have optional event-name [11]. Server-Sent Events is not a substitution for polling and long-polling [23]. However, depending on the purpose for which Server-Sent

Events is used, it may serve as an optimized version of them [23]. Server-Sent Events is designed to standardize and decrease complexity in HTTP's Comet-based (Long-held HTTP request) strategy for future web-application [24].

Figure 2.5 portrays establishing a connection in Server-Sent Events. As it is indicated in Figure 2.5, the client sends a regular HTTP request to the Server-Sent Events' server. The only difference between regular HTTP request and Server-Sent Events client's request is the content type of request message.

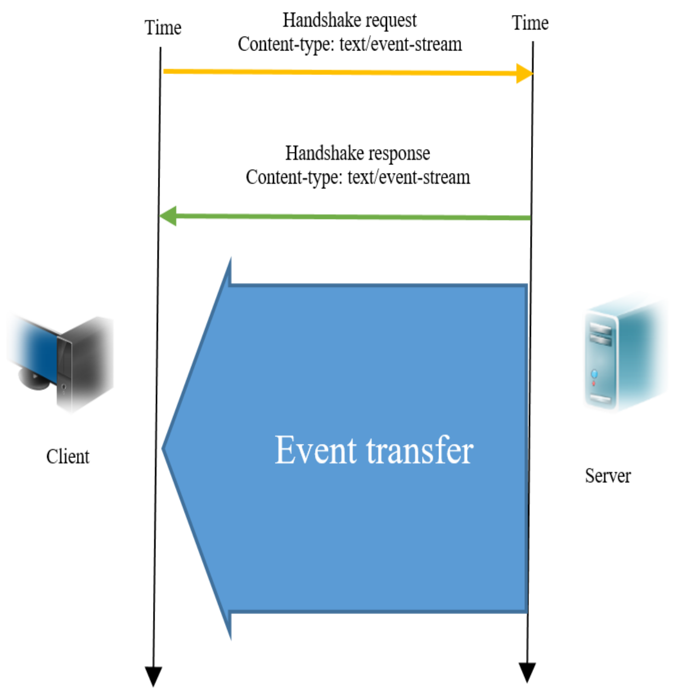


Figure 2.5: Server-Sent events diagram

The client must specify text/event-stream to let the server know the content type needed. If the server supports event streaming, it will respond with the same content type (text/event-stream) and from that point, inside an unlimited (time) streaming connection, the events will be sent (in the format of text) to the client whenever it is ready on the server.

This protocol uses a subscribe-publish model which means when the client subscribes to a channel, it will receive updates from the server (real-time) without needing any request from the client [23]. Also in case of any disconnection, the client can leverage the Last-Event-ID header which leads to let the server know about the missed events and the server will resend all the missed events [23]. In fact, the server restarts streaming events from the point the client left off [20].

2.2.4 Websocket

What HTTP does is acceptable for most of the network infrastructures. These days HTTP (and HTTP based) technology is used more by companies as it is more firewall friendly and also more ISP (Internet Service Provider) and organizations support its features. But HTTP, as a synchronous technology can not completely satisfy the new services like steaming data transfer, which needs to be more compatible and flexible with asynchronous data inside the web. In fact, the applications which need real-time data, have to use different protocols according to their criteria to meet their needs and that is the reason why different protocols are defined in Internet Protocol Suite [9]. The User Diagram Protocol (UPD) and Transmission Control Protocol (TCP) are two well-known protocols that are used in real-time communication. While UDP does not guarantee the packet delivery of the same sequence of data, TCP guarantees that. For instance, VOIP technology is over UDP and most of the web applications use application layer protocol over TCP. However, as it was mentioned earlier, among the application layer protocols that transmit the data over TCP, HTTP is used more frequently. But, considering the restriction of HTTP regarding its header overhead and synchronous structure, we introduce another protocol in this subsection.

The protocol that was recently introduced as part of HTML5 standard is called Websocket which can be claimed to be the main upgrade in web communications history [10]. Websocket is an application level, bi-directional and full duplex communication protocol which means it can send and receive the data at the same time simultaneously on top of TCP by bringing the flexibility of TCP to the web applications. Websocket can be used by any client and server as long as they are implementing the Websocket protocol. Also all the modern browsers are compatible with Websocket [43].

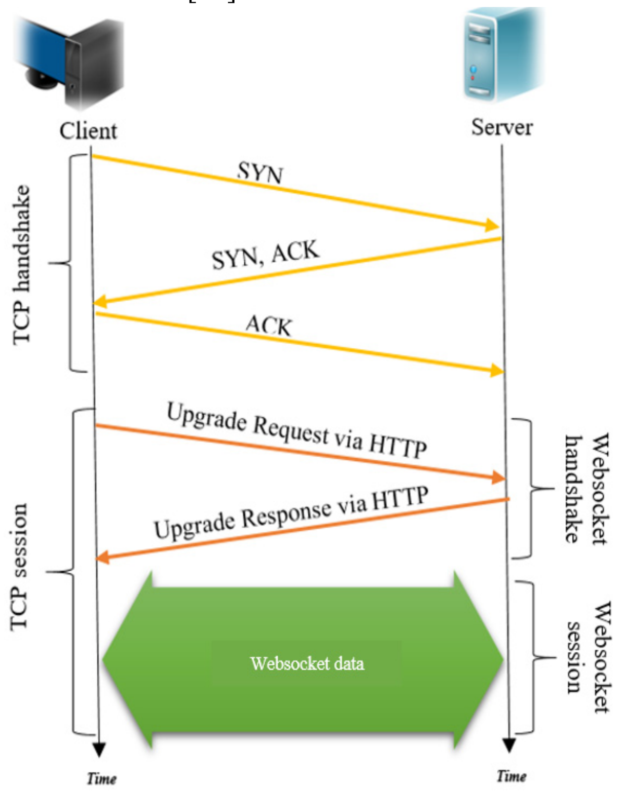
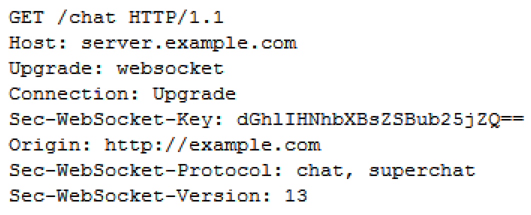


Figure 2.6: Websocket diagram

In fact, the bi-directional technique in Websocket performs as an alternative for polling technique in HTTP [12]. Websocket is similar to TCP not only in functionality, but also in handshakes with HTTP 1.1 and also it inherits all the benefits of existing web infrastructure such as: native web browser support, firewall/proxy compatibility, URL-based endpoint and omitting the length limit [13]. Websocket uses an asynchronous way to transmit the data (Figure 2.6).

Websocket starts with TCP handshaking that contains three messages and starts from the client to the server (three-way handshaking). Later, to establish a Websocket session, two messages will be exchanged starting from client to the server. For the same purpose, Websocket is using HTTP handshaking level (by using the benefit of HTTP protocol) that starts by sending upgrade request from the client.

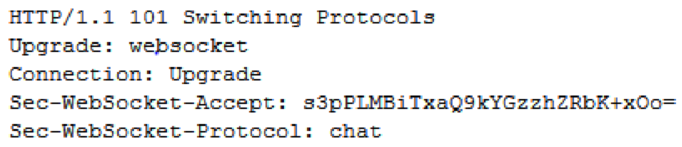
The client sends Websocket Upgrade Request via HTTP GET request. HTTP header contains the information that are needed to be upgraded to the Websocket connection (Figure 2.7).



```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Figure 2.7: Websocket upgrade request [14]

If the server supports the Websocket protocol, it will answer the Websocket Upgrade Response (Figure 2.8).



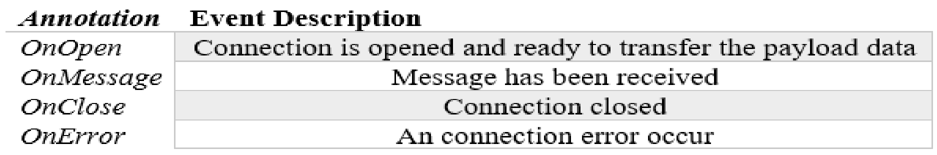
```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Figure 2.8: Websocket upgrade response [14]

From this point, the data between client and server will be transmitted in asynchronous and full-duplex mode and client and server can send application-specific payload to each other [13]. Websocket is a message-oriented protocol. Based on Websocket API, Websocket protocol has a feature called Event-Handling which handles the events efficiently. The client and the server are able to communicate with each other at the specific time when connection is established, message is received, connection is closed and any error occurs. Websocket Event-

Handling makes it easier to communicate between client and server asynchronously [30].

When a Websocket connection is established, the server will push the updates to the client in an asynchronous way [31]. Moreover, the client will receive message about OnOpen (when connection is established) and for each message it will receive message about OnMessage from the server (Figure 14.2). Also the client will receive message, in case of any error or when connection is closed due to OnError and OnClose methods, respectively.



Annotation	**Event Description**
OnOpen	Connection is opened and ready to transfer the payload data
OnMessage	Message has been received
OnClose	Connection closed
OnError	An connection error occur

Figure 2.9: Websocket Event-Handling

It is worth mentioning that by using Websocket protocol, clients and servers are allowed to convert their HTTP connection to completely different protocols in the format that they agree on [20]. Utilizing the Websocket protocol will create a persistent connection that reduces the number of transmitting packets by eliminating the HTTP overhead and request [14]. Considering these features of Websocket, we selected it as another candidate protocol to be evaluated in this research.

3 Method

The Method chapter explains about the method we used to solve the problem under this study. This Chapter consists of three sections. In Section 3.1 we will explain about the research method that we used. Section 3.2 will clarify why we selected and used our specific research method. Section 3.3 explains about how we collected our data and increased their validity.

3.1 Scientific Approach

In this thesis we will nominate some appropriate protocols for text-streaming according to project criteria and evaluate them in our simulated environment based on some KPIs. In a number of previous studies with the similar subject [13, 16] a simulated environment was created and quantitative method was used to answer the research question. Quantitative method is investigating observable phenomena via statistical, computational or mathematical technique in a systematic empirical way [42]. Any numerical form of data can be quantitative data and can be analyzed in quantitative method [42].

In this study we will evaluate our nominated protocols through numerical data we collected using a set of tools. Therefore, in this thesis a quantitative method will be used and a set of experiments will be conducted in order to collect the required data to find the answer to the research questions.

We started by reading about each protocol's specification to collect some information about them and prepare our list. In the next step we will build our experiments environment by implementing client and server by each protocol and simulate the functionality of real robots and real servers. The quantitative and main data for this study will be collected through experiments and the mentioned data will be analyzed in the next step. For the same purpose we will use some standard analysis tools such as *RawCap* to collect the data, and *Wireshark* and *Message Analyzer* in order to visualize our data. Later we will interpret the results based on the visualized data. More explanation about the tools used for the experiments are given in Chapter 4.

3.2 Method Description

After our literature review and reading articles with the similar subjects, we decided that the standard way to compare different protocols is to generate the results from experiments. As it was explained in Chapter 2, some criteria are defined based on the project requirements in order to select our protocols. By reading about each protocol's specifications we could gather the information and decide which protocols meet our criteria.

We also defined some KPIs to evaluate protocols in experimental environment. The mentioned KPIs can not be measured by reading protocols' specifications since it is not a trustable way to measure them by comparing their structure and specification.

