Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Péter Somogyi

Analysis of server-smartphone application communication patterns

Master's Thesis
Budapest, June 15, 2014

Supervisors:	Professor Jukka Nurminen, Aalto University
	Professor Tamás Kozsik, Eötvös Loránd University
Instructor:	Máté Szalay-Bekő, M.Sc. Ph.D.

Aalto University
School of Science

Aalto University School of Science Degree programme in Computer Science and Engineering	ABSTRACT OF THE MASTER'S THESIS

Author: Péter Somogyi

Title: Analysis of server-smartphone application communication patterns

Number of pages: 83	**Date:** June 15, 2014	**Language:** English
Professorship: Data Communication Software		**Code:** T-110

Supervisor: Professor Jukka Nurminen, Aalto University
Professor Tamás Kozsik, Eötvös Loránd University

Instructor: Máté Szalay-Bekő, M.Sc. Ph.D.

Abstract:

The spread of smartphone devices, Internet of Things technologies and the popularity of web-services require real-time and always on applications. The aim of this thesis is to identify a suitable communication technology for server and smartphone communication which fulfills the main requirements for transferring real-time data to the handheld devices.

For the analysis I selected 3 popular communication technologies that can be used on mobile devices as well as from commonly used browsers. These are client polling, long polling and HTML5 WebSocket. For the assessment I developed an Android application that receives real-time sensor data from a WildFly application server using the aforementioned technologies.

Industry specific requirements were selected in order to verify the usability of this communication forms. The first one covers the message size which is relevant because most smartphone users have limited data plan. The next part discusses reliability issues of the analyzed technologies covering WebSocket connection drop and proxy server caching. Latency tests are conducted as well and the final part discusses the security aspects and how the other requirements are affected when encrypted connections are used.

The results show that WebSocket and long polling are relatively good ways to deliver real-time information to smartphone devices. However, the WebSocket connection can drop unexpectedly due to the lack of keep alive messages which are generally not sent on the network. Moreover, latency radically increases when secured WebSocket connection is used.

Keywords: smartphone, server, real-time data, HTTP polling, long polling, WebSocket, message size, reliability, latency, performance, security

Acknowledgements

I would like to thank my supervisors Tamás Kozsik from Eötvös Loránd University and Jukka K. Nurminen from Aalto Unicersity, as they both gave me many advices and feedbacks for this Master's Thesis.

I would also like to express my deepest gratitude to my advisor, Máté Szalay-Bekő, for his continuous support, feedback and the discussions. Without his help I would never have been able to finish my thesis.

Finally, I would also like to thank my parents and sister, because they were always supporting me and encouraging me with their best wishes.

Abbreviations and Acronyms

3G	Third Generation Networks
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
HTML5	Hypertext Markup Language v. 5
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
LTE	Long-Term Evolution
OBD-II	On-Board Diagnostics v. 2
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol
WS	WebSocket
WSDL	Web Services Description Language
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Table of Contents

1. Introduction 8
   1.1. Internet of Things 9
   1.2. Challenges 10
   1.3. CarCare Service 12
2. Technologies 15
   2.1. AJAX 15
   2.2. REST 16
   2.3. Polling 18
   2.4. Comet/Long Polling 19
   2.5. WebSocket 21
3. Developed Applications 23
   3.1. Application Server 23
      3.1.1. Polling 23
      3.1.2. Long Polling 25
      3.1.3. WebSocket 27
      3.1.4. Deployment 28
   3.2. Android Demo Application 30
      3.2.1. Client Polling and Long Polling 31
      3.2.2. WebSocket 32
   3.3. Device Simulator 34
4. Assessment 36
   4.1. Message Size 36
      4.1.1. XML and JSON 36
      4.1.2. Polling 38

4.1.3. Long Polling 41
4.1.4. WebSocket 42
4.1.5. Estimations 49
4.2. Reliability 50
4.2.1. WebSocket Connection Drop 50
4.2.2. Network Type Change 52
4.2.3. Proxy Caching 53
4.3. Latency 54
4.3.1. Polling 55
4.3.2. Long Polling 57
4.3.3. WebSocket 58
4.3.4. Overview 60
4.4. Load Test 60
4.5. Security 62
4.5.1. TLS 63
4.5.2. Generating Self-Signed Certificate 66
4.5.3. Message Size 68
4.5.4. Latency 70
5. Conclusion 72
5.1. My Results 72
5.2. Future Works 74
5.2.1. Proof the Results 74
5.2.2. Scalability and Load Test 75
5.2.3. Other Technologies 75
5.2.4. Other Message Types 76
6. Appendices 77

6.1. Appendix A 77

7. Bibliography 79

1. Introduction

As a result of Internet of Things many devices share data with each other including our smartphones too. Homes are equipped with smart thermostats and automated lighting, vehicles can communicate with each other on the road or we even can receive notification to our mobile phones which road should we take to avoid traffic jams. There are many other fields which require devices or "things" to communicate with each other and with our smartphones. The fast spread of smartphones devices has changed the way we communicate and use the internet nowadays. The number of smartphone applications that require temporary as well as continuous network communication with a central server increased in the recent years [1].

However, selecting a suitable communication technology can be challenging based on the specific requirements of a field. The aim of this thesis is to identify a technology mainly for transferring real-time sensor data to smartphone devices with widely available technologies. In my Master's Thesis I will analyze 3 different communication technologies for server and smartphone communication, namely client polling, long polling and HTML5 WebSocket protocol. I will focus on the message size, reliability of the technologies, latency, load test and security options. For the analysis I will use an Android application and a Java EE application server. Furthermore, I will use a service as an example – called CarCare – which gathers telemetric and sensor data from cars on-board computer using the OBD-II interface focusing on one possible use case of this service.

In this first chapter I will identify the challenges with this communication patterns as well as explain the CarCare service that I will use as an example. The second chapter is a short explanation of the used technologies. The third chapter describes the application that I used for the assessments; this covers the Android application, an application server and a device simulator. The fourth chapter is the assessment of the 3 selected technologies based on 5 industry specific requirements. In the final chapter I outline my findings about the technologies and explain possible future works.

1.1. Internet of Things

“The Internet of Things (IoT) is a scenario in which objects, animals or people are provided with unique identifiers and the ability to automatically transfer data over a network without requiring human-to-human or human-to-computer interaction. IoT has evolved from the convergence of wireless technologies, micro-electromechanical systems (MEMS) and the Internet.

A thing, in the Internet of Things, can be a person with a heart monitor implant, a farm animal with a biochip transponder, an automobile that has built-in sensors to alert the driver when tire pressure is low -- or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network. So far, the Internet of Things has been most closely associated with machine-to-machine (M2M) communication in manufacturing and power, oil and gas utilities. Products built with M2M communication capabilities are often referred to as being smart.” [2]

It is expected that the number of network connections between devices will reach 50 billion by 2020. From this 50 billion connection 80% is predicted to be between things, not human-human or human-computer interactions. This great number of connection can produce huge amount of information which will generate new markets. IDC predicts that the market for big data will reach $16.1 billion in 2014.

Connected car is one of the top ten industries where Internet of Things evolves. A revenue forecast by Business Insider [3] claims that currently $25 billion revenue generated from connected cars and expected to double in the next 4 years. Internet will become common in cars, furthermore, the growth of smartphone apps that target cars and drivers will radically increase. As an example, Waze [4] – a social traffic application – reached 80 million registered users within 2.5 years.

The telecom and automotive industries are two huge markets that are approaching each other. Sync [5] – Ford's in-car communication system – has been implemented in more than 4 million cars in the US market since it was launched and plans to target 13 million drivers in the near future.

"Both the automotive and the telecom industries are at a historic crossroads," says 54-year-old William Clay 'Bill' Ford Jr. "It also happens that helping define the future of mobility is a great personal passion of mine. One hundred years ago, the automobile redefined personal mobility. Today, portable communication devices are redefining personal mobility. And I believe that in the future, we will redefine personal mobility together." [6]

Connected car is an umbrella term with many ongoing researches. For example, Google is working on a self-driving car with significant results in the last years [7]. Apple CarPlay [8] is another big step towards connected vehicles which tries to join cars with the activities you do on your smartphone, such as navigation, phone calls, music and a lot more.

1.2. Challenges

This relatively new field which is called Internet of Things generates great amount of data and communication between specific devices and servers. This data is typically processed locally or on the server side, however, the users generally want to access these gathered information, such as the temperature change in the apartment in relation with the outside temperature or the fuel consumption of the car on different routes. Due to the spread of smartphones and tablets we want to access these data on the go using any kind of handheld devices.

Generally these devices have limited capabilities, for example most of the time we use smartphones with mobile internet connection having limited data plan or unreliable connection due to travelling in areas with partial network coverage. On the other hand we still want to access this information on other devices too, such as our notebook with a browser. Thus, the used communication technology should not be specific to a device or a browser in order to reduce development costs and maintainability.

Selecting a suitable technology which fulfills specific requirements can be challenging. Since I focus on communication for smartphone devices the size of the transferred messages are relevant due to the limited network plans as well as to improve network performance on the server side. Security is usually fundamental for network communications, thus an appropriate technology should support secured communication as well. There are scenarios which require reliable communications. Most of the times reliability is crucial between the devices and the central server for example a health monitoring or security system, however, reliable connection can be also necessary between the smartphone device and the server as well. Latency can be also important in real-time specific applications. In addition to these, minimizing server loading is also essential in case high number of user access the system simultaneously.

According to these requirements I selected 3 technologies to analyze which can be suitable for transferring real-time data from server to smartphone devices, these communication technologies are client polling, long polling and WebSocket. In this thesis I try to analyze these technologies according to the aforementioned requirements.

In this thesis I will try to answer for the following questions:

- *What is an optimal communication technology to transfer real-time data to smartphone devices?*
- *Which communicating technology has the lowest network traffic?*
- *Which is the most reliable technology for real-time data transfer for smartphone devices?*
- *How security affects the analyzed technologies?*

1.3. CarCare Service

As an example I will use a service called CarCare to demonstrate and analyze the 3 communicating technology focusing on the above requirements. In this subchapter I describe the CarCare service.

Cars have a lot of data that can be valuable for the drivers, other users or even the government can benefit from the available data. The number of products and services radically grows which exploit this data and it is expected to increase in the following years.