Table 3.1 illustrates a summary of all the requirements that we specified to select our candidates. The first three requirements (in Table 3.1) that are explained in Chapter 2 are those criteria about which we collected information by reading each protocol's specification:

- Being able to implement in client-server relation to transfer text stream
- Packet delivery guarantee
- Being able to use HTTP protocol advantages

Candidates	*Requirements*						
	Text-streaming in client server relation	*Packet delivery guarantee*	*Use HTTP advantages*	*Delta-Time*	*Payload length for one message*	*Bandwidth usage*	*Interruption and number of TCP packet transmission*
TCP	✓	✓	-	-	-	-	-
HTTP1.1	✓	✓	✓	-	-	-	-
Server-Sent Events	✓	✓	✓	-	-	-	-
Websocket	✓	✓	✓	-	-	-	-

Table 3.1: protocols' requirement

Based on the information in Table 3.1, our nominated protocols met most of the requirements. The last four KPIs in Table 3.1 are those which should be measured through experiments in order to evaluate the protocols. For instance, some evaluation metrics like bandwidth usage can only be measured through experiment and by using experimental method. In our experiments, the following KPIs will be measured for the protocols:

1- *Delta-time:* Which is the time passed to receive one message from source to the destination. A low delta-time means the given message has been received to the destination faster.

2- *Payload length:* Which is the size of packet (normally is shown in bytes) that contains the necessary data (given message) and may be different for each protocol because of differences in architecture of each protocol. Payload length can affect the amount of generated network traffic.

3- *Bandwidth usage:* Which more depends on payload length (packets with bigger payload length need more bandwidth to be transferred) and the number of packets which are exchanged between client and server to transmit message from the source to the destination. It is worth mentioning that more bandwidth usage needs more hardware resources to transfer the data.

4- *Interruption in data transmission*: Interruption can be generated by duplicate ACKs, application glitch and lost segments [32].

In order to measure these KPIs, a simulated lab will be used to put the protocols under stress. Also we will get help from a number of standard network tools including *RawCap*, *Wireshark* and *Message Analyzer* in order to monitor and visualize the data. The mentioned tools will also be used to collect the data that are needed to evaluate the nominated protocols.

Java programming language will be used to implement fake robot by each protocol to simulate the functionality of robots and generate fake data, similar to the data which will be generated by real robots. Also, we will use some external library in order to implement fake robots. In the next step, we will implement four different servers by each protocol to receive fake data from each mentioned robot. In the implementation of servers, we will use Java programming language and the necessary external libraries. In the program that we will develop for each protocol, it is possible to echo the messages in a specific size and specific delta-time interval between client and the server. It means that it is possible to increase or decrease the size of message. Also the speed of generating the messages can be changed. As it was explained earlier, in LNU Data Stream Center project, the data will be transferred from the servers to the scientist-clients and the same condition applies there.

At the beginning the idea was to use LNU's server to perform experiments, but it was not agreed by the University and we will use one machine to do our experiments. Our client and server implementation will be done on a same host machine and they will be connected to each other on a localhost network. The hardware and software specifications of mentioned machine is given in Table 4.1.

	Client	*Server*
CPU	Intel Core i7 – 3630QM 2.4-3.4 GHZ	Intel Core i7 – 3630QM 2.4-3.4 GHZ
RAM	16 GB DDR3	16 GB DDR3
Network	Qualcomm Atheros AR8151 PCI-E Gigabit Ethernet Controller (NDIS 6.30)	Qualcomm Atheros AR8151 PCI-E Gigabit Ethernet Controller (NDIS 6.30)
Operating System	Windows 10 64-bit	Windows 10 64-bit
TCP implementation	Java.net.socket (Java JDK 8)	Java.net.socket (Java JDK 8)
HTTP implementation	jersey (Jersey.client 1.9.1)	jetty (Jetty 9.3.0)
Websocket implementation	jetty (Jetty 9.3.0)	jetty - servlet (Jetty 9.3.0)
Server-Sent Event Implementation	jersey (Jersey.client 1.9.1)	jersey (Jersey.media.SSE 2.22.2)

Table 4.1: Experimental environment configuration

In our implementation, we will use the following external library and framework:

- *com.googlecode.json-simple* (json-simple 1.1.1) in order to generate message in json format. This library will be used in implementation of fake robots for TCP, Websocket and HTTP1.1 protocols. Also this library will be used in Server-Sent Events' server.
- *Spring framework* (3.2.5) to configure and setup Websocket's fake robot and server.

Also, REST (Representational State Transfer) architecture will be used in implementation of HTTP and Server-Sent Events in order to facilitate communication with the server. REST is a software architecture style that counts on a stateless client-server communication and tries to reduce network communication and latency, while at the same time increases the independency and scalability of server [33]. To communicate with a REST server, these four basic actions are needed: Create, read, update and delete which are all together called CRUD [34].

Then, data transmission will be started in our simulated environment and by using our network analysis tools we will collect the required data, visualize the data and evaluate them based on experiment's results. For the purpose of this thesis, given the limited time, resources and facilities, we settled with the mentioned approach; otherwise, we preferred to do our experiments in a real infrastructure (for example in Linnaeus University's IT department, over its network and on real servers).

3.3 Reliability and Validity

A research can be considered as valid when it uses an appropriate tool for measurement and it is reliable when the result and the experiment can be repeated in other environments and situations [37]. In this study we will use standard network monitoring tools to collect our data. At the beginning, we will implement the fake robots and servers for each protocol separately in order to transmit text-stream. During the streaming we will use our candidate protocols to govern the mentioned data transmission. In order to increase internal validity, we will disable all the possible processes on localhost traffic that may affect the setup except those which are needed for the experiments. Therefore, a network mapping method like NAT will not have effect on the result. Also we will repeat the experiments to make sure about the internal validity of the results. It should be mentioned that the result of this research will not affect learning and habituation since the client and server that are implemented in this thesis are not able to learn or experience.

In order to increase the external validity, we will generate the messages in JSON format in data transmission since the message format in real project is JSON. Afterwards, the protocols will be put under stress by increasing the message size and decreasing the time interval for sending the messages. This is because the size of article that will be read by real robot may be bigger than what is expected. This way we will increase external validity and understand each protocol's behavior and how they handle big size message transmission.

Another factor that we suppose may increase the external validity is to implement clients and servers in a way to be able to decrease the interval time between generating the messages. The reason is that in real LNU project, the number of robots may be increased based on the scientists request and by increasing the number of robots, the interval time between generating the messages will be decreased.

In order to increase the reliability of the research we will do the experiments in the same environment for each protocol. Also we will transfer the same message by each protocol to ensure the consistency of this study. Moreover, we do experiments with reliable and standard tools and we will repeat our experiment several times in some cases where the results are a bit fluctuated. Then we will calculate the average from our experimental results. It is important to note that this experiment is to be conducted on wired network and the results of the same experiments on wireless network can be different as the condition for wireless network is different. For example, wireless network is more affected by environmental noises and bit error while wired network is more affected by wire related problems like congestions [19]. Therefore, the result from the present experiment can not be generalized for the wireless network.

4 Implementation

This chapter explains about our implementation scenario. Section 4.1 clarifies more about the tools we used during the experiments. In Section 4.2 the experimental environment is described and section 4.3 is devoted to implementation codes, but only the main parts of implementation codes are explained. Subsection 4.3.1 to 4.3.4 are dedicated to implementation of fake robot and server for TCP, HTTP1.1, Server-Sent Events and Websocket respectively.

4.1 Tools Used

Our review of related literature indicated that in other similar studies, tools like WireShark Network Protocol Analyzer [25], NetBeans [26] HTTP Server Monitor and the like have been used [17]. However, during this project we need to monitor and analyses packets in more detail. So we will disregard tools like NetBeans HTTP Server Monitor and Eclipse TCP/IP monitor and will continue with tools that have more features to get information about packets payload content, size, creation time and so on.

Also in order to more easily collect information about the packets, we decided to use the analysis tools which have features showing the packets as stack; these features are called Stack-View. To elaborate more on Stack-View, assume that 3 packets are involved in transmitting a message, Stack-view will show this message as a stack with three layers; each layer shows one of the mentioned packets.

In this study, for observing and analyzing the network traffic, the following tools will be used:

1- *RawCap*: A free command-line tool that can be used to monitor network traffics on windows and is using raw sockets [28]. Also it can create log file to be used in WireShark and Message Analyzer. During this project, RawCap is used to collect the localhost-traffic logs and make file with *pcap* suffix which can be opened and analyzed in Wireshark and Message Analyzer.

2- *Wireshark*: This tool helps to analyze network protocols and understand what happens inside network in depth [25] and generate statistic for network traffic. Wireshark is used in this project to help in analyzing the given pcap file and generate the bandwidth usage graph.

3- *Message Analyzer*: This tool has the same functionality as WireShark, but contains more features (like Stack-View) to analyze packets in detail. Message Analyzer is used to capture the packets in detail and help in analyzing network traffic [29]. During this project, Message Analyzer is used in order to help us to find delta-time data and generate the graph to analyze data interruption.

4.2 Implementation

As it was discussed in Section 3.2, we implemented fake robot to simulate the functionality of real robots. In all of our implementations, we used a class called *FakeRobot* to create the customized messages in the desired size and in JSON format. This class contains a big text message by default, the size of which is customizable and can be increased to double, triple and more, based on requested size.

From now on, in this section we will call the fake robots as "*clients*" to make it easier to follow. Clients replace the real robots in order to generate text message. The size of such messages is customizable and the time interval between sending the messages can be changed.

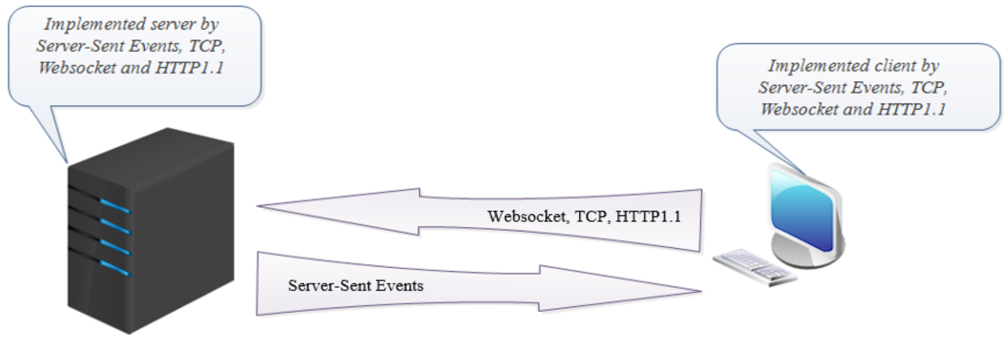

Implemented server by Server-Sent Events, TCP, Websocket and HTTP1.1
Implemented client by Server-Sent Events, TCP, Websocket and HTTP1.1
Websocket, TCP, HTTP1.1
Server-Sent Events

Figure 4.1: Transmitting fake text (in JSON format) from client to the server by Websocket, HTTP1.1 and TCP protocols

As our implementation shows in Figure 4.1, transmission of Websocket, HTTP1.1 and TCP protocols is from client to the server side, while transmission of Server-Sent Events is from the server to the client. This is because Server-Sent Events is a unidirectional protocol that transmit the data only form server to the client. In this thesis we implemented a simple version of client and server, using each protocol in order to do our experiments. In the next sections of this chapter, we will present only the main part of our implementation's code that is important to understand the process of this study. Further detailed explanations and implementation codes can be found in References list [53].

4.2.1 Client and Server Using TCP Protocol

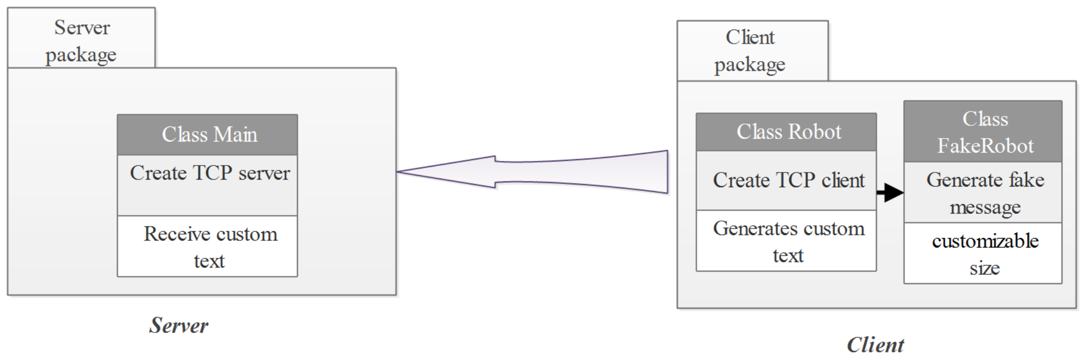


Figure 4.2: TCP implementation summary

As the summary shows in Figure 4.2, our client implementation consists of two classes, *FakeRobot* and *Robot*. *Robot* plays the role of the main class and *FakeRobot* is used in order to create a big message with the desired size. Then we created TCP socket on desired IP and port and transferred the message to the server within the desired time intervals. When the server starts, it will listen to the defined port (*serverPort*) and allows the client to establish the connection. The connection will shut-down when the client sends a line that contains *closeConnection* only.

4.2.2 Client and Server Using HTTP1.1 Protocol

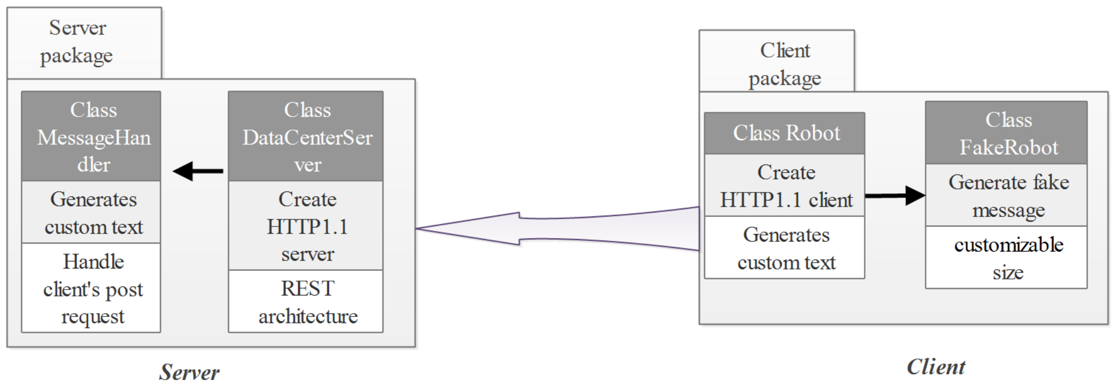


Figure 4.3: HTTP1.1 implementation summary

As Figure 4.3 above indicates, in implementation of HTTP1.1 the direction of text streaming is from client to the server. Client implementation consists of two classes. We avoid repeated explanation about *FakeRobot.* We named the main client class as *Robot.* In this class *TimerTask* has been used to manage time interval and the FakeRobot creates the messages. As it is shown in Table 4.1, we used *Jersey* to create our HTTP client and send messages by *post* request to the REST server.

In the server side we used *Jetty* library to implement the server which contains two classes. We named the first class *DataCenterServer* and it is the main class which creates, configures and runs the server. The second class is named *MessageHandler* that uses @POST annotation to handle *post* request from the client and to receive the message. In the *DataCenterServer* we setup server on a port that we want to start data transmission on.

4.2.3 Client and Server Using Server-Sent Events Protocol

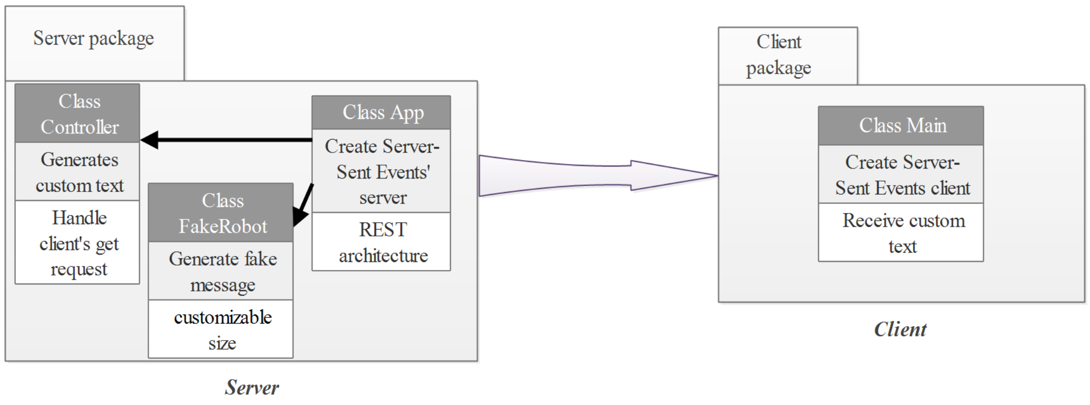


Figure 4.4: Server-Sent Events implementation summary

We used *Jersey* in order to implement Server-Sent events' client and server. In our implementation, the client can send get-request to the server and a while-loop gets event from the server as long as the events are available; otherwise, the connection will be closed.

As the summary indicates in Figure 4.4, the server side that answers the client's get-request consists of three classes. Apart from *FakeRobot* that was explained earlier, *DataCenterServer* is responsible to configure and run the server on the given port and IP address. Another class named *MessageHandler* is responsible to generate text and handle the *Get* request from the client. In this class we used *FakeRobot* like our other implementations to generate the customized text that is saved in *fakeMessage*. Then we used a *thread* to manage generating messages by sending *fakeMessage* within a desired time interval.

4.2.4 Client and Server Using Websocket Protocol

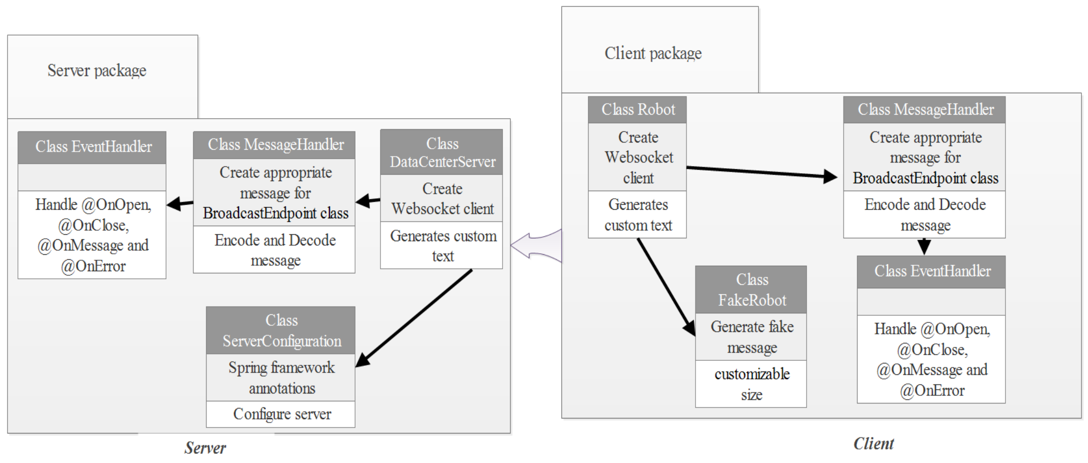

Server package
Class EventHandler
Handle @OnOpen, @OnClose, @OnMessage and @OnError
Class MessageHandler
Create appropriate message for BroadcastEndpoint class
Encode and Decode message
Class DataCenterServer
Create Websocket client
Generates custom text
Class ServerConfiguration
Spring framework annotations
Configure server
Server
Client package
Class Robot
Create Websocket client
Generates custom text
Class MessageHandler
Create appropriate message for BroadcastEndpoint class
Encode and Decode message
Class FakeRobot
Generate fake message
customizable size
Class EventHandler
Handle @OnOpen, @OnClose, @OnMessage and @OnError
Client

Figure 4.5: W implementation summary

As the summary presents in Figure 4.5, our client consists of four classes. Apart from *FakeRobot*, *Robot* is the main class that creates and configures Websocket client. Here, like our previous implementations, *fakeRobot* is used in order to generate the customized message with the desired size. It is worth mentioning that *Java.Thread* is used to manage time interval of generating messages. As it was explained earlier in Section 2.2.4, (according to Websocket API) this protocol is handling the events by some annotations and *EventHandler* plays the role of event handler. This class uses *MessageHandler* to create appropriate messages and handle *@OnOpen* and *@OnMessage* events. *MessageHandler* is used in *EventHandler* in order to create messages. In fact, *MessageHandler* is responsible to increase the security of data transmission by encoding and decoding the desired messages.

Our server implementation consists of four classes. Two of them have almost the same functionality as client's classes. *DataCenterServer* is the main class that contains the configuration of Spring framework and setup the server. *EventHandler* is using annotations to handle the events and *MessageHandler* creates the appropriate message for *EventHandler*. Moreover, we used Spring framework annotations in *ServerConfiguration* in order to configure the server.

5 Evaluation

This chapter is devoted to our experiments and evaluation. We divided this chapter into five sections. In Section 5.1 we will explain about our evaluation metrics. The first experiments will be presented in Sections 5.2 where we will start to understand protocol's behavior in detail by sending one message using each protocol. A summary and conclusion of the first experiment will be discussed in Section 5.3 and subsection 5.3.1, respectively. Later in subsection 5.4 we will continue our experiment by changing the size of messages and time-interval of generating messages, respectively. Finally, Section 5.5 will present a summary of our experiments in Section 5.4.

5.1 Evaluation Scenario

Our evaluation scenario was to primarily focus on each protocol individually by sending one message using each protocol. Later on, we checked the protocols behavior under stress by sending a large number of messages (as stream) through these protocols. Also we increased the size of messages and decreased the interval time of sending-message, respectively. Then we measured our defined KPIs for each protocol under mentioned conditions to find which protocol behaves better at message transfer considering time, bandwidth usage and interruption during the data transmission. The results that will be analyzed later in this chapter have been generated by a number of standard network monitoring and analysis tools.

It is noteworthy that our message is a text (the result is the same as that of JSON format) with the average size of newspaper articles which equals to 1592 bytes (calculated by java.lang.String.lenght). The logic behind this is that the typical message size of the robot application that Linnaeus University is planning to build is 1592 bytes and we used the same message size in our experiments.

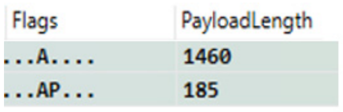
5.2 First Experiment: Single Message Transfer

In this experiment we evaluate the result for sending one message. We start with describing the experiment in detail for Websocket and then present a summary of the same experiment for other protocols in next sections.

At the beginning we sent one message by Websocket protocol to check the results regarding delta-time and payload length. In order to calculate the payload length and delta-time we needed to do calculations. This is because the size of message is a bit big and also more than one packet of data is involved in transferring one message (the relation is demonstrated Table 5.1). The payload length was the same during all of our five experiments. However, regarding the delta-time calculation, we repeated this experiment five times because the results of delta-time were slightly fluctuating (Figure 5.3) in milliseconds and we decided to conclude based on the average of the figures.

According to Figure 5.1 which has been generated by Message Analyzer, the payload length for sending one message is the product of sum of 1460 bytes and 185 bytes which equals 1645 bytes (the relation is shown in Table 5.1 and based on Figure 5.1 the flag A is ACK and AP is ACK-PUSH which respectively mean the traffic is accepted and push data). This is because those two packets contain the messages that are transmitted by Websocket and also the standard maximum transmission unit (MTU) is 1500 bytes [27].

It should be mentioned that all four protocols are the same as Websocket and use TCP packets in transport layer to transfer the message. The results indicate an overhead size of 3.2% for one message that is generated by Websocket protocol (the relation is shown in Table 5.2).