CarCare project was started as an innovation idea at Ericsson Hungary and I would like to analyze different communication methods how can the server and smartphone communicate with each other in similar use cases. I am involved in the development since February 2014, mostly working on the server application and the data analytics software components.

Figure 1 – Novatel OBD-II device

The service uses telemetric and diagnostics information gathered from the car's on-board computer. To access the data, a small device (Figure 1) needs to be connected to the OBD-II connection in the car [9]. This is a standardized interface that is available in every car that was produced after 2001 in the European Union [10]. The main usage of this port is to access to the status of the various vehicle sub-systems mainly for diagnostic reasons. The accessible information includes but not limited to speed of the car, rpm, fuel consumption, trouble codes, engine coolant temperature and emission of the vehicle. The connected device may have some additional sensors to extend the range of collectible information; in our case it includes a GPS receiver, an accelerometer and a GSM module are also implemented into the equipment.

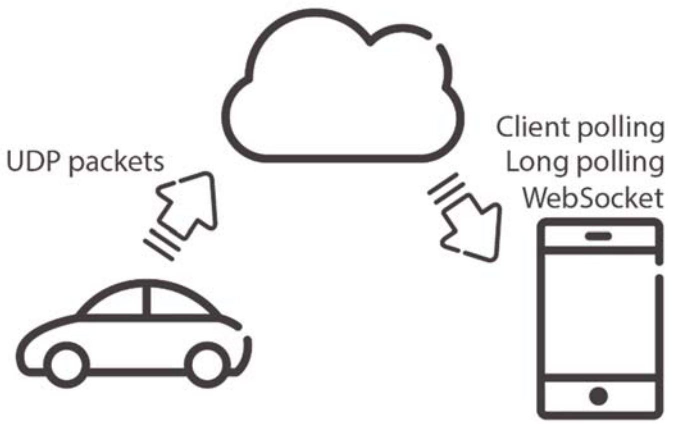


Figure 2 – Basic communication flow of sensor data

The device can be configured to continuously transfer specific data to the cloud, as well as notify the server based on some pre-defined rules. For example speed and GPS location information can be transferred continuously and the server can be informed in case the check engine alarm lights up (the yellow lamp on the dashboard indicating problems with the engine which needs to be investigated at a repair station).

In order to use the data, the smartphone application needs to retrieve it from the cloud server and the smartphone should visualize it for the user. To deliver this real-time information to the user's phone I can use different communication protocols. Currently, during the project we implemented long polling technology for this purpose; however, there are other existing technologies that can be suitable. For my master thesis I will analyze 3 different communication technologies, namely client-side polling, long polling and WebSocket, which is part of the HTML5 Connectivity area and defined in its specifications.

Many possible services can be implemented using this huge amount of available sensor and telemetric data; for example:

- **Driving feedback**

  The driver receives suggestions based on the driving style in order to reduce fuel consumption and drive more eco-friendly. The service can recognize incorrect gear shifting, unnecessarily high RPM, intensive breaking as well as quick turning.
- **Remote monitoring of the vehicle**

  The driver can see the car's position on a map based on the provided GPS location information as well as record and replay travelled routes.
- **Car diagnostics**

  Since the main purpose of this interface is to access diagnostic data related to the car, there are many data that can be helpful. The driver could be notified about trouble codes, engine oil level and any kind of information related to the car.
- **Fleet management**

  This field can also benefit from this service, mostly because of the location coordinates, but the diagnostic features can be also important for companies operating big fleets.

For my Master's Thesis I focus on the communication between the smartphone devices and a central server. I analyze different communication protocols that can be suitable for different kinds of messages that need to be transferred between the clients and the cloud in order to provide good user experience. My Master's Thesis does not focus on the communication process between the application server and OBD-II device located in the car.

2. TECHNOLOGIES

In this chapter I would like to briefly introduce the different technologies that I analyzed to support data delivery between the server and smartphone devices. The 3 main technologies that I will introduce are client polling, long polling and WebSocket protocol. Furthermore, I will introduce other technologies that are required to support the aforementioned communication models. I will analyze message size, reliability, latency, server-side load testing and security of these 3 communication technologies in Chapter 5.

Hyper Text Transfer Protocol (HTTP) is a protocol that was designed for request-response communication between the client and the server. I need HTTP for my thesis because the client polling and long polling technologies are purely based on the HTTP protocol, furthermore, the WebSocket protocol's opening handshake also requires an initial HTTP request and response. The form of the communication is that the client – in my case the smartphone application – submits an HTTP request and the server responds with the requested resources. With HTTP/1.0 a new connection was established to fetch every resource, as an improvement, HTTP/1.1 introduced the reusable connection [11]. After the connection was established, the client is able to download multiple resources, such as HTML page, images, using the same connection [12].

2.1. AJAX

Asynchronous JavaScript and XML (AJAX) is a set of technologies introduced by Jesse James Garrett. AJAX enables asynchronous, interactive web applications by providing more continuous communication with the server so that the client does not need to reload the entire page, only fetch specific resources that are required for a certain user action [13]. Due to the fact that I want to deliver real-time generated data to the smartphone, I need a continuous connection; therefore I need to use AJAX requests.

Asynchronous events mean that the action is performed in the background separately from the main execution process. These background events do not interrupt the flow of the web application. By this, AJAX provides the ability to the developers to deliver data between the browser and the server asynchronously in order to provide a more fluid user experience [14].

Most of the popular web services use AJAX to provide seamless interaction for the site's visitors. One great example is Google Maps where the users can zoom and move around on the map and the newly visited areas are fetched from the server in the background using AJAX-based requests providing a seamless user experience.

2.2. REST

Representational State Transfer (REST) is a software architectural style for designing stateless network applications which usually runs over standardized HTTP protocol. REST is a lightweight alternative of Remote Procedure Calls (RPC) and Web Services such as Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL) standards. REST was originally introduced and defined in Roy Fielding doctoral dissertation in 2000 at UC Irvine [15].

Most of the modern web-services provide REST API for HTTP-based communication. Due to its popularity I used REST architecture for the server application. Another alternative would be to use SOAP technology for delivering sensor data to the smartphone application. However, researches claim that REST-based implementation is proved to be more efficient in network bandwidth and latency than SOAP-based services [16].

The following are the key principles of the REST architecture: [17]

- Resources are identified with a unique URI (Uniform Resource Identifier)
  For example *www.mydomain.com/cars/23/* could identify a car with id 23 and *www.mydomain.com/cars* could return a list of all cars.
- Link resources together
  The different resources can link to each other; a car could have a link to the owner resource and a different one to the manufacturer's website
- Standardized operations
  CRUD [18] operations are supported for the resources: create, read, update and delete
- Multiple data representation
  The representation of the resource can be transferred between the server and the client in multiple formats. For example the client could retrieve the car's details in text format, XML, JSON or any other that both parties can understand.
- Stateless communication
  REST is stateless, although the application can have a state. A server should not have to retain some sort of communication state for any of the clients it communicates with beyond a single request.

Table 1 shows the connection between the standard HTTP methods and CRUD operations. The table also indicates which operations are safe and idempotent. Safe operation does not cause side effect, the client only retrieves data. In case the same operation can be executed multiple times and it results in the same state, then that operation is idempotent. Repeating different idempotent operations may result in different states. For example READ – DELETE – READ.

HTTP method	CRUD	safe	idempotent
GET	read	yes	yes
POST	create	no	no
PUT	update/create	no	yes
DELETE	delete	no	yes

Table 1 – HTTP methods in a typical RESTful API

The goal of the REST architecture is to provide a scalable system in order to support great number of components and simultaneous interactions with clients. REST also provides general interfaces, therefore, it supports interoperability. In addition to these, the architecture supports independent deployment of components as well as intermediary components to reduce latency, enforce security and encapsulate legacy systems.

2.3. Polling

Client side polling – also known as *HTTP polling* or just *Polling* – is one of the easiest ways to deliver real-time information to clients. I chose this communication technology to analyze due to its simple implementation and popularity. It is based on HTTP request and response messages to deliver the latest information. HTTP polling is a periodic, synchronous call established by the client to retrieve newly available information. The requests are made regularly and the server responds immediately with the most recent information.

Despite the fact that no new information is available, the server will send back a response with the most recent data or an empty content. Generally, real-time information is not so predictable and a lot of unnecessary connections are established between the server and client. As a result, many connections are opened and closed unnecessarily in low-message-rate situations causing big HTTP overhead, moreover, decrease network throughput as well [19].

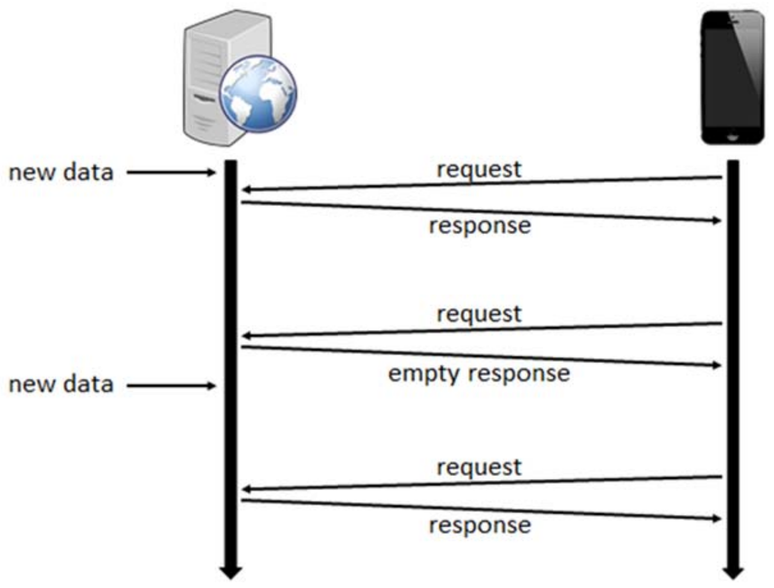


Figure 3 – Client polling sequence diagram

The polling interval should be configured based on the specific service's need. The possible refresh time can be varied from a few seconds to couple of minutes. In case the client wants to receive all update then the refresh interval should be shorter than the most frequent messaging rate. For example, if sensor data can arrive to the server in every 10 seconds and the client wants to be informed from all of them, then the polling interval should be lower than 10 seconds. Therefore, the client could retrieve all the information from the server-side. Although, this still does not guarantee that the client will get all information in case of network errors, moreover, high latency can also cause information loss.