Flags	PayloadLength
...A....	1460
...AP...	185

Figure 5.1: payload length to Send one message by Websocket Connection

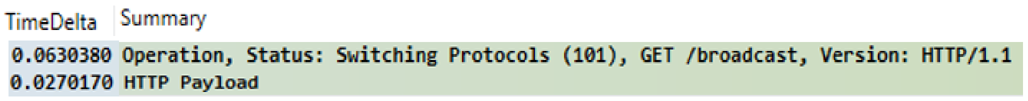
Payload length first message		*Payload length second message*		*Payload length to send one message in Websocket*		
Variable	Value	Variable	Value	Variable	Formula	Value
P_f	1460 bytes	P_s	185 bytes	P_{WS}	$P_{WS} = P_f + P_s$	1645 bytes

Table 5.1: payload length calculation

Actual message size		*Payload length to send one message in Websocket*		*Overhead of Websocket for one message*			
Variable	Value	Variable	Value	Variable	Formula	Value	Value percentage
M_s	1592 byte	P_{WS}	1645 byte	O_{ws}	$O_{ws} = P_{WS} - M_s$	53 byte	3.22 %

Table 5.2: Overhead calculation

In order to calculate delta-time (Figure 5.2) we should sum up the time needed for HTTP handshaking (Switch the protocol that contains request and response between client and server as Explained in Subsection 2.2.4) and the time spent to transfer actual message (the relation is shown in Table 5.3 – generated by Message Analyzer).



TimeDelta	Summary
0.0630380	Operation, Status: Switching Protocols (101), GET /broadcast, Version: HTTP/1.1
0.0270170	HTTP Payload

Figure 5.2: Websocket upgrade-protocol message (Stack view) – generated by Message Analyzer

Delta-time to switch protocol		*Delta time to transfer message*		*Delta time to send one message by Websocket*	
Variable	Value	Variable	Value	Formula	Value
$\Delta T_{Switch\text{-}protocol}$	0.063	$\Delta T_{payload\text{-}transfer}$	0.027	$\Delta T_{Switch\text{-}protocol} + \Delta T_{payload\text{-}transfer}$	0.09

Table 5.3: Delta-time calculation

To convince ourselves that the above results is repeatable, we performed the same experiment five times. Then we calculated the average from the results (Figure 5.3).

First experiment	*Second experiment*	*Third experiment*	*Fourth experiment*	*Fifth experiment*	*Average*
0.087	0.105	0.101	0.099	0.09	0.0964

Figure 5.3: Average Delta-time of five-times experiment

As it is illustrated, the average is close to ΔT_{WS} that is calculated in (4). It is worth mentioning that the switch protocol transaction is happening once during a Websocket connection and it can be neglected during data streaming [13]. However, we decided to include this part to more clarify the delta-time for one message transmission. Moreover, we tried to check the bandwidth usage of this transmission and the results is shown in Figure. 5.4. This figure has been generated by Wireshark and the X-axis in it, shows the time in the scale of one second. The Y-axis shows the number of TCP packets that are involved in transmitting the message. The result demonstrates that about 10 TCP packets have been exchanged in order to transfer one message from the client to the server.

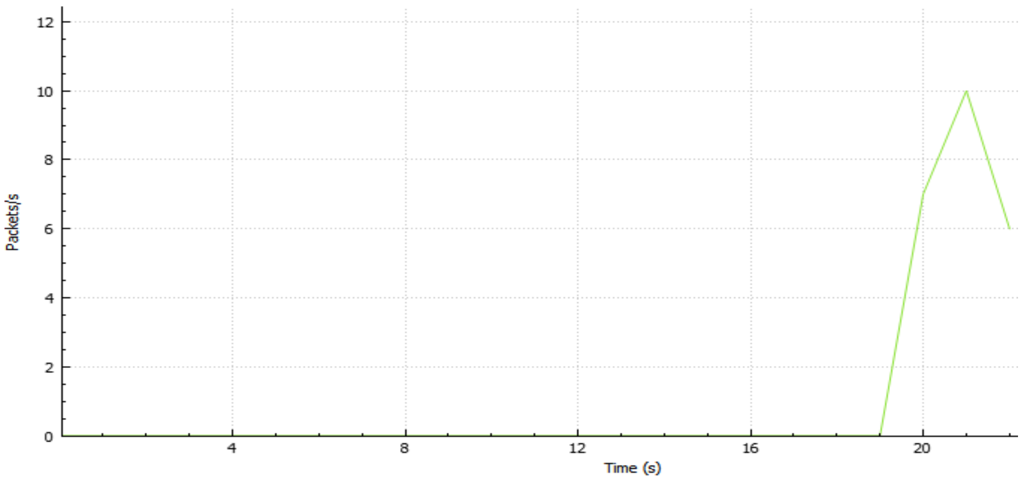


Figure 5.4: Bandwidth usage for 1 text message by Websocket (per 1sec)- Generated by WireShark

It should be mentioned that the only part that we are interested in, for bandwidth usage's graph, is the height of graph which represents the number of packets that are used. According to Websocket API, our client will receive event handler's messages in OnMessage and OnOpen conditions. We ignored OnError, since we had not any error. OnClose has also been ignored because it happens after receiving the message.

In the next step we followed the same scenario for other candidate protocols but since the process details are the same, we only present the results. We sent one message by TCP socket, HTTP1.1 and Server-Sent Events protocols and the results and summary of bandwidth usage, delta-time and payload length are presented in Section 5.3.

5.3 First Experiment's Summary

Protocols	*Time to receive one message(s)*	*Payload size (byte) – Overhead size (percentage)*	*Bandwidth usage (packets/s*
Server-Sent Events	0.10082	1732 – 8.08%	13
Websocket	0.0964	1645 – 3.22%	10
HTTP1.1	0.0093	1807 – 11.89%	11
TCP	0.0016	1597 – 0.31%	6

Table 5.4: Experiment results of transferring one message

5.3.1 First Experiment's Conclusion

The results demonstrate (Table 5.4) that since TCP is an underlying protocol for HTTP1.1, Websocket and Server-Sent Events [13,2,3], the payload size, bandwidth usage and Delta-time to receive the message for TCP were less than other protocols and therefore it performs better compared to other candidates. It should be mentioned that Websocket header in handshake procedures is only two bytes which is smaller than HTTP1.1[31]. Also the size of header in Websocket protocol affected its payload size and bandwidth usage, just as in HTTP1.1 and Server-Sent Events. Since we could find about protocols performance by transferring one message in previous experiment and this thesis concentrates more on data streaming, we continued our experiments as detailed in section 5.4, by transferring text-stream using each candidate.

5.4 Second Experiment: Stream Message Transfer

Previously, in the first experiment we checked the protocols' behavior for each message to understand how each protocol structure affect data transmission when transferring one message. This part will present the experiment's result of sending text message continuously as stream. We divided this experiment to three sub-sections. We continued the experiments by using the candidates to send text-stream while increasing the size by 10 times and decreasing the time-interval of sending messages. In fact, in this part we put the protocols under stress of text-streaming to check the results and understand about the performance of each protocol.

In all three sections we visualized the bandwidth usage and interruption during the data transmission by *Stevens-graph*. Via Stevens-graph we can analyze and find the interruptions clearer. A flat area in Stevens-graph (Figure. 5.5) means an interruption which can be generated by ACK or double ACK. A gap in graph means an interruption that may happen because of data loss or lack of data to transmit and Stevens-graph will grow faster if the number of data transmissions increases. In Stevens-graph, X-axis shows time and Y-axis is TCP sequence number that represents bytes sent and increases by each one byte of TCP data sent [25].

5.4.1 Sending Messages as Stream

In this section we started to send messages as stream. First we visualized Stevens-graph for Websocket as in Figure. 5.5 in which X-axis shows the time in seconds and Y-axis shows the TCP sequence number.

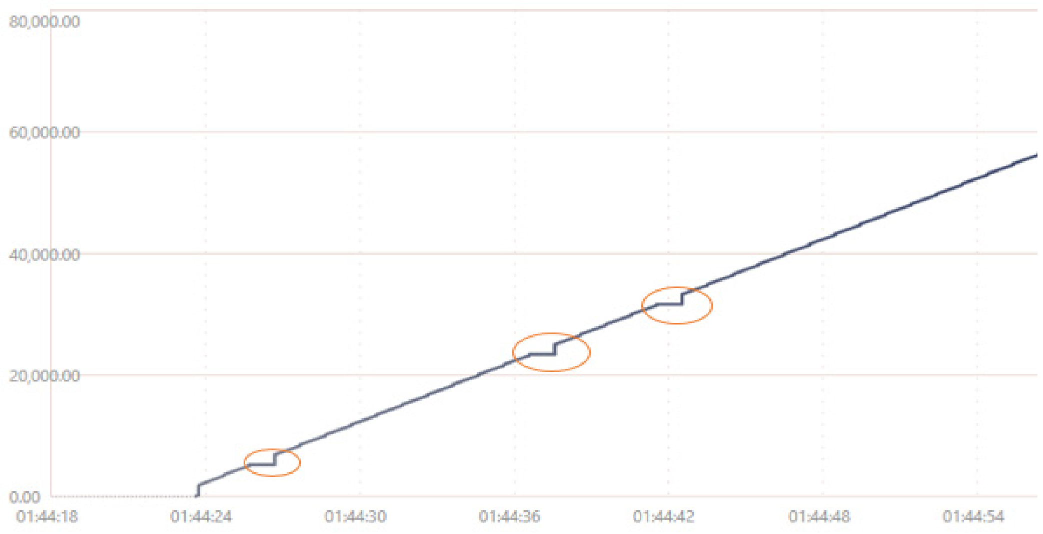


Figure 5.5: Interruption during the stream-data transmission by Websocket – Generated by Message Analyzer

We start to show the result for Websocket to become more familiar with Stevens-graph and what we can understand from this graph (Figure. 5.5). In generated Stevens-graph for Websocket (Figure. 5.5) we noticed the interruptions in transmission. However, there is not any gap during the transmission and it means we did not have any data loss. To compare the amount of messages transmitted by each protocol we generated Figure 5.6 through which we can indicate and compare the growth rate for each protocol.

Server-Sent Events grows a bit faster than other protocols and as it was explained earlier, it means that the number of TCP packets that are transmitted via Server-Sent Events is more than other protocols.

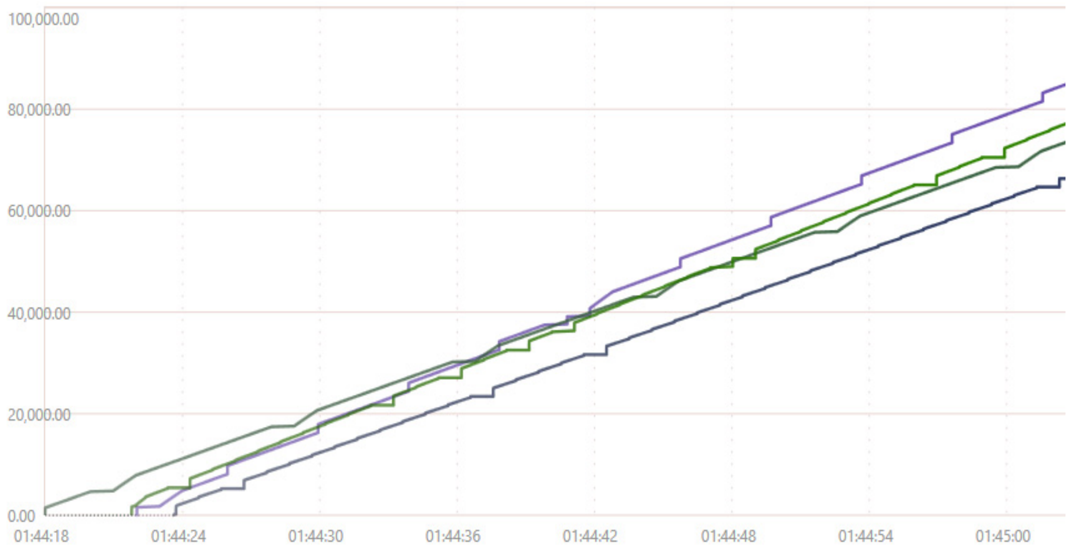


Figure 5.6: Stevens Graph for Websocket ■ Server-Sent Events ■ HTTP1.1 🟩 TCP ■

HTTP1.1 uses request and response to transmit the data. Based on the Figure. 5.6, the interruption for HTTP1.1 is more than other protocols because a time period should be spent to send request to the server and receive the answer. However, the amount of bytes that is sent via HTTP1.1 is greater compared to Websocket and TCP.

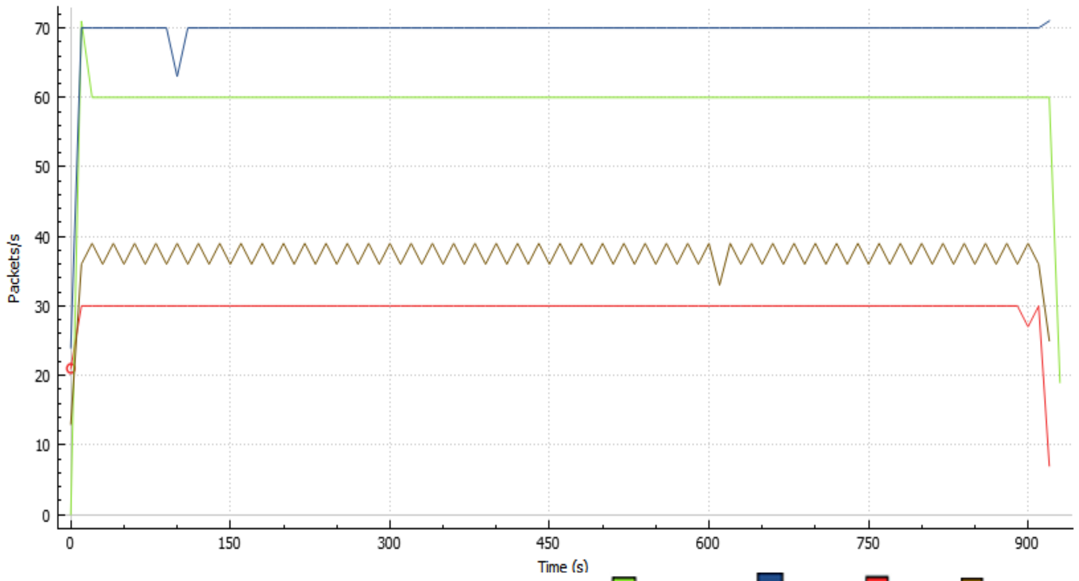


Figure 5.7: Bandwidth usage Websocket 🟩 HTTP1.1 🟦 TCP 🟥 SSE 🟫

Figure 5.6 illustrates that the interruption for the TCP protocol is less than other protocols and Websocket is the one which is a bit similar to TCP, interruption wise. Exchange request and response in HTTP1.1 per each message not only increased the interruptions in data transmission, but also affected bandwidth usage (Figure. 5.7) by using more bandwidth resources compared to other protocols. In Figure 5.7, Y-axis shows the number of packets that are exchanged to transmit the messages and X-axis shows the time in seconds. It is worth mentioning that in all of bandwidth usage's figures in this study we limited the time to ten seconds to be more focused on details. As it is mentioned in Chapter 2, Websocket API uses massage in full-duplex manner to update client about the condition of data transmission (Event handling), but the bandwidth usage of Websocket is not as much as HTTP1.1 which needs request and response per each message transmission.

On the other hand, Server-Sent Events needs at least one HTTP request from the client to start data transmission [23] and that is the reason why the bandwidth usage of Server-Sent Events is less than Websocket and HTTP1.1, but not less than TCP which needs only a three-way handshaking to start data transmission [18]. Actually, Websocket, Server-Sent Events and HTTP1.1 are not real transport protocols and use TCP to transfer the data which leads them to have additional high level handshaking compared to TCP [13].

5.4.2 Increased Message Size

In next step of experiment, we increased the stress level by increasing the size of each message by ten-times. In fact, we transferred big size of text-stream using those protocols and concentrated on their performance regarding bandwidth usage and interruption during the transmission. Since the size of article on website can be big and the text which will be generated by robots can be different, so we should check the protocols to understand their behavior in case of transferring big size text-stream.

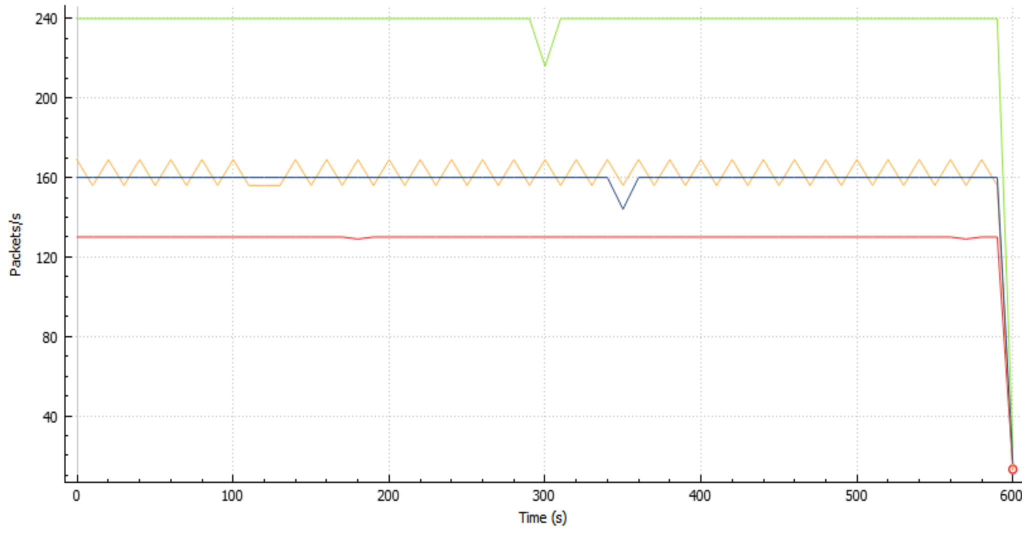


Figure 5.8: Bandwidth usage Websocket 🟩 HTTP1.1 🟦 TCP 🟥 SSE 🟫
(Message size increased 10 times)

The results of bandwidth usage visualization are presented in Figure. 5.8. The X-axis illustrates the time in seconds while the Y-axis shows the packets that use bandwidth and are exchanged to transfer the messages. Figure. 5.8 demonstrates that increase in the message size has affected Websocket more than other protocols. Websocket, with its event handling mechanism, exchanges more packets in order to transfer a big text streaming while TCP, due to the three-way handshaking mechanism, still has the minimum bandwidth usage. To get more detail about the relation between bandwidth usage and number of transmitted TCP packets we visualized Stevens-graph.

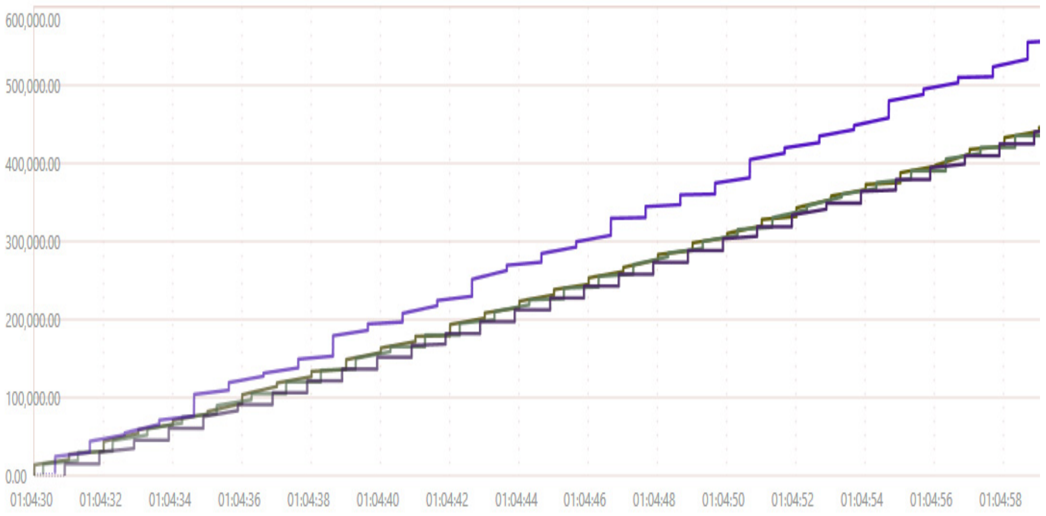


Figure 5.9: Stevens-graph for HTT1.1 ■ Websocket ■ Server-Sent Events ■ TCP ■
(Message size increased 10 times)

The Stevens-graph is presented in Figure. 5.9 in which the X-axis shows time in seconds, and the Y-axis shows the number of TCP packets that are exchanged to transfer the messages. The interruption during the transmission in Server-Sent Events is still less than other protocols and its graph grows faster which means more messages are transferred via Server-Sent Events protocol (Fig. 5.9).

On the other hand, HTTP1.1 still has more interruptions compared to other protocols. Websocket and TCP are two protocols that interact in a rather similar way towards increasing the message size. Also, as it is shown in Figure. 5.9, TCP and Websocket transferred the messages a bit faster than HTTP1.1 because their graph grew a little faster compared to HTTP1.1.

5.4.3 Decreased Time-Interval

In the final step of the experiment, we decreased the send-message interval to 1 millisecond (Figure. 5.10) and visualized the bandwidth usage and Stevens-graph, respectively. Figure. 5.10 illustrates the bandwidth usage of protocols in which the

X-axis shows the time in seconds, and the Y-axis represents the number of TCP packets that are exchanged by each protocol.

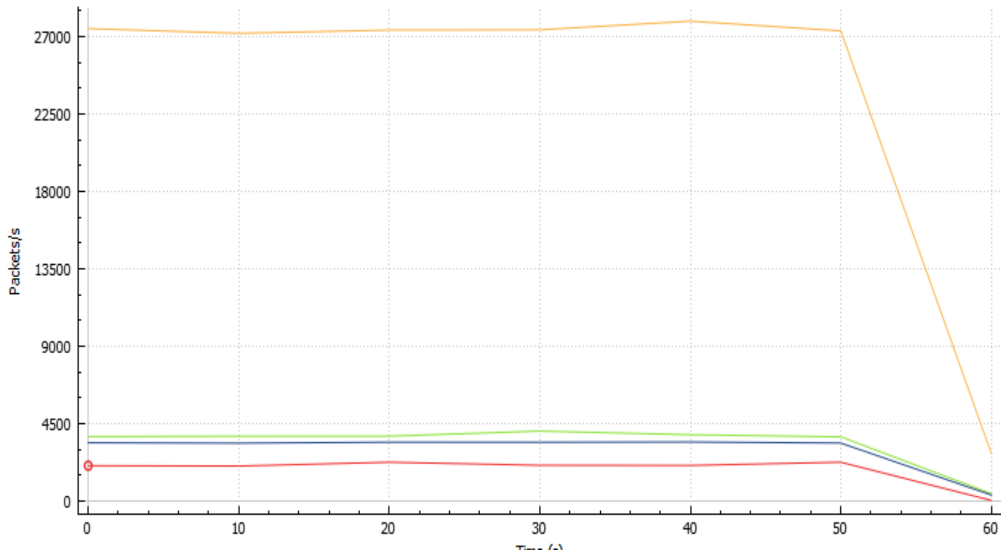


Figure 5.10: Bandwidth usage Websocket 🟩 HTTP1.1 🟦 TCP 🟥 Server-Sent Events 🟫
(Time interval decreased to 1 millisecond)

Figure. 5.9 shows the generated Stevens-graph. As it is explained at the beginning of this chapter, the X-axis represents the time in seconds and the Y-axis represents TCP sequence number.