2.4. Comet/Long Polling

Comet is a more sophisticated web application model that allows web servers to send data to the clients. Comet includes different technologies that focus on client-server interaction using common techniques [20]. I selected long polling communication technology for the assessment, because it similar to the client polling, however it aims to improve network bandwidth and latency.

Similar to the aforementioned HTTP polling, long polling connection is also based on HTTP requests and responses, however, the client initiated request is asynchronous and kept open until the server is able to answer to the request. As soon as the server receives the information – for example new sensor data arrives to the server – it is forwarded to the client and the server closes the connection. Thereafter, client immediately opens a new connection to receive upcoming updates. If new data is not received until the timeout, the server will respond to the request with and empty response and close the open HTTP connection. This flow makes it possible to for the server constantly send new data to the client as soon as it becomes available [21]. The long polling technique is described on Figure 4.

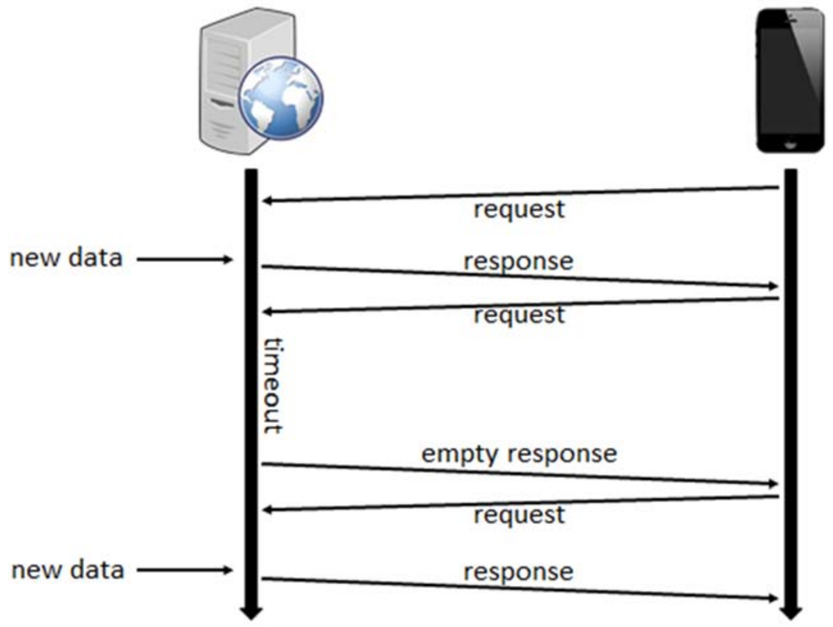


Figure 4 – Long polling sequence diagram

The main benefit of long polling compared to HTTP polling can be found in those cases when the server does not have the required data for a long time. Because the server blocks the asynchronous HTTP request, the client has to initiate new connection less frequently which reduces network traffic and server load.

The benefit of HTTP long polling mechanism is to allow the server to respond to a request only when a particular event or timeout has occurred. In order to minimize as much as possible both latency in server-client message delivery and the processing/network resources needed, the long polling request timeout should be a high value. However, high timeout values can cause problems: the client might receive a 408 Request Timeout or 504 Gateway Timeout. A recommended timeout is 300 seconds or lower, but most network infrastructures include proxies and servers whose timeout is shorter which can interrupt the connection [22].

2.5. WebSocket

For the third communication technology I selected WebSocket because it is a new technology that aims to provide a bi-directional communication channel with reduced message size, low latency. These characteristics show great potential for real-time applications. The WebSocket API is developed by the W3C (World Wide Web Consortium) and the WebSocket protocol by the IETF (Internet Engineering Task Force). The WebSocket API is now supported by modern browsers and includes methods and attributes needed to use a full duplex, bidirectional WebSocket connection. The API enables you to perform necessary actions like opening and closing the connection, sending and receiving messages [23].

The WebSocket protocol was designed to work together the existing web infrastructure. WebSocket is a full-duplex, bidirectional, single-socket connection. The WebSocket connection is initiated with a HTTP request from the client which contains a special Upgrade header. This indicates that the client wants to upgrade the HTTP connection to a different protocol, WebSocket. In case the server is able to communicate over WebSocket connection it sends back HTTP 101 SWITCHING PROTOCOLS message, thereafter the connection is upgraded to WebSocket reusing the same connection [24] [25].

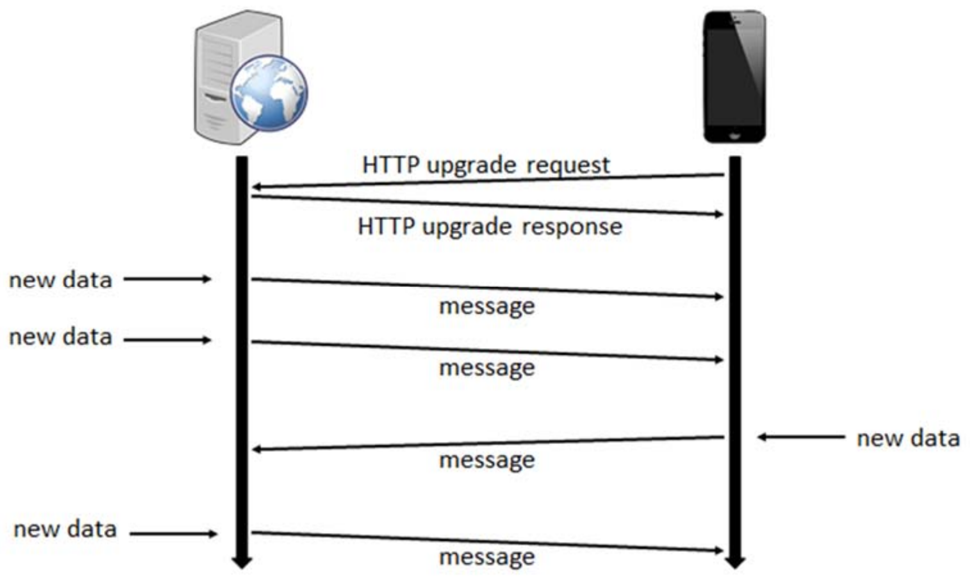


Figure 5 – WebSocket sequence diagram

The WebSocket API enables both parties to send and receive messages in a simple way using a full-duplex communication channel. An example communication flow with the opening handshake is visible on Figure 5. Using the API, we can send text-based and binary messages, in addition to this, WebSocket enables simple, secured communication using TLS protocol with the *wss://* URI prefix. It is supported to use higher-level protocols which run over the standard WebSocket connections. During the opening handshake the required protocol has to be passed to the server in the Sec-WebSocket-Protocol. One standard sub-protocol is called Extensible Messaging and Presence Protocol (XMPP) which is a message oriented, XML based protocol [26].

The main benefit of WebSocket is that it reduces latency because once the WebSocket connection is established, the server and client can send messages using the same connection without closing it. This single request greatly reduces latency over polling, which sends a request at intervals, regardless of whether messages are available. Because WebSocket was designed in a way to reduce bandwidth, the messages (frames) are represented on the network with binary syntax and minimal framing [23].

3. Developed Applications

For demonstration and testing purpose of the thesis, I have developed different applications using the technologies I covered in the previous chapter. One of these is an application server that receives incoming simulated telemetric data and sends this information towards the clients using the previously mentioned HTTP polling, long-polling and WebSocket technologies. In addition, a native Android app and few other applications were developed for measurement purpose.

3.1. Application Server

For the thesis I used WildFly version 8.0.0 Final [27] application server, formerly known as JBoss Application Server. This is the latest version of Red Hat's open source application server that supports Java Enterprise Edition 7 with Servlet 3.1 compatibility. This provides non-blocking I/O processing support for servlets, as well as HTTP protocol's upgrade functionality. These newly introduced features in WildFly were required for the asynchronous HTTP requests used in long-polling interface and the HTTP protocol's upgrade mechanism of WebSocket handshake.

I generated a self-signed SSL certificate for the WildFly server to allow the clients and server to communicate over a secured connection. It was needed to analyze the secure communication protocols as well, especially to use WebSocket over SSL/TLS (*wss://*) secure protocol. More information about the self-signed certificate and security can be found in Chapter 0.

3.1.1. Polling

Due to the fact that the server cannot notify and send the data to the client as soon as it arrives, the data has to be stored on the server side. There are two main ways how this can be achieved. One of them is caching the incoming data in memory and the other one is storing the data in database or file and retrieve it when the clients want to access it. For the implementation I selected the second option, because it provides a stateless service which is one of the main principles of REST APIs. The server stores the incoming sensor data using Java Persistence API and stores it in a H2 relational database system.

```
@Path("clientpolling/{carid}")
public class ClientPollingEndpoint {

    @Inject
    SensorDataDao sensorDataDao;

    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    @Path("/sensordata")
    public SensorData getLatestSensorData(@PathParam("carid") final long
            carId, @QueryParam("since") @DefaultValue("0") final long
            since) {
        return sensorDataDao.findLatestByCarIdSince(carId, since);
    }
}
```

Figure 6 – Polling interface

When a client fetches telemetric data for the selected car, the server retrieves the newest one from the database and sends it back in JSON format. The client can provide an optional query parameter for the REST request. In case it is provided, the server only sends back the sensor data that is newer than the submitted timestamp. If no data meets the requirements the server sends back HTTP 204 NO CONTENT response which means that the request was processed successfully but no data is available on the server side to return.

```
http://www.petersomogyi.com/rest/clientpolling/1/sensordata?since=1396464327
```

Figure 7 – HTTP polling REST interface

The implementation of the REST interface to retrieve the most recent sensor data using client polling is visible in Figure 6. For this I used Jersey [28] JAX-RS which is an API for developing RESTful web services in Java. It uses simple annotations which make the code more readable and maintainable. The annotations that I used for the client polling are explained in Table 2.