In Figure 5.10 we noticed that bandwidth usage of Server-Sent Events is significantly more than that of other candidates. Stevens-graph (Figure. 5.11) proved that the Server- Sent Events transferred more TCP packet of data (messages) compared to other candidates since the generated Stevens-graph by Server-Sent-Events grew noticeably.

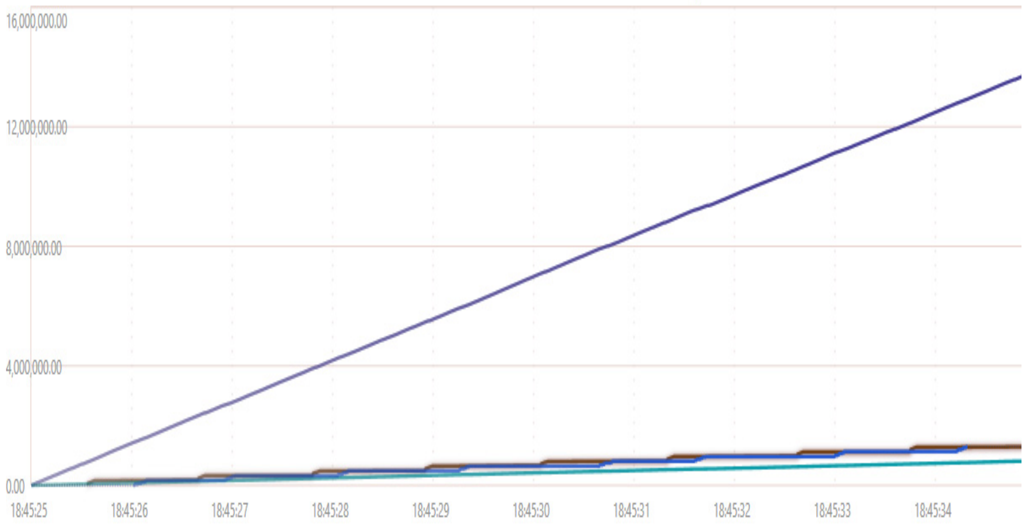


Figure 5.11: Stevens-graph for Websocket 🟦 HTTP1.1 🟦 Server-Sent Events 🟪 TCP 🟫
(Time interval decreased to 1 millisecond)

By using Server-Sent Events protocol, the server needs only one request from client to start unidirectional streaming from the server [23]. The results prove the efficiency of this protocol in transmitting fast unidirectional text-stream. The traditional TCP protocol that only needs three-way handshaking to start data transmission [18], still has minimum bandwidth usage (Fig. 5.10) while interruption during data transmission is not much and is almost the same as Websocket (Figure. 5.11). It has also almost the same number of TCP packet transmission (message) (Fig. 5.11) as Websocket. Bandwidth usage of Websocket is a bit more than HTTP1.1 and TCP protocols (Figure. 5.10) because of event handling mechanism in Websocket API [30], but since its Stevens-graph grew with the same rate as TCP, the number of messages that are transmitted by Websocket is almost close to that of TCP protocol (Figure. 5.11). HTTP1.1 request and response mechanism proved inappropriate in case of transmitting fast text-stream. The bandwidth usage of HTTP1.1 is more than TCP protocol since its Stevens-graph grew slower than that of other protocols and it means the number of messages transferred using this protocol is less compared to other candidates (Figure. 5.11).

As it is explained earlier in Subsection 5.4.2, we limited the time in bandwidth usage figures to ten seconds and in order to present the results in one second (Table 2), we divided the bandwidth values by 10. Below is an example:

Bandwidth usage of HTTP1.1 in Figure 14.6 in ten second: 70
Bandwidth usage of HTTP1.1 in one second = 70/10 = 7

The same scenario was used for all other protocols and the result is presented in Table 5.4. Also Table 5.5 indicates the results of Stevens-graph for one second. In order to produce the average for one second (Table 5.5) we divided the results of 30 second by 30. In the coming section we provide a summary about the protocols behavior in this experiment.

5.5 Second Experiment’s Summary

Table 5.4 indicates that HTTP1.1 used more bandwidth in Scenario 1, also it had less Stevens-graph’s value is Scenario 2 and 3 (Table 5.5) which means more interruption during the data transmission in the mentioned scenarios. On the other hand, TCP, as the traditional full duplex protocol to transfer the bi-directional data streaming, behaved acceptably in our experiments. The bandwidth usage of TCP in all stages was less than other protocols, also its Stevens-graph’s value was low which means the data interruption for TCP was close to Server-Sent Events and had the minimum amount of interruption. Websocket exchanges the messages further to actual data to increase the control on transmission and handle the events [17]. However, the Stevens-graph’s value for Websocket is close to TCP. As it is illustrated in Table 5.4, the bandwidth usage by Websocket is influenced by the size of the message. Based on Table 5.5 and summary of part 1 of the experiment, Server-Sent Events showed a better behavior compared to Websocket and HTTP1.1. This is because HTTP1.1 needs to handshake and send request for every new message and Server-sent Events is using only one handshake and request during the data transmission.

In Scenario 3 (Table 5.4) where the interval time is decreased to 1 millisecond, Server-Sent Events generated appropriate result.

Protocol	*Bandwidth Usage (packet/second)*		
	Scenario 1 Message size: Normal - Interval: 1sec.	*Scenario 2 Message size: 10 times - Interval: 1sec.*	*Scenario 3 Message size: Normal Interval: 1milisec.*
TCP	3	13	203
Server-Sent Events	3.9	17	2,700
Websocket	6	24	367.5
HTTP1.1	7	16	327.5

Table 5.4: summary of bandwidth usage (packet/second for one second) by each protocol.

Protocol	*Stevens-graph (TCP-sequence-number/second)*		
	Scenario 1 Message size: Normal - Interval: 1sec.	*Scenario 2 Message size: 10 times - Interval: 1sec.*	*Scenario 3 Message size: Normal Interval: 1milisec.*
TCP	1592	14,882	148,783
Server-Sent Events	1739	19,008	1,488,044
Websocket	1385	14,664	146,964
HTTP1.1	1556	14,166	88,633

Table 5.5: Stevens-graph results (TCP-sequence-number/second for one second) for each protocol.

6 Conclusion and Future Work

This chapter is the concluding chapter and is divided to two sub-sections. The conclusion of this study will be explained in Sub-section 6.1. Also a table that summarizes the results of this thesis and suggests the best suited protocol will be included in the same Sub-section. Sub-section 6.2 is dedicated to future work and the prototype plan about this thesis.

6.1 Conclusion

In this study we selected to use and focused on Websocket, TCP, Server-Sent Events and HTTP1.1 protocols that can be used to transfer text-streams based on the criteria defined by the project requirements. We divided our experiment process into two parts. In the first part, we got information about how the protocols' structure affect their behavior to transfer one message. For the same purpose we compared protocols in terms of delta-time, payload length and bandwidth usage. The results illustrated that TCP protocol performs slightly better than others. As the main purpose of this thesis is evaluating protocols for text-streaming, we continued the experiment process and in the second set of experiments we put the protocols under stress by transferring large number of texts, increasing the size of texts and decreasing the time interval (to force to transmit the data faster). Then we used RawCAP, Wireshark and Message Analyzer to monitor the network traffic and analyze the result of experiment.

Lack of access and information about implementation of scientist's clients in subscriber side that we discussed in Sub-section 1.2, and the specification of some protocols convinced us not to consider the experiment results for TCP in scientist sides and Server-Sent Events in publisher side. Because we don't have access to open any port for TCP in scientist side, the information about the client implementation in that side is not given as well (e.g. is a browser or standalone application). Furthermore, Server-Sent Events is a unidirectional protocol from server to client, only. Hence it is not possible to use this protocol in subscriber side. Based on the results of the experiments done in this research we are suggesting that the protocols be as follows:

Suggestion for publisher side in the order of priority (Trusted side):

1- TCP
2- Websocket
3- HTTP1.1

Based on the results of the experiments, TCP protocol as a traditional bi-directional and full duplex protocol behaved better than other protocols. It used less bandwidth while its interruption during the data transmission was also less than others. As it was explained earlier, publisher side is trusted and there is no firewall and restriction with regard to implementation information in this side. So TCP protocol is our first suggestion to be used in publisher side.

Websocket is our second suggestion for publisher side. One of its strengths is that it can use HTTP handshake and its advantages (Explained in Subsection 2.2.4). However, it is using protocol's information overhead (compared to TCP protocol). Also Websocket sends some extra messages (according to Websocket API) to increase control over data transmission. Such extra messages affect bandwidth usage and cause interruption in data transmission.

Our third suggestion for publisher side is HTTP1.1. Its request and response structure and larger header size (compared to Websocket) causes more bandwidth usage and less data transmission by this protocol.

Suggestion for subscriber side in the order of priority (Untrusted side):

1- Server-Sent Events
2- Websocket
3- HTTP1.1

The first suggestion for subscriber side is Server-Sent Event since its bandwidth usage and interruption in data transmission was less than other protocols and it means it could transfer more messages compared to Websocket and HTTP1.1. Server-Sent Events only needs one request and one handshake from the client to start the data transmission and that is the reason it behaved better, especially compared to HTTP1.1.

The second suggestion is Websocket because the experiment results indicate that its bandwidth usage and interruption in data transmission is more than Server-Sent Event. Especially when the size of message is increased and interval time is decreased.

HTTP1.1 is our third suggestion for subscriber side since its bandwidth usage and interruption during the data transmission was more than Server-Sent Events and Websocket. Table 6.1 summarizes the protocols in the order of priority as suggested in this thesis:

Part 1 Scientist side (Trusted side)	Part 2 Subscriber side (Untrusted side)
TCP	Server-Sent Event
Websocket	Websocket
HTTP1.1	HTTP1.1

Table 6.1: Protocol suggestion. The highest priority is on top.

6.2 Future work

Further studies will be needed to make first prototype of entire scenario based on Figure 1. If we had more time we could setup our version of Data Stream Centre and we would follow our suggestion and use TCP in part 1 and Server-Sent Events in part 2 (Table 6.1). Apart from the mentioned approach, future research can address stall problem which may happen when the speed of sending and receiving data in the source and destination are not the same. Also the experiments can be extended to clarify more about each protocol's behavior. It would be really useful if we had information about resource usage (for example CPU and RAM) for each protocol in order to help organizations about the hardware resources that need to be allocated to each protocol.

By increasing the use of satellite links it would be appropriate to have evaluations of wireless network to increase the information about the network's behavior on wireless network and understand how each protocol manage data transmission on that type of network.

These are all the limitation of this study which can be considered in the future and more comprehensive studies.

References

[1] B. Krishnamurthy, J. C. Mogul and D. Kristol, "Key differences between HTTP/1.0 and HTTP/1.1", *Computer Networks*, vol. 31, no. 11-16, pp. 1737-1751, 1999.

[2] S. Vinoski, "Server-Sent Events with Yaws", *IEEE Internet Computing*, vol. 16, no. 5, pp. 98-102, 2012.

[3] C. Toft, H. Persson, “HTTP/1.1 performance from an embedded perspective” M.S. thesis, Dept. Computer Science, BTH Univ., Stockholm, Sweden, 2004.

[4] H. Klevjer, K. Varmedal, A. Jøsang,” 'Extended HTTP Digest Access Authentication” M.S thesis, Dept. Informatics, Oslo Univ., Oslo, Norway, 2013.

[5] D. Gourley and B. Totty, *HTTP: The Definitive Guide*. Beijing: O'Reilly, 2002.

[6] R. Fielding, J. Gettys, J. Mogul, L. Masinter, P. Leach, T. Berners-Lee and H. Frysyk, *W3.org*, 1999. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616.txt. [Accessed: 15- Mar- 2016].