Annotation	Description
@Path	Specifies the relative path of a resource class or method
@PathParam	Binds a path segment to a variable
@QueryParam	Binds a query parameter to a variable
@DefaultValue	Specifies a default value to variable in case the key is not found
@GET	Specifies the request type of the resource
@Produces	Specifies the supported response media types

Table 2 – Used JAX-RS annotations

3.1.2. Long Polling

The long polling implementation (Figure 8) differs greatly from the client polling. Although it uses the same annotation as the other polling interface, however, in this case the server has to block the requests and respond to those in case sensor data arrives for the selected car.

```java
@Path("longpolling/{carid}")
public class LongPollingEndpoint {
    private static final long TIMEOUT = 30000;
    private static Map<Long, List<AsyncResponse>> waitingResponses = new
            ConcurrentHashMap<>();

    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    @Path("/sensordata")
    public void sensorDataRequest(@Suspended final AsyncResponse response,
            @PathParam("carid") final long carId) {
        response.setTimeout(TIMEOUT, TimeUnit.MILLISECONDS);
        response.setTimeoutHandler(new LongPollingTimeoutHandler());
        storeResponse(carId, response);
    }

    public void sendSensorData(final SensorData data) {
        final long carId = data.getCarId();
        final List<AsyncResponse> responses =
            waitingResponses.remove(carId);
        if (responses == null)
            return;

        for (final AsyncResponse resp : responses) {
            if (resp.isSuspended()) {
                resp.resume(data);
            }
        }
    }
}
```

Figure 8 – Long polling implementation

I store the incoming requests in a map connected with the car they want to subscribe to. When a new sensor data arrives, another interface calls the *sendSensorData* method which collects all the suspended requests for this car and responds with the newly arrived sensor data.

```java
public class LongPollingTimeoutHandler implements TimeoutHandler {

    @Override
    public void handleTimeout(final AsyncResponse asyncResponse) {
        final Response response = Response.noContent().build();
        asyncResponse.resume(response);
    }
}
```

Figure 9 – LongPollingTimeoutHandler class

The application sets the timeout of the incoming responses to 30 seconds and registers a *LongPollingTimeoutHandler* class which is a subclass of *TimeoutHandler*. My handler class overrides *handleTimeout* method which is fired when the associated response reaches the timeout. This method is responsible for sending HTTP 204 NO CONTECT response to the client after the timeout.

3.1.3. WEBSOCKET

For the WebSocket endpoint I used the *@ServerEndpoint* annotation which specifies the URI of the interface. To store the active sessions I used a Map similarly to the long polling implementation. A great benefit of WebSocket that predefined annotations can be used to override specific methods. These are shown in Table 3.

Annotation	Description
@OnOpen	This method is called when a new client connects to the server.
@OnClose	When a connection closes this method is called. The close reason can be accessed from the method parameter.
@OnMessage	It is used once a new message arrives to the server. Multiple methods can be annotated with this to handle text, binary, ping and pong messages differently.
@OnError	It is used to decorate a method which is called once Exception is being thrown by any method explained before.

Table 3 – WebSocket annotations

Figure 10 shows a part of the WebSocket endpoint implementation with the used annotations and method signatures [29]. When a new connection arrives to the server I add the session to the *connections* and remove it in the *onClose* method when the client disconnects. The *textMessage* and *binaryMessage* methods are responsible to handle the incoming messages and the errors are processed in the *onError* method. [30]

```
@ServerEndpoint("/websocket/{carid}/sensordata/")
public class WebSocketEndpoint {
    private static Map<Long, List<Session>> connections = new
              ConcurrentHashMap<>();

    @OnOpen
    public void onOpen(@PathParam("carid") final long carId, final
              Session session, final EndpointConfig endpointConfig) {
        addConnection(carId, session);
    }

    @OnClose
    public void onClose(@PathParam("carid") final long carId, final
              Session session, final CloseReason closeReason) {
        removeConnection(carId, session);
    }

    @OnError
    public void onError(@PathParam("carid") final long carId, final
              Session session, final Throwable thr) {
        ...
    }

    @OnMessage
    public void textMessage(@PathParam("carid") final long carId, final
              String message, final Session session) {
        ...
    }

    @OnMessage
    public void binaryMessage(@PathParam("carid") final long carId, final
              byte[] b, final Session session) {
        ...
    }
...
}
```

Figure 10 – WebSocket endpoint snippet

3.1.4. Deployment

During the testing I have deployed the server-side application to a remote server in order to achieve more realistic results for the tests. I selected Amazon Elastic Compute Cloud (Amazon EC2) [31] service which provides high scalability and seamless integration with other Amazon cloud-based services. Many popular web-services, such as Netflix, AirBnb, Adobe and Lionsgate, use

Amazon's cloud infrastructure due to its scalability, performance and good price structure.

"Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios." [31]

I selected the micro instance with Ubuntu Server 14.04. The server application is running on WildFly 8.0.0.Final using Java Enterprise Edition version 7. Since it is a free instance the server has limited capabilities. The used system's specifications during the tests can be found in Table 4:

Component	Specification
Processor	Intel® Xeon® CPU 2.00 GHz
Memory	0.613 GB
Hard drive	Amazon EBS storage
Network speed	~ 100 Mbps
Operating system	Ubuntu 14.04 x86-64

Table 4 – System specifications

3.2. Android Demo Application

For demonstration purpose I developed a basic Android demo application where the user is able to receive simulated sensor and telemetric data from the application server. Figure 11 shows the screen where the user can select the car and desired communication technology from the list *(WebSocket, Long polling and Client polling)*. After this the user is able to connect to the server, by default to the http://petersomogyi.com domain.

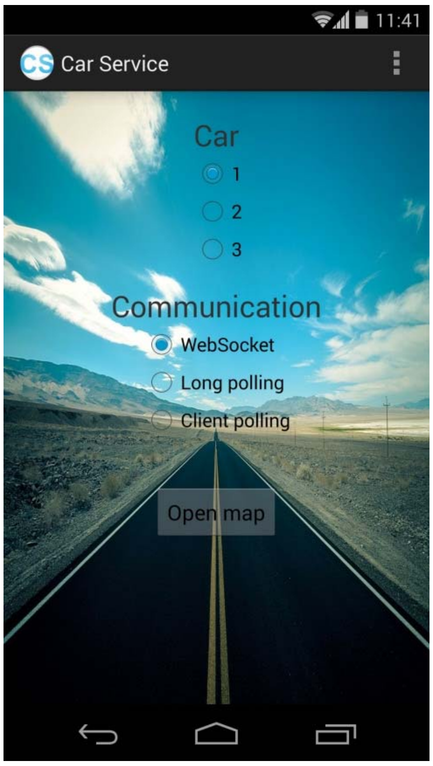


Figure 11 – Android demo app start screen

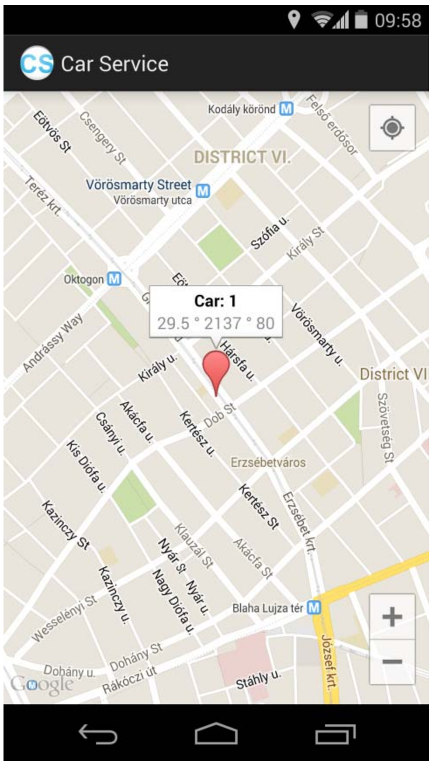


Figure 12 – Android demo app map screen

In case the application is connected to the server it receives the incoming data from the server using the desired communication method. As visible on Figure 12, a marker is displayed on the map with the car's current position as well as some basic data from the car are shown. These are speed, RPM and engine coolant temperature.

3.2.1. Client Polling and Long Polling

For the polling I start a new background thread which executes a HTTP GET request and retrieves the last sensor data from the server in JSON format. After the application receives the response the data I store the timestamp in a variable and for the next request I pass this value in the query parameter to retrieve only the newer sensor and telemetric data from the server. Between 2 requests I suspend the thread for 4 seconds.

```
final Thread t = new Thread() {

    @Override
    public void run() {
        while (active) {
            final String json = httpRequest(lastTimestamp);
            lastTimestamp = getLastTimestamp(json);
            applicationBusinessLogic.textMessage(json);

            try {
                sleep(UPDATE_INTERVAL);
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

};
```

Figure 13 – Client polling thread implementation

For the HTTP request I use the Apache DefaultHttpClient and set the User-agent and Accept headers. After successful execution I convert the received payload from input stream to JSON and send it back to the application which will display the data on the map.

```
private String httpRequest(final long lastTimestamp) {
    String json = null;
    HttpClient httpClient = new DefaultHttpClient();
    httpclient.getParams().setParameter(CoreProtocolPNames.USER_AGENT,
         System.getProperty("http.agent"));
    final HttpGet getRequest = new HttpGet(URL + lastTimestamp);
    getRequest.addHeader("Accept", "application/json");
    HttpResponse response;

    try {
        response = httpClient.execute(getRequest);
        if (response.getStatusLine().getStatusCode() == 200 && active) {
            final HttpEntity entity = response.getEntity();
            final InputStream instream = entity.getContent();
            json = convertToString(instream);
        }
    } catch (final IOException e) {
        e.printStackTrace();
    }

    return json;
}
```

Figure 14 – HTTP call for client polling

The long polling implementation is identical with the client polling, for the smartphone client it makes no difference that server replies to the HTTP call later. The only difference is in the background thread that sends the requests. In the long polling case the application does not have to wait between 2 calls to reduce the idle time, because the data that arrives during this time cannot be retrieved by the client.

3.2.2. WebSocket

In order to support WebSocket in a standard Android application I had to use a third party library. The one I selected is called Java WebSocket [32] which can be used to develop server and client applications, the library is written purely in Java using java.nio with MIT license. This same library can be used Java 1.5 applications as well as on Android devices with version 1.6 or higher. In a previous implementation I used Autobahn [33] to support the WebSocket protocol; however, it was not suitable for me due to its limitations. The current version does not implement the secure WebSocket protocol – although it is not written in the documentation – so this was the main reason why I had to replace it with Java WebSocket library.

Connecting to a WebSocket server is easy with the Java WebSocket library, only the URI of the server has to be passed to the constructor and the *onMessage, onOpen, onClose* and *onError* needs to be defined. These methods manage the incoming text messages as well as the events for opened and closed connection and error handling. The implementation can be seen in Figure 15.

```
WebSocketClient webSocketConnection;

webSocketConnection = new WebSocketClient(serverUri) {

    @Override
    public void onOpen(final ServerHandshake handshake) {
        applicationBusinessLogic.connected();
    }

    @Override
    public void onMessage(final String message) {
        applicationBusinessLogic.textMessage(message);
    }

    @Override
    public void onClose(final int code, final String reason,
      final boolean remote) {
        applicationBusinessLogic.disconnected(code, reason);
    }

    @Override
    public void onError(final Exception ex) {
        ex.printStackTrace();
    }
};

webSocketConnection.connect();
```

Figure 15 – Creating a WebSocket connection with Java WebSocket library

In the reliability chapter I will discuss the reasons why I needed to implement a background thread in the Android application. Here I just want to explain how I did it using the library. Figure 16 shows the thread that is responsible for sending the ping messages. Inside the loop the application sends a frame on the open connection with the *Opcode.PING* content and sleeps the thread for a predefined time which is passed as the constructor's parameter. I set the heartbeat interval to 5 seconds because I found out this is appropriate for cellular as well as Wi-Fi networks.

```
private class PingMessageHandler {

    private boolean active = false;
    private final Thread pingThread;

    public PingMessageHandler(final int interval) {
        this.pingThread = new Thread(new Runnable() {

            @Override
            public void run() {
                while (active) {
                    final FramedataImpl1 frame =
                         new FramedataImpl1(Opcode.PING);
                    frame.setFin(true);
                    webSocketConnection.getConnection()
                         .sendFrame(frame);
                    try {
                        Thread.sleep(interval);
                    } catch (final InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
}
```

Figure 16 – WebSocket ping thread implementation

3.3. Device Simulator

For the tests I created a console Java application which simulates the incoming sensor data from the device. Since the aim of my thesis is to analyze the communication between the server and the smartphone I did not emulate the behavior and packet structure of the OBD-II device.

```
time,lat,lon,elevation,accuracy,bearing,speed
2014-03-24T07:39:33Z,47.503816,19.064874,148.000000,55.000000,142.699997,13.000000
2014-03-24T07:40:42Z,47.501163,19.067571,156.000000,46.000000,143.399994,11.500000
```

Figure 17 – CSV file structure for the GPS coordinates

The application reads the GPS coordinates from a csv file which has to be passed as an argument to the application and should have the same structure as

Figure 17 shows. The application generates sensor data using the latitude, longitude, speed values from the CSV file. After that it sends the data in JSON format to the application server using an HTTP POST request to the *http://petersomogyi.com/rest/input* URL. The GPS data file and the car ID needs to be passed as a program argument and a third optional parameter can be added to specify the interval between the 2 sensor data.

```
java –jar devicesim.jar <gps-file> <carId> [<interval>]
```

Figure 18 – Start command for device simulator

4. Assessment

In this chapter I will analyze the 3 communication technologies that I selected and introduced previously – namely polling, long polling and WebSocket. For the assessment I will examine the network bandwidth, reliability of the technologies, latency, server-side load test and the effect in case secure connection is used. For these assessments I used WireShark for analyzing network traffic, the Java EE application server, the Android application that I developed for demonstration purpose and the mentioned testing tools.

4.1. Message Size

The size of transferred network packages make an important part in selecting an optimal technology for sever and smartphone communication. On the client side it is mainly important factor if the communication is running over mobile internet connection due to the frequently used data plans. In case the used network is Wi-Fi, the message size traffic is not a significant criterion.

On the other hand, smaller message size is relevant on the server side when it has to serve multiple clients simultaneously. For this reason, reducing network throughput by reducing the average size of the messages could be an important factor in selecting optimal communication protocol to increase the server's performance.

4.1.1. XML and JSON

The data that needs to be sent between the client and the server has to be serialized in order to send it over the network. There are two often used data serialization approaches in web applications, these are JavaScript Object Notation (JSON) and Extensive Markup Language (XML) [34]. In this part I briefly explain the 2 different representations and explain why I selected JSON format to serialize the data.

The Extensible Markup Language (XML) is a subset of the Standard Generalized Markup Language (SGML) and XML was designed due to the complexity of SGML [35]. It was designed for ease of implementation and for interoperability with both SGML and HTML. XML is used in many fields of computing due to its universal data representation format. It is widely used for remote procedure calls, describing web services using WSDL, configuration files, RSS and many other areas. The major design principles of XML include simplicity and human readability. The intent of an XML document is self-descriptive and embedded in its structure.[36] The tags in an XML document are not predefined, but specific for the application that understands them. Figure 19 shows a simple XML example.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sensordata>
    <carId>1</carId>
    <timestamp>1398112134064</timestamp>
    <gpsLatitude>47.53398</gpsLatitude>
    <gpsLongitude>19.079723</gpsLongitude>
    <speed>1.25</speed>
    <rpm>1250</rpm>
    <coolantTemperature>101</coolantTemperature>
</sensordata>
```

Figure 19 – XML message example

JSON (JavaScript Object Notation) is a lightweight text-based data-interchange format which is easy for humans to read and write and for machines to parse and generate. It is based on a subset of the JavaScript Programming Language [37]. It is primarily used to transfer data between server and clients in a compact format. JSON is estimated to parse up to one hundred times faster than XML in modern browsers, but it lacks of input validation and extensibility [38]. Figure 20 shows the same object represented in JSON format.

```
{
  "carId":1,
  "timestamp":1398112134064,
  "gpsLatitude":47.53398,
  "gpsLongitude":19.079723,
  "speed":1.25,
  "rpm":1250,
  "coolantTemperature":101
}
```

Figure 20 – JSON message example

Comparing the size, XML documents are bigger. In the above 2 examples using the same object XML needs 281 characters, however, to represent the same object using JSON it only takes 134. Moreover, parsing JSON messages is generally faster [38]. For my thesis I selected to use JSON format due to its compact data representation and performance.

4.1.2. POLLING

For the AJAX-based polling implementation there are two HTTP messages transferred for each messages, a request initiated by the smartphone and a response message sent by the server. Figure 21 shows the HTTP request sent by the Android application to retrieve the latest sensor data from the server running on *http://petersomogyi.com* host. The request describes the used HTTP/1.1 method which is GET as well as the resource's URL */rest/clientpolling/1/sensordata*. The Accept header specifies that the client can receive messages in JSON format. The size of this request is 285 bytes including the framing of the sent request. Most of the presented headers are generated by the Apache's *DefaultHttpClient* used in the Android implementation. It is possible to contain additional headers – like "Accept: application/json" – or override the default values too.

```
GET /rest/clientpolling/1/sensordata?since=1398112128995
HTTP/1.1

Accept: application/json

Host: petersomogyi.com

Connection: Keep-Alive

User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.2; Android
SDK built for x86 Build/KK)
```

Figure 21 – Client polling HTTP request example

The server replies with HTTP 200 OK answer and sends the available sensor data in JSON format in the response's payload. The size of the HTTP response including the TCP framing is 366 bytes. From this, the payload's size is only 134 byte so it means that the HTTP and TCP framing is 232 bytes, which is 63% of the total traffic in this case.

My previous implementation of the server used the standard Jackson JSON converter and it caused a different behavior. The response builder was not able to determine the JSON message's size at the build time. Due to this the response was sent in 3 different TCP packets and assembled on the client side to one HTTP response. The size of the messages was 239, 192 and 61 bytes which is a total of 492 bytes. The reason why the server sends the response in multiple packages is due to the "Transfer-Encoding: chunked" header. It is commonly used in cases when the content's length is unknown or the message's size is large, although, in my case the payload was only 134 bytes long. I modified the server to construct the JSON object beforehand and with this modification the response builder was able to determine the size of the message payload and now it is able to send back the response message in only 1 TCP fragment. This configuration on the server side radically reduced the transferred message's size, in this example to 75%.

Response:

```
HTTP/1.1 200 OK
Connection: keep-alive
X-Powered-By: Undertow 1
Server: Wildfly 8
Transfer-Encoding: chunked
Content-Type: application/json
Date: Tue, 21 Apr 2014 20:28:54 GMT
```

Payload:

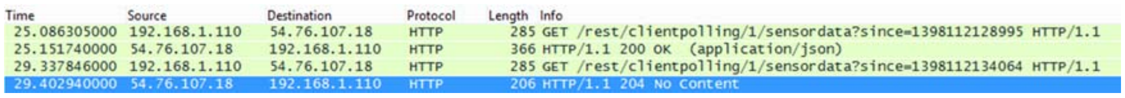
```
{"carId":1,"timestamp":1398112134064,"gpsLatitude":47.53398,
"gpsLongitude":19.079723,"speed":1.25,"rpm":1250,
"coolantTemperature":101}
```

Figure 22 – Client polling HTTP response example with payload

One commonly used way in HTTP polling implementations is to send a timestamp parameter and the server sends back only newer data and does not send back the latest one all cases. This simple modification reduces the response size in the case of no data is available; thereby it does not use as much network resources. The size of the message is 206 bytes which is only 56% of the response shown in Figure 22, although, the client does not receive any meaningful information. The NO RESPONSE message is visible in Figure 23.