[7] R. Fielding, "RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1", *Tools.ietf.org*, 1999. [Online]. Available: https://tools.ietf.org/search/rfc2616. [Accessed: 03- Mar- 2016].

[8]"OSI", *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/OSI. [Accessed: 18- Feb- 2016].

[9] G. Muller, “HTML5 WebSocket protocol and its application to distributed computing” M.S. thesis, Dept. Engineering, Creanfield Univ., Cranfield, England, 2014.

[10] V. Wang, F. Salim and P. Moskovits, *The definitive guide to HTML5 WebSocket*. Berkeley, Calif.: Apress, 2013.

[11] S. Loreto, P. Saint-Andre, S. Salsano and G. Wilkins, "RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", *Tools.ietf.org*, 2011. [Online]. Available: https://tools.ietf.org/html/rfc6202. [Accessed: 04- Feb- 2016].

[12] D. Skvorc, M. Horvat and S. Srbljic, "Performance evaluation of Websocket protocol for implementation of full-duplex web streams", *2014 37Th International Convention On Information & Communication Technology, Electronics & Microelectronics (MIPRO)*, p. 1003, 2014.

[13] I. Fette and A. Melkinov, "RFC 6455 - The WebSocket protocol", *Tools.ietf.org*, 2011. [Online]. Available: http://tools.ietf.org/html/rfc6455. [Accessed: 21- Jan- 2016].

[14] P. Lubbers, B. Albers and F. Salim, *Pro HTML5 programming*. New York: Apress, 2010.

[15] M. Allman and A. Falk, "On the effective evaluation of TCP", *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, p. 59, 1999.

[16] A. Almasi, Y, Kuma, “Evaluation of WebSocket Communication in Enterprise Architecture” Bsc. thesis, Dept. Computer Science, Göteborgs Univ., Göteborg, Sweden, 2013.

[17] J. Hamerski, E. Reckziegel and F. Kastensmidt, "'Evaluating memory sharing data size and TCP connections in the performance of a reconfigurable hardware-based architecture for TCP/IP stack", *2007 IFIP International Conference On Very Large Scale Integration*, 2007.

[18] M. Del Rey, "TRANSMISSION CONTROL PROTOCOL", *Ietf.org*, 1981. [Online]. Available: https://www.ietf.org/rfc/rfc793.txt. [Accessed: 18- July-2016].

[19] A. Anzaloni and M. Listanti, "TCP performance over satellite networks", *Aerospace Conference, 2003. Proceedings. 2003 IEEE*, vol. 1, pp. 1 - 449, 2003.

[20] S. Vinoski, "Server-Sent Events with Yaws", *IEEE Internet Computing*, vol. 16, no. 5, pp. 98-102, 2012.

[21] A. Bersvendsen, "Dev.Opera — Event Streaming to Web Browsers", *Dev.opera.com*, 2006. [Online]. Available: https://dev.opera.com/blog/event-streaming-to-web-browsers/. [Accessed: 18- Mar- 2016].

[22] L. Hickson, "Server-Sent Events", *W3.org*, 2012. [Online]. Available: https://www.w3.org/TR/eventsource/. [Accessed: 18- Apr- 2016].

[23] E. Estep, "Mobile HTML5: Efficiency and Performance of webSockets and Server-Sent Events", *Nordsecmob.aalto.fi*, 2013. [Online]. Available: http://nordsecmob.aalto.fi/en/publications/theses2013/thesis_estep/. [Accessed: 04- Mar- 2016].

[24] R. Gregor, "HTML5 Server-Push Technologies, Part 1 Blog | Oracle Community", *Today.java.net*, 2010. [Online]. Available: https://today.java.net/article/2010/03/31/html5-server-push-technologies-part-1. [Accessed: 18- Feb- 2016].

[25]"Wireshark · Go Deep.", *Wireshark.org*, 2016. [Online]. Available: http://www.wireshark.org/. [Accessed: 01- May- 2016].

[26] R. STROBL, "NetBeans IDE", *Oracle.com*, 2016. [Online]. Available: http://www.oracle.com/technetwork/developer-tools/netbeans/overview/index.html. [Accessed: 02- Feb- 2016].

[27] M. Behnam, R. Marau and P. Pedreiras, "Analysis and optimization of the MTU in real-time communications over Switched Ethernet", *IEEE Symposium On Emerging Technologies And Factory Automation, ETFA*, no. 6059021, pp. 1 - 7, 2016.

[28]"RawCap - A raw socket sniffer for Windows", *Netresec.com*, 2015. [Online]. Available: http://www.netresec.com/?page=RawCap. [Accessed: 18- Jan- 2016].

[29]"Download Microsoft Message Analyzer from Official Microsoft Download Center", *Microsoft.com*, 2016. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=44226. [Accessed: 04- Feb- 2016].

[30] L. Hickson, "The WebSocket API", *W3.org*, 2012. [Online]. Available: https://www.w3.org/TR/websockets/. [Accessed: 18- Jan- 2016].

[31] Z. Lijing and S. Xiaoxiao, "Research and development of real-time monitoring system based on WebSocket technology", *Mechatronic Sciences, Electric*

Engineering and Computer (MEC), Proceedings 2013 International Conference on, pp. 1955 - 1958, 2013.

[32] [1]"TCP Stevens Graph", *Technet.microsoft.com*, 2016. [Online]. Available: https://technet.microsoft.com/en-us/library/dn799011.aspx. [Accessed: 03- Jun- 2016].

[33] L. Li and C. Wu, "Design and Describe REST API without Violating REST: A Petri Net Based Approach", *Web Services (ICWS), 2011 IEEE International Conference on*, pp. 508 - 515, 2011.

[34] R. Esteller-Curto, E. Cervera, A. del Pobil, R. Marin, I. You, L. Barolli, A. Gentile, H. Jeong, M. Ogiela and F. Xhafa, "Proposal of a REST-based architecture server to control a robot", *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 708 - 710, 2012.

[35] D. Keim, M. Krstajic, C. Rohrdantz and T. Schreck, "Real-Time Visual Analytics for Text Streams", *Computer*, vol. 46, no. 7, pp. 47-55, 2013.

[36] M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 requirements of real-time stream processing", *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42-47, 2005.

[37] J. Scotland, "Exploring the Philosophical Underpinnings of Research: Relating Ontology and Epistemology to the Methodology and Methods of the Scientific, Interpretive, and Critical Research Paradigms", *English Language Teaching*, vol. 5, no. 9, 2012.

[38]"Apache Storm", *Storm.apache.org*, 2016. [Online]. Available: http://storm.apache.org/. [Accessed: 24- Jun- 2016].

[39]"Redis", *Redis.io*, 2016. [Online]. Available: http://redis.io/. [Accessed: 24- Jun- 2016].

[40]"User Datagram Protocol", *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/User_Datagram_Protocol. [Accessed: 24- Jun- 2016].

[41]"HTTP persistent connection", *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/HTTP_persistent_connection. [Accessed: 25- Jun- 2016].

[42]"Quantitative research", *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Quantitative_research. [Accessed: 26- Jun- 2016].

[43]"WebSocket", *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/WebSocket. [Accessed: 28- Jun- 2016].

[44] Z. Li and Z. ZHANG, "25 - Real-time Streaming and Robust Streaming H.264AVC Video", *IEEE*, vol. 2244-0, pp. 353 - 356, 2004.

[45] K. OBANA, T. SHIMIZU, T. MUROOKA, K. HAYASHI and K. OGUCHI, "Evaluation of the MXQ Mechanism by Audio and Video Applications", *IEEE*, vol. 8601-9, pp. 514-518, 2004.

[46] T. Das and K. M. Sivalingam, "TCP improvements for Data Center Networks", *IEEE*, vol. 4673-5329-8, pp. 1-10, 2013.

[47] Ting Wang, Zhiyang Su, Yu Xia and M. Hamdi, "Rethinking the Data Center Networking: Architecture, Network Protocols, and Resource Sharing", *IEEE Access*, vol. 2, pp. 1481-1496, 2014.

[48] P. Zhao, J. Li, J. Xi and X. Gou, "A Mobile Real-time Video System Using RTMP", *IEEE*, vol. 4673-2981-1, pp. 61 - 64, 2012.

[49] U. Tos and T. Ayav, "Adaptive RTP Rate Control Method", *IEEE*, vol. 7695-4459-5, pp. 7 - 12, 2011.

[50]"The Streaming APIs | Twitter Developers", *Dev.twitter.com*, 2016. [Online]. Available: https://dev.twitter.com/streaming/overview. [Accessed: 08- Feb- 2016].

[51] T. Sai Gopal, R. Jain, R. Lakshmi Eswari P, J. G and S. Reddy Kamatham, "Securing UDT protocol: Experiences in integrating Transport Layer Security Solutions with UDT", *IEEE*, vol. 4799-0047-3, pp. 87 - 91, 2013.

[52]"WebRTC Home | WebRTC", *Webrtc.org*, 2016. [Online]. Available: https://webrtc.org/. [Accessed: 11- Mar- 2016].

[53] M. Mohammadnezhad, "powerofviva/mm222wn", *GitHub*, 2016. [Online]. Available: https://github.com/powerofviva/mm222wn. [Accessed: 25- Aug- 2016].

A Appendix 1

A.1 FakeRobot Class

We implemented fake robot to simulate the functionality of real robots. In all of our implementations, we used a class called *FakeRobot* to create the custom messages in the desired size. The mentioned class has a parameter that is taking the message size as integer. The implementation of *FakeRobot* is shown in Figure 7.1*:*

```
public class FakeRobot {
 int textSize;
String customText = "...." ;                                                   // custom big text

 FakeRobot(int messageSize){
 textSize = messageSize;
   }

 String NextMessage() {
 String fakeMessage = message;
 for (int i = 0; i <textSize -1 ; i++) {                                    // create message with desired
size
     fakeMessage = fakeMessage + message;
   }
JSONObject json = new JSONObject();                                          // create custom JSON
json.put("text: ", customText);                                              //put the big-text(our message)
in json
    ...
String fakeMessage = json.toJSONString();
return  fakeMessage;
   }
```

Figure 7.1: FakeRobot class – Used in all protocol implementation

This class contains a big text message which is named *customText* by default. It should be mentioned that the size of message can be increased to double, triple and more based on the requested size (textSize of 1 means the default message size, 2 means double message size, 3 means triple message size and it will continue with the same sequence). The desired message size will be stored in *textSize* and inside *NextMessage* method a for-loop is responsible to increase the message size according to the value that is specified in *textSize.* Then the message format will be converted to json structure and at the end, *NextMessage* method will return the message with the customized size.

A.2 Client and Server Using TCP Protocol

Our client implementation consists of two classes, *FakeRobot* and *Robot*. *Robot* plays the role of main class; the main part of its code is presented in Figure 7.2:

```
try{

FakeRobot fakeRobot = new FakeRobot(1);   // create an object from FakeRobot class by default size
String fakeMessage = fakeRobot.NextMessage();        // put the desired message in fakeMessage
socket = new Socket(ServerURL,serverPort);              // assign server URL and port to connect
                                          //setup an printwriter by using outputstream of socket object
PrintWriter printwriter = new PrintWriter(socket.getOutputStream(),true);
 while (true) {
   printwriter.println(fakeMessage);                                        // sent data to the server
   try {
   Thread.sleep(...);                                                   // set custom interval
   }
}
```

Figure 7.2: TCP-client's Robot class (main class)

As we explained earlier, we used *FakeRobot* class in order to create a big message by default size (size 1) and put the mentioned message in *fakeMessage*. Then we used *java.net.Socket* to create TCP socket with the desired IP and port. The *Printwriter* has been configured in order to transfer the *fakeMessage*. Also we used *Thread.sleep()* to manage the time intervals of message generation.