```
HTTP/1.1 204 No Content
Connection: keep-alive
X-Powered-By: Undertow 1
Server: Wildfly 8
Content-Length: 0
Date: Tue, 20 May 2014 20:28:46 GMT
```

Figure 23 – Client polling HTTP response with no content



```
Time          Source          Destination     Protocol  Length Info
 25.086305000 192.168.1.110   54.76.107.18    HTTP         285 GET /rest/clientpolling/1/sensordata?since=1398112128995 HTTP/1.1
 25.151740000 54.76.107.18    192.168.1.110   HTTP         366 HTTP/1.1 200 OK  (application/json)
 29.337846000 192.168.1.110   54.76.107.18    HTTP         285 GET /rest/clientpolling/1/sensordata?since=1398112134064 HTTP/1.1
 29.402940000 54.76.107.18    192.168.1.110   HTTP         206 HTTP/1.1 204 No Content
```

Figure 24 – Request and response messages captured by WireShark

4.1.3. Long Polling

Similarly to HTTP polling, the long polling technology is also based on AJAX messages. The client initiates the connection with an HTTP request and the server replies with a response. The main difference is due to the asynchronous call used in long polling, so that the server only sends back the response in case new data arrives and until that the connection is blocked.

As a result of this the size and content of the messages are the same, in my example only the requested URL is different (*/rest/clientpolling/1/sensordata?since=1398112128995* and */rest/longpolling/1/sensordata*). The size of the GET request in Figure 25 is only 263 bytes.

```
GET /rest/longpolling/1/sensordata HTTP/1.1

Accept: application/json

Host: petersomogyi.com

Connection: Keep-Alive

User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.2;
Android SDK built for x86 Build/KK)
```

Figure 25 – Long polling HTTP request example

The main difference is in low message rate situations, because the server blocks the response and does not send back HTTP 204 NO CONTENT message immediately, only after the timeout expires. In the CarCare service's example that I used, the messages arrive to the client in every 5 second in case the car is moving; otherwise no sensor data is transmitted. Considering this message rate I configured the regular polling to initiate a new request in every 4 seconds, and the long polling timeout was set to 30 second.

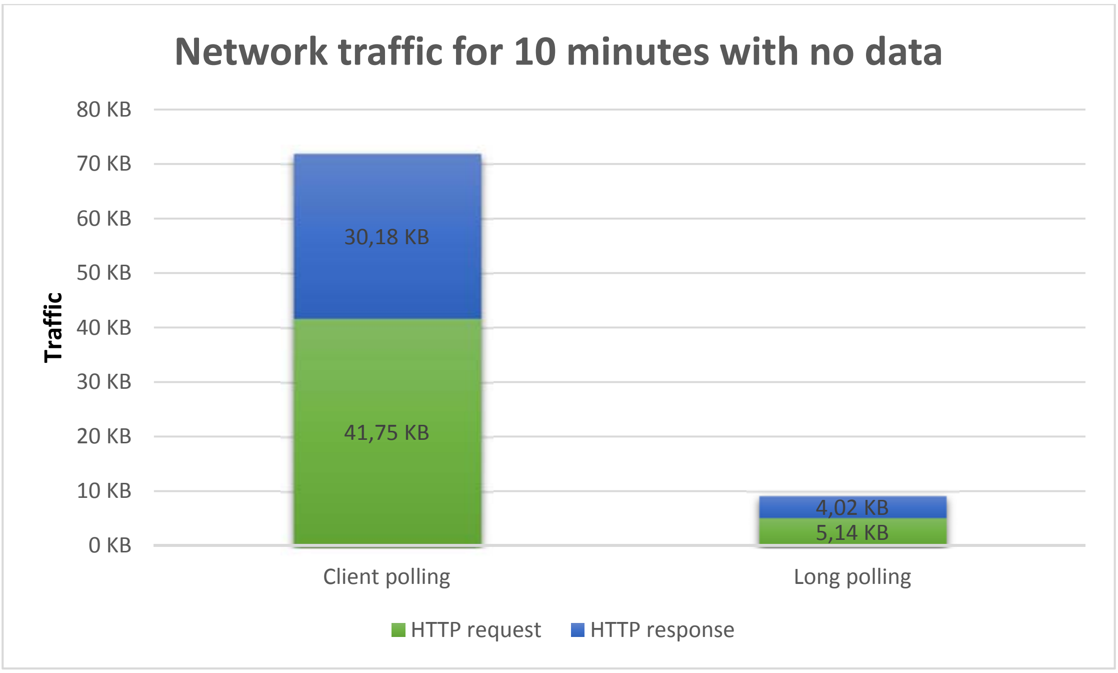


Figure 26 – Client polling and long polling for 10 minutes with no sensor data

Assuming that no sensor data arrive to the server for 10 minutes then the long polling implementation will send 20 different requests. However, this number is 150 in the client polling example due to the synchronous HTTP requests. The diagram in Figure 26 shows the total traffic using the 2 technologies for 10 minutes with no sensor data.

	Request	Response
Client polling	285B * 150 = 41.75KiB	206B * 150 = 30.18KiB
Long polling	263B * 20 = 5.14KiB	206B * 20 = 4.02KiB

Table 5 – Client polling a long polling size comparison

4.1.4. WebSocket

WebSocket was designed in a way to minimalize message framing and create a bidirectional communication socket connection between the host and client. This communication is separated to three main parts, the opening handshake, messages and closing handshake. In the next subchapters I will analyze these focusing on the transferred packages.

WebSocket opening handshake

In order to establish a WebSocket connection, the client has to send an HTTP request to the server which contains a special header, called Upgrade. This indicates that the client wants to upgrade the HTTP connection to WebSocket protocol. The message example in Figure 27 shows WebSocket handshake between the Android app and the application server running under *petersomogyi.com* domain.

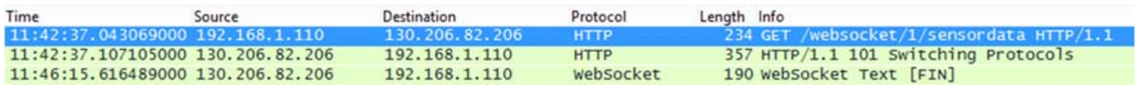
```
GET /websocket/1/sensordata HTTP/1.1
Host: petersomogyi.com:80
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Key: Zd+Y3WBHzV7u3WcoFJFnYg==
Sec-WebSocket-Version: 13
```

Figure 27 – HTTP request from the client

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
X-Powered-By: Undertow 1
Sec-WebSocket-Location:
      ws://petersomogyi.com:80/websocket/1/sensordata
Server: Wildfly 8
Upgrade: WebSocket
Content-Length: 0
Sec-WebSocket-Accept: wDFeipVsF+GStDUG/QttRozS8PM=
Date: Fri, 02 May 2014 09:42:42 GMT
```

Figure 28 – HTTP response from the server

The size of the client request in Figure 27 is 234 byte and the HTTP upgrade response's (Figure 28) size is 357 byte. These messages between the client and server contain the required headers, although there are optional ones that are not present. Due to this, the size of the opening handshake can increase on both sides, but since it has to be transmitted only once in each WebSocket session the size of these messages are not significant.



```
Time               Source          Destination     Protocol    Length Info
11:42:37.043069000 192.168.1.110   130.206.82.206  HTTP           234 GET /websocket/1/sensordata HTTP/1.1
11:42:37.107105000 130.206.82.206  192.168.1.110   HTTP           357 HTTP/1.1 101 Switching Protocols
11:46:15.616489000 130.206.82.206  192.168.1.110   WebSocket      190 WebSocket Text [FIN]
```

Figure 29 – WebSocket opening handshake captured by WireShark

Table 6 explains the WebSocket specific headers for the opening handshake [39].

Header	Description
Sec-WebSocket-Key required	Sent from the client to the server in the opening handshake to prevent cross-protocol attacks.
Sec-WebSocket-Accept required	Sent from the server to the client in the opening handshake, to confirm that the server understands the WebSocket protocol. The value is calculated from the Sec-WebSocket-Key value sent by the client.
Sec-WebSocket-Version required	Sent from the client to the server in the opening handshake to indicate version compatibility.
Sec-WebSocket-Extensions optional	Sent from the client to the server, and then from the server to the client. This header helps the client and server to agree on a set of protocol-level extensions to use for the duration of the connection.

Header	Description
Sec-WebSocket-Protocol optional	Sent from the client to the server, then from the server to negotiate the used protocol. This header advertises the protocols that a client-side application can use. The server uses the same header to select at most one of those protocols.

Table 6 – WebSocket specific headers in the opening handshake

After a successful upgrade procedure, the connection transforms to the data-framing format used for WebSocket messages. The handshake process fails if the server does not respond with the 101 response code, Upgrade header, and Sec-WebSocket-Accept header. The value of the Sec-WebSocket-Accept response header is calculated from the Sec-WebSocket-Key request header and that must match exactly what the client expects [23].

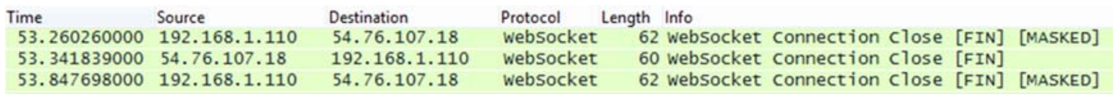
WebSocket closing handshake

The WebSocket connection can be closed from both sides. The party that initiates the action has to specify the purpose using a code and a reason message. Table 7 shows the standard RFC-6455 closing codes. It is possible to use other close codes as well, 3000-3999 has to be registered with Internet Assigned Numbers Authority and 4000-4999 can be used for application specific reasons [23].

Code	Description
1000	Normal close
1001	Going away
1002	Protocol error
1003	Unacceptable data type
1004 - 1006	Reserved, unused
1007	Invalid data
1008	Message violates policy
1009	Message too large
1010	Extension required
1011	Unexpected condition
1015	TLS failure

Table 7 – Defined WebSocket Close Codes

The opcode of the closing message is 8; the closing code and reason are encoded in the message payload. The size of the closing message is 60 bytes but can be larger due to the reason message's size in the payload. The other side has to send a closing message too and in case both parties received this message, the connection closes. Since the connection can close at any time due to any application or network problems, it might happen that there is no closing handshake at the end of the connection. The following screenshot shows the transferred TCP packets for the WebSocket closing handshake initiated by the client.



```
Time            Source          Destination     Protocol   Length Info
 53.260260000   192.168.1.110   54.76.107.18    WebSocket     62 WebSocket Connection Close [FIN] [MASKED]
 53.341839000   54.76.107.18    192.168.1.110   WebSocket     60 WebSocket Connection Close [FIN]
 53.847698000   192.168.1.110   54.76.107.18    WebSocket     62 WebSocket Connection Close [FIN] [MASKED]
```

Figure 30 – WebSocket closing handshake captured by WireShark

Message format

In case the connection is open, both the client and server are able to send messages to each other at any time in full-duplex mode. The messages are represented on the network with a binary syntax that marks the boundaries between frames and includes type information. Frames can be combined to form messages. Typically there is one frame for each message, but a message can be composed of many frames. Figure 31 shows the structure of the WebSocket messages.