And the main part of our TCP server is shown in Figure 7.3:

```
try{

        serversocket = new ServerSocket(serverPort);                  //setup server port to listen on it
        socket = serversocket.accept();                           // wait till a client connect to the port
                                                        //set up communications to transmit text-data
        InputStreamReader inputstreamreader = new InputStreamReader(socket.getInputStream());
                                                        //use inputstream to establish a bufferreader
        BufferedReader bufferedreader = new BufferedReader(inputstreamreader);
        String readLine = "";
        boolean finished = false;
        while (((readLine = bufferedreader.readLine()) != null) && (!finished)){
          System.out.println("Received from Client: " + readLine);
          if (readLine.compareToIgnoreCase("CloseConnection") == 0) finished = true;
        }
```

Figure 7.3: TCP-Server's main class

When the server starts, it will listen to the defined port (*serverPort*) and allow the client to establish the connection. *Bufferedreade* uses *Inputstreamreader* to setup communication to receive the text (line by line) as long as the data is available and the line of message is not equal to *CloseConnection*. In fact, the connection will shut-down when the client sends a line that contains *CloseConnection* only. We handled all the exceptions for the server as well.

A.3 Client and Server Using HTTP1.1 Protocol

The simple implementation of HTTP1.1 client is shown in Figure 7.4:

```
 public static void main(String[] args) {                              // used TimerTask to manage time interval
TimerTask mytask = new JerseyClientPost();
Timer timer = new Timer();
timer.schedule(mytask, …, …);                                                     // time in milisecond
  }
@Override
public void run() {
try {
FakeRobot fakeRobot = new FakeRobot(1);  // create an object from FakeRobot class by default size
String fakeMessage = fakeRobot.NextMessage();        // put the desired message in fakeMessage
Client client = Client.create();                                                 // jersey HTTP client
                                                                   // specify server address,port and CRUD
WebResource webResource = client.resource("ServerURL:ServerPort");
                                                                   // post the data on HTTP REST server
ClientResponse response = webResource.type("application/json").post(ClientResponse.class,
fakeMessage);
   }
}
```

Figure 7.4: HTTP1.1-client's main class

In implementation of HTTP1.1 the direction of text streaming is from client to the server. Client implementation consists of two classes and we avoid repeating explanation about *FakeRobot.* We named the main client class as *Robot.* In this class *TimerTask* has been used to manage time interval and the FakeRobot creates the messages. As it is shown in Table 4.1, we used *Jersey* to create our HTTP client and send *post* request to the REST server. We specified the server's address and port to use *post* request which is sending the message to the server. At the end we handled the exceptions as well.

The main part of HTTP1.1 server is demonstrated in Figure 7.5:

```
public class DataCenterServer {
public static void main(String[] args) throws Exception {
Server jettyServer = new Server(ServerPort);                                  // Define server's port
                                                              // Create and manage serverlet
ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);
context.setContextPath("/");
jettyServer.setHandler(context);
ServletHolder servletHolder = context.addServlet(org.glassfish.jersey.servlet.ServletContainer.class,
          "/*");
servletHolder.setInitOrder(0);                          // set initialize holder, parameter and value
                                                     // get the canocical name of MessageHandler-
class
servletHolder.setInitParameter("jersey.config.server.provider.classnames",
MessageHandler.class.getCanonicalName());
try {
    jettyServer.start();                                                   // run the server
        }
      }
   }
}
```

Figure 7.5: HTTP1.1-server's main class

In the server side we used *Jetty* library to implement the server with two classes. First class is named *DataCenterServer* which is the main class that creates, configures and runs the server. The second class is named *MessageHandler* that uses @POST annotation to handle *post* request from the client and receive the message. In the *DataCenterServer* we setup server on a port that we want to start the data transmission. As it was mentioned earlier, we used *Jetty* to implement our server. We created and managed the servlet to configure the jetty server, also we set the initialize order to the default (0 means on used) and defined the canonical name of *MessageHandler* to communicate to.

The main part of Post method inside Controller-class is as Figure 7.6:

```
public class MessageHandler {
…
@POST
@Path("/post")          //in this thesis we are only using post action to post the message on the server
@Consumes(MediaType.APPLICATION_JSON)
public void createDataInJSON(String data) {
     …
   }
 }
```

Figure 7.6: HTTP1.1-server's MessageHandler class

This method is using *@POST* annotation to handle post requests from the client.

A.4 Client and Server Using Server-Sent Events Protocol

We used *Jersey* in order to implement Server-Sent events' client and server. The client code to setup a connection to the server is as Figure 7.7:

```
Client client = ClientBuilder.newBuilder().register(SseFeature.SERVER_SENT_EVENTS).build();
                                                        // set server information to connect
WebTarget target = client.target("http:// ServerIP: ServerPort /server-path/");
                                                        // request to get events from the server
EventInput eventInput = target.request().get(EventInput.class);
while (!eventInput.isClosed()) {
                                              //read events as long as there is event available
 final InboundEvent inboundEvent = eventInput.read();
        …
        }
```

Figure 7.7: Server-Sent Events'- client main class

In the codes above, the target is defined. By target we mean the server that we want to connect to receive the message (in Server-Sent Events data that is transferred is called event). *target* contains the necessary information that is needed to establish connection to the server such as server's IP and port. By knowing that information, client can send get-request to the server and a while-loop gets event from the server as long as the events are available; otherwise, the connection will be closed (if events equals to null).

The server side that answers the client's get-request consists of three classes and is using REST architecture. Apart from *FakeRobot* that explained earlier, *DataCenterServer* is responsible to configure and run the server on a given port and IP address, also the exception has been handled as well. Our simple implementation of Server-Sent Events in server side is as Figure 7.8:

```
public class DataCenterServer {
public static void main(String[] args) {
try {                                                    //Configure Server-Sent Events’
server
final ResourceConfig resourceConfig = new ResourceConfig(MessageHandler.class, SseFeature.class);
final HttpServer server = GrizzlyHttpServerFactory.createHttpServer(URI.create("http:// ServerIP:
ServerPort /"),     resourceConfig, false);
server.start();                                       //start the server on given IP and given port
Thread.currentThread().join();
      }
      }
   }
}
```

Figure 7.8: Server-Sent Events’- server main class

Another class named *MessageHandler* is responsible to generate the text and handle the *Get* request from the client. In this class we used *FakeRobot* like our other implementations to generate the custom text that is saved in *fakeMessage*. Then we used a thread to manage generating messages by sending *fakeMessage* within the desired time intervals. *MessageHandler* is implemented as Figure 7.9:

```
@Path("server-path ")
public class MessageHandler {
…
@GET
@Produces(SseFeature.SERVER_SENT_EVENTS)
public getMessageQueue() {
final EventOutput fakeMessage = new EventOutput();
FakeRobot fakeRobot = new FakeRobot(1);// create an object from FakeRobot class by default size
String fakeMessage = fakeRobot.NextMessage();        // put the desired message in fakeMessage
new Thread() {                                    // send the message inside a thread
public void run() {
try {
while (true) {
fakeMessage.write(new OutboundEvent.Builder().name("SSE-messages").data(String.class,
fakeMessage).build());
Thread.sleep(…);                                                  // manage time interval
       }
     }
   }
  }
}.start();
return fakeMessage;
}
```

Figure 7.9: Server-Sent Events’- MessageHandler class

A.5 Client and Server Using Websocket Protocol

Our client consists of four classes. Apart from *FakeRobot*, *Robot* is the main class that creates and configures Websocket client. As it is shown in following codes, a name for the client is defined, Also the necessary information to establish connection to the server such as server port and IP address have been assigned. Here, like our previous implementations, *fakeRobot* is used in order to generate the custom message with the desired size inside a while-loop. It is worth mentioning that *Thread* is used to manage time interval of generating messages. Main parts of our client implementation is as presented in Figure 7.10:

```
public class Robot {
public static void main( final String[] args ) {
String client = "WS_client";                                  // define client's name
final WebSocketContainer container = ContainerProvider.getWebSocketContainer();      // setup the client
final String uri = "ws://ServerIP:ServerPort/broadcast";
try( Session session = container.connectToServer( EventHandler.class, URI.create( uri ) ) ) {
while (true) {
  FakeRobot fakeRobot = new FakeRobot(1);           // create an object from FakeRobot class by default
size
  String fakeMessage = fakeRobot.NextMessage();                  // put the desired message in
fakeMessage
                                                                //send message with defined client's
name
  session.getBasicRemote().sendObject( new Message( client, fakeMessage) );
  Thread.sleep(…);                                              //manage time interval
        }
    }
…                                                               //catch and handle exceptions
  }
}
```

Figure 7.10: Websocket-client main class

According to Websocket API, this protocol handles the events by some annotations and *EventHandler* plays the role of event handler. This class uses *MessageHandler* to create appropriate messages and handles *@OnOpen* and *@OnMessage* events. It is once more emphasized that we only show the important parts of our implementation. Our *EventHandler* class is implemented as Figure 7.11:

```
@ClientEndpoint(encoders = {MessageHandler.MessageEncoder.class}, decoders =
{MessageHandler.MessageDecoder.class})
public class EventHandler {
@OnOpen
public void onOpen(final Session session) { … }
@OnMessage
public void onMessage(final Message message) { … }
}
```

Figure 7.11: Websocket-client EventHandler class

MessageHandler is being used in *EventHandler* in order to create messages. In fact, *MessageHandler* is responsible for increasing the security of data transmission by encoding and decoding the desired messages. *MessageHandler* code is implemented as Figure 7.12:

```
public class MessageHandler {
                                                         //constructor, getter and setter
public static class MessageEncoder implements Encoder.Text< Message > { // serialize object to string
     …                                                    // define encoder
@Override
public String encode( final Message message ) throws EncodeException {
   }
```

```
     …
 }
public static class MessageDecoder implements Decoder.Text< Message > {  // deserialize string to object
private JsonReaderFactory factory = Json.createReaderFactory( Collections.< String, Object >emptyMap() );
    …                                                                        // define dencoder
@Override
public Message decode( final String str ) throws DecodeException {
     …
     }
   }
}
```

Figure 7.12: Websocket-client MessageHandler class

Our server implementation consists of four classes. Two of them have almost the same functionality as client's classes and we will not present their codes. *DataCenterServer* is the main class that contains the configuration of Spring framework and setup the server. First, we defined the server port, then sat up server configurations by creating the root of Spring context and default servlet and at the end we started the server.

The main part of our server implementation is as presented in Figure 7.13:

```
public class DataCenterServer  {
public static void main( String[] args ) throws Exception {
Server server = new Server(ServerPort);                       //define server port
final ServletContextHandler context = new ServletContextHandler();
      …                                               //Create the 'root' Spring application context
final ServletHolder defaultHolder = new ServletHolder( "default", DefaultServlet.class );
      …                                                       //create default servlet
                                                      // create instance of Websocket container
WebSocketServerContainerInitializer.configureContext(context);
server.start();                                               // Start the server
    }
}
```

Figure 7.13: Websocket-server main class

EventHandler is using annotations to handle the events and *MessageHandler* creates the appropriate message for *EventHandler*. Moreover, we used Spring framework annotations in *ServerConfiguration* in order to configure the server. The annotations include @inject and @Bean which manage the dependency injection and backbone of application. Also @postContrust will be executed after dependency injection is done to configure the server. The *ServerConfiguration*'s main methods which should be implemented are as presented in Figure 7.14:

```
@Configuration
public class ServerConfiguration  {
@Inject private WebApplicationContext context;
public class SpringServerEndpointConfigurator extends ServerEndpointConfig.Configurator {
@Override
public < T > T getEndpointInstance( Class< T > endpointClass ) throws InstantiationException { … }
  }
@Bean
public ServerEndpointConfig.Configurator configurator() { … }
@PostConstruct
```

```
 public void init() throws DeploymentException { … }
}
```

Figure 7.14: Websocket-client MessageHandler class

The body of this methods can be different depending on the environment in which the server is to be implemented. But the necessary methods are the same for different environments and are similar to those we used in this thesis.