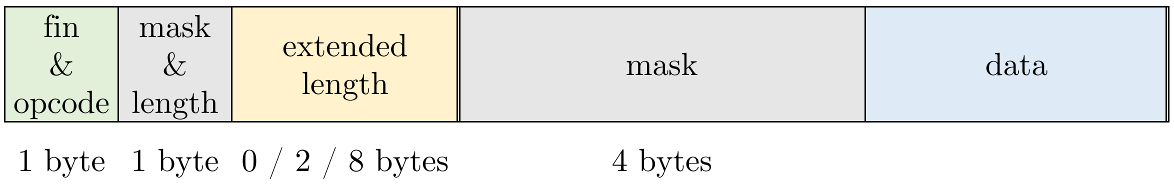


Figure 31 – WebSocket frame structure

One of the main goals of WebSocket is the reduction of unnecessary network traffic. For this reason the frames contains limited information, such as the type of the message, length and masking. The first bit of a frame indicated that this is the final fragment in the message.

The opcode in the first byte of the frame indicates the message type. The WebSocket protocol enables to send text-based messages in the payload using UTF-8 format which is indicated with opcode 0x01. It is also supported to transfer binary messages which have opcode 0x02. Other types of messages that use the same format are ping and pong messages, as well as the closing message.

WebSocket encodes fragment length using variable number of bits to support small messages to use a compact encoding at the same time enable to transfer medium and big messages. In case the message's size is under 126 byte, the length is sent in the second byte of the message. If the message size is bigger, then the length is encoded using 2 extra bytes in the extended length part of the message. For even larger messages, WebSocket supports to use 8 extra bytes to indicate the message size [39].

Masking has to be used for messages sent from client to server to hide their content. The aim of the masking is to prevent eavesdropping. The masking key has to be 4 bytes long and selected by the client for each message and has to be unpredictable.

```
1... .... = Fin: True
.000 .... = Reserved: 0x00
.... 0001 = Opcode: Text (1)
0... .... = Mask: False
.111 1110 = Payload length: 126 Extended payload length (16
              bits)
Extended payload length (16 bits): 134

Payload:
   Text: {"carId":1,"timestamp":1398087275548,"gpsLatitude":
   45.505245,"gpsLongitude":0.063358,"speed":43.25,
   "rpm":2487,"coolantTemperature":78}
```

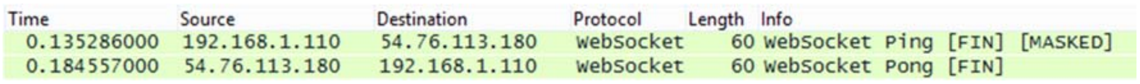
Figure 32 – Example WebSocket message

Figure 32 shows a text message sent from the server to the Android application captured by WireShark. The size of the JSON payload in the message is 134 bytes and all the data sent through the network interface is 192 bytes which means the extra WebSocket framing is 58 bytes.

WebSocket ping

In order to have a reliable WebSocket communication I had to send ping messages from the clients to the server. More information about the need of ping messages is discussed in 4.2.1. The Ping and Pong messages are like regular WebSocket messages; only the opcode is different, 9 for the ping and 10 for the pong. Due to the similarity these messages can also contain payload but in my implementation I sent these messages without it to minimize the network traffic. The size of these messages are 60 bytes and during my experiments I found out that sending ping messages in every 5 minutes is sufficient. According to this the accumulated size of the ping and pong messages for one client in one day is 33.75 KiB.

$$(24 * 12\, message) * (60B + 60B) = 33.75\, KiB$$



```
Time          Source          Destination     Protocol   Length Info
  0.135286000 192.168.1.110   54.76.113.180   WebSocket      60 WebSocket Ping [FIN] [MASKED]
  0.184557000 54.76.113.180   192.168.1.110   WebSocket      60 WebSocket Pong [FIN]
```

Figure 33 – WebSocket Ping and Pong messages

4.1.5. Estimations

Focusing on the CarCare service example, the estimated network traffic might be calculated based on the travelling time of a typical citizen. According to the NewGeography's and FIA Foundation's research, the average commute time per capita in the European Union's metropolitan areas is 50 to 60 minutes per day depending on the size of the city [40] [41]. Considering that the OBD-II device is configured to transmit the sensor and telemetric data to the server in every 5 seconds when the engine is running, than the smartphone with an open connection would receive 600 – 700 relevant messages from the server per day.

Based on the aforementioned message sizes of the 3 communication protocols and the configuration of the device I estimated the following network traffic for client polling, long polling and WebSocket for 24 hours. The estimations can be found in Appendix A

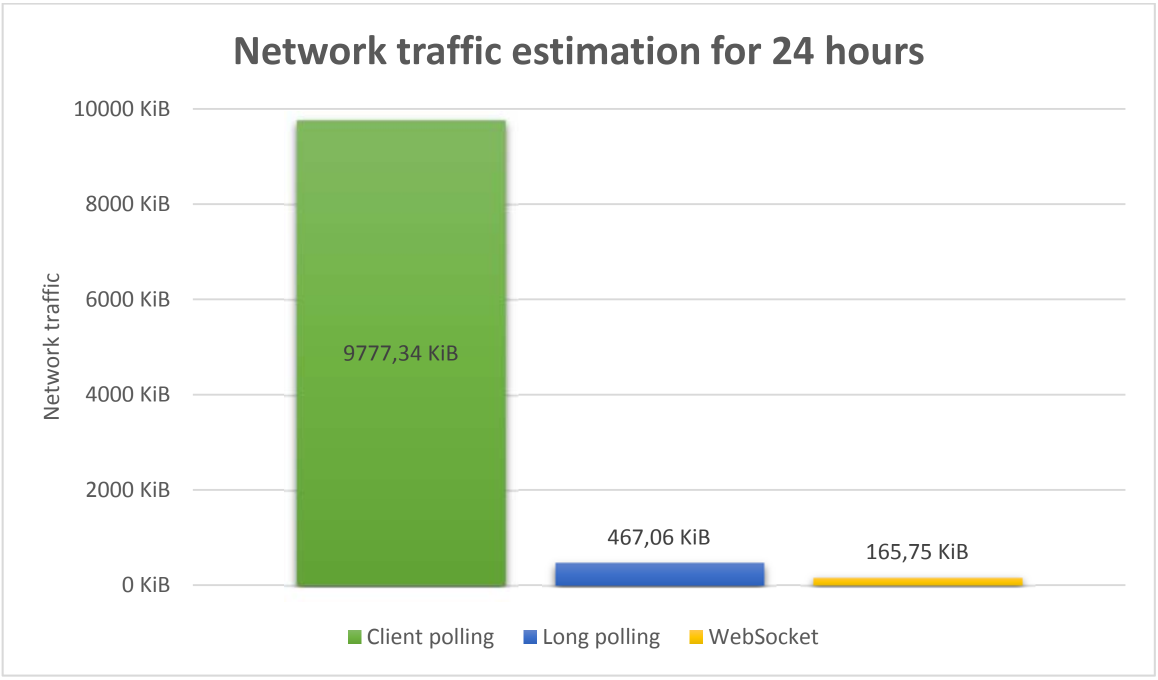


Figure 34 – Traffic estimation for 24 hours timeframe

According to these calculations the client polling implementation is not recommended at all due to the lot of unnecessary requests, the long polling and WebSocket communication forms are more efficient in network bandwidth. Even so there is almost 3 times more traffic generated for the long polling implementation compared to WebSocket protocol for the aforementioned scenario. According to this, in case the network traffic is crucial, I recommend WebSocket communication based on this analyzed criterion, but in most cases this 300 KiB additional traffic does not make significant difference.

4.2. Reliability

Reliability is one of the most significant factors in recent network oriented applications. It is important to send and receive the same data, namely it does not change during the transmission or recognize any network failures such as lost connection. Transmission Control Protocol (TCP) provides a reliable protocol with ordered data transfer, retransmission of lost packages and error-free data transfer. TCP is one of the core protocols which is on top of the Internet Protocol (IP) [42].

The 3 technologies I analyze use TCP protocol and because of this HTTP and WebSocket are generally considered as reliable technologies, due to the acknowledgement messages, retransmission, ordered delivery as well as the error detection and correction which are provided by the TCP protocol.

4.2.1. WebSocket Connection Drop

During the analysis I noticed that neither the library I use for the Android application does not send ping messages to the server and nor the server does not send the ping messages automatically. After further investigation I found out that the WebSocket standard specifies the ping and pong messages' structure, however, it is not specified how regularly the client or the server should send this messages to the other party. Due to this, many of the WebSocket implementations, including the official WebSocket site's echo test [43] does not send ping messages.

Due to this deficiency I detected an important problem with WebSocket. After relatively long period without data transfer the connection is interrupted between the server and the client. It happens because the network devices closed the idle TCP connection and after that it is not possible to send messages. To identify the critical period I conducted a test where the client opened a WebSocket connection and after that the server sent short messages with different interval. I preformed this test with different settings, using LTE connection, home Wi-Fi access as well as enterprise Wi-Fi using proxy server. I repeated the test 5-10 times using the aforementioned 3 network access and after these measurements I recognized the period until the connection is still alive highly depends on the used network access.

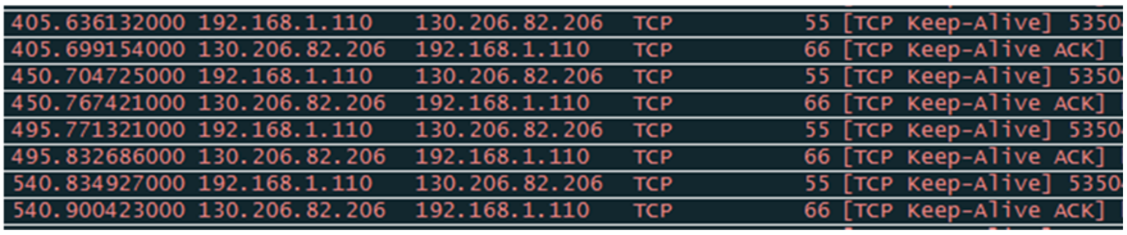
Network type	Amazon EC2 server
LTE connection	8 – 15 minutes
Home Wi-Fi	20 – 40 minutes
Enterprise Wi-Fi	60 – 80 minutes
Local network	3 hours+

Table 8 – WebSocket connection drop times

Table 8 contains the results of this experiment. For these tests I used the Amazon EC2 server instance that I described in Chapter 3.1.4, but apart from that I made another experiment when the mobile device and server were in the same local network connected to the same router. This time the connection was alive even after 3 hours of inactivity, but this test cannot be considered as a real environment. Due to this I focused on the other 3 scenario and according to this suggested to use ping messages with 5 minutes intervals.

Unfortunately, this is a crucial reliability issue, because this connection drop was not recognized by client and the server. Sending data during this period did not cause any network failures, however the messages were not available on the other side of the connection. This problem can be solved by sending ping and pong messages periodically. A background thread should be configured to send this messages, it can be implemented in the server or client side. I implemented it on the client side because in this way it does not require as much resources on the server side and it can serve more clients simultaneously. When I configured the Android application to send these messages in every 5 minutes I haven't experimented similar connection failures. Using higher interval length might cause problem based on my experiments, for example 10 minutes can cause problem with LTE connection using T-Mobile network in Budapest.

Note: Using desktop Chrome browser and the same application server I noticed that the client sends TCP keep alive messages to the server and due to this the connection is not interrupted even after long idle time. These messages were sent in a regular interval, in every 45 seconds which is shown in the WireShark screenshot in Figure 35*. It is important to stress that these TCP packages are not equivalent with the WebSocket ping and pong messages. Using other desktop or mobile browsers these keep-alive messages were not sent.*



```
405.636132000 192.168.1.110   130.206.82.206  TCP        55 [TCP Keep-Alive] 5350
405.699154000 130.206.82.206  192.168.1.110   TCP        66 [TCP Keep-Alive ACK]
450.704725000 192.168.1.110   130.206.82.206  TCP        55 [TCP Keep-Alive] 5350
450.767421000 130.206.82.206  192.168.1.110   TCP        66 [TCP Keep-Alive ACK]
495.771321000 192.168.1.110   130.206.82.206  TCP        55 [TCP Keep-Alive] 5350
495.832686000 130.206.82.206  192.168.1.110   TCP        66 [TCP Keep-Alive ACK]
540.834927000 192.168.1.110   130.206.82.206  TCP        55 [TCP Keep-Alive] 5350
540.900423000 130.206.82.206  192.168.1.110   TCP        66 [TCP Keep-Alive ACK]
```

Figure 35 – TCP keep-alive messages from Chrome browser

4.2.2. Network Type Change

With mobile devices it is common that the used network type changes from Wi-Fi to cellular network, especially if the user is on the way. Of course in this case the network connection breaks and the server is not able to deliver messages to the mobile device only if the application reconnects to the server. This reliability issue can cause outages mostly for the WebSocket protocol, but also long polling and client side polling affected by this.

The network change can be easily detected by the handheld devices, but it requires special process using WebSocket protocol to initiate a new connection every time this scenario happens. With Android system the `ConnectivityManager` broadcasts the `CONNECTIVITY_ACTION` action whenever the connectivity details have changed. The developer can register a broadcast receiver in the manifest to listen for these changes and resume (or suspend) the background updates accordingly [44].

Contrary, using long polling technology without any modification the application can receive data from the server even after the network connection changed. This is possible because the technology is based on multiple, independent connections and only 1 asynchronous call would be affected, the next request would automatically use the new network connection. Although, due to the connection change the client might not receive some data from the server that was sent during this time. Of course, using the same broadcast receiver that I mentioned previously it is possible to detect this outage and initiate a new request from the client immediately.

The client polling solution is not affected as much as the previous ones thanks to the synchronous HTTP calls. The round trip time of the request and response messages was about 0.065 second using the Amazon cloud server located in Ireland and the Android device in Hungary with Wi-Fi connection. Considering the message frequency typically used for client polling purposes – in the CarCare case I set it to 5 seconds – the probability that the connection changes between cellular network and Wi-Fi access during a request-response message is low. Furthermore, the client can easily detect the problem due to a socket error or an unsuccessful HTTP response code.

4.2.3. Proxy Caching

When I performed the tests using Ericsson's network access I observed an interesting problem that affects the 2 polling technologies. That corporate network uses proxy servers to access the internet and this cached the previous response that was actually sent by the external server, therefore the smartphone application received outdated, incorrect data [45]. I noticed this scenario using the long polling implementation. After the device received the first response from the server and sent a new request then it immediately received the response despite the fact that there were no message from the server, furthermore, the server does not even received the sent request.

```
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: 0
```

Figure 36 – HTTP headers to disable caching

One fundamental solution is to set headers in the response message that instructs the client as well as the proxy server not to cache the response message. These headers are visible in Figure 36. In my case setting these headers was not sufficient. I was not able to identify what was the cause of this problem; I assume it is due to a configuration of the used proxy server.

Another way disable caching is to provide an additional query parameter to the requested URL which contains a random number or text. In case the client provides this extra parameter the neither the proxy server not the client will cache the content because the URLs with different query parameters are considered as different resources [46]. The optimal range of the extra parameter depends on the size of the cache, a popular Hungarian news site – www.index.hu – previously used this technique for mobile devices and the query parameter was a random number between 1 and 1000.

This reliability issue does not affect the WebSocket-based implementation, only the two HTTP based polling requests.

4.3. Latency

Latency in web performance means how much time it takes for the client to retrieve data. The latency highly depends on distance between the server and the client, the used network type and speed, the number of network components between them as well as the used implementation. There are specific scenarios when the latency should be as low as possible, for example stock trading and online multiplayer gaming require really low latency. For my case example, latency is not that relevant requirement; nevertheless, I also analyze the 3 communication protocols based on this criterion.

For the test I calculated the average latency of the messages and also the standard deviation. Generally it is easier to predict the behavior of a technology which deviation is lower since bigger differences in the latency are less frequent. During the latency tests the approximate round trip time of ping messages between my computer and the Amazon cloud instance was 48 ms.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}\,, where\ \mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Equation 1 – Calculating standard deviation

4.3.1. Polling

Client polling is much different in terms of latency than the other 2 communication protocols that I analyzed. We can consider the round trip time of the request and response message as the latency which should be relatively low. Based on my experiments, for 150 connections the average time to send a request and receive the response from the Amazon server took approximately 66ms, only the first request took longer time which was 0.26 seconds. The change of the latency can be seen on Figure 37. The vertical axis shows the latency between 0.05 and 0.15 seconds, thus the first message's deviation is not completely visible.

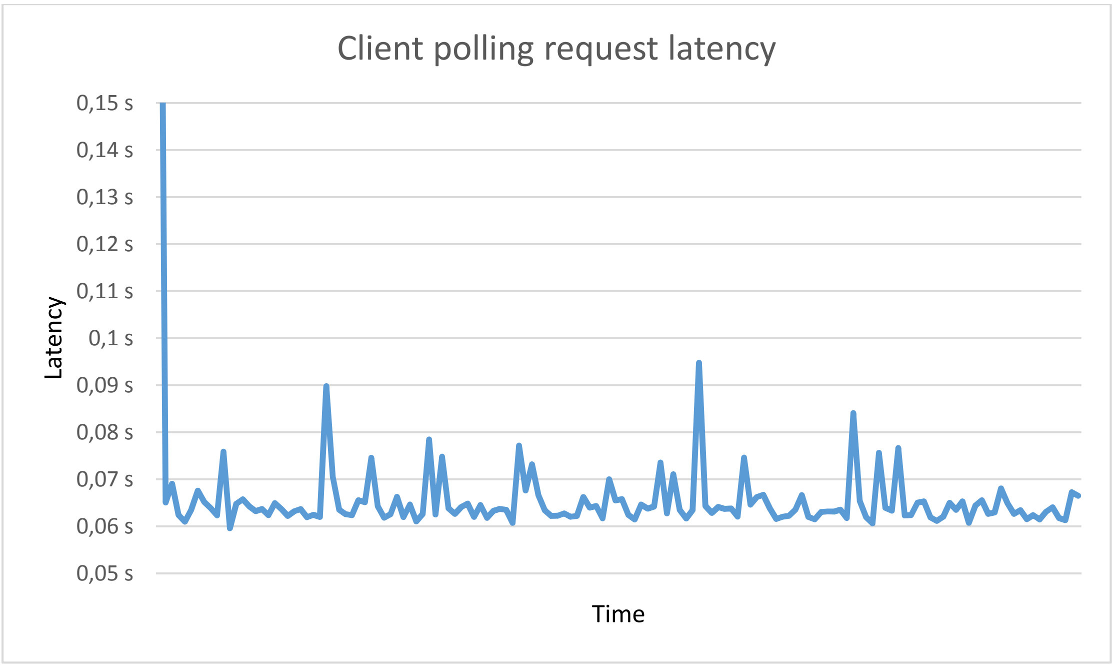


Figure 37 – Client polling request latency

However, this does not guarantee that the smartphone application will receive the sensor data updates from server with this delay due to the client polling's behavior. In the CarCare example I used 4 seconds for the polling interval and it means the new sensor data can arrive to the client even after 4 seconds after it was generated, furthermore, the average latency is half of the interval, 2 seconds. In the best case scenario the client polling implementation is the round trip time latency. For this the new data has to arrive to the server at the same time as the request from the client.

I run a different test where the Android client polled the server in every 4 seconds and the sensors arrived to the server in 5 seconds interval. In this test I calculated the delay between the sent sensor data to the server and the time it arrived to the smartphone application. The result of this experiment can be found in Figure 38. As it shows the time for the data to arrive to the smartphone took quite long time, the average latency for the 286 data was 2.15 seconds which is basically half of the polling interval. It is important to note that the standard deviation is also high, 1.19 seconds. It is possible to reduce the latency by polling more often but of course it greatly increases network traffic.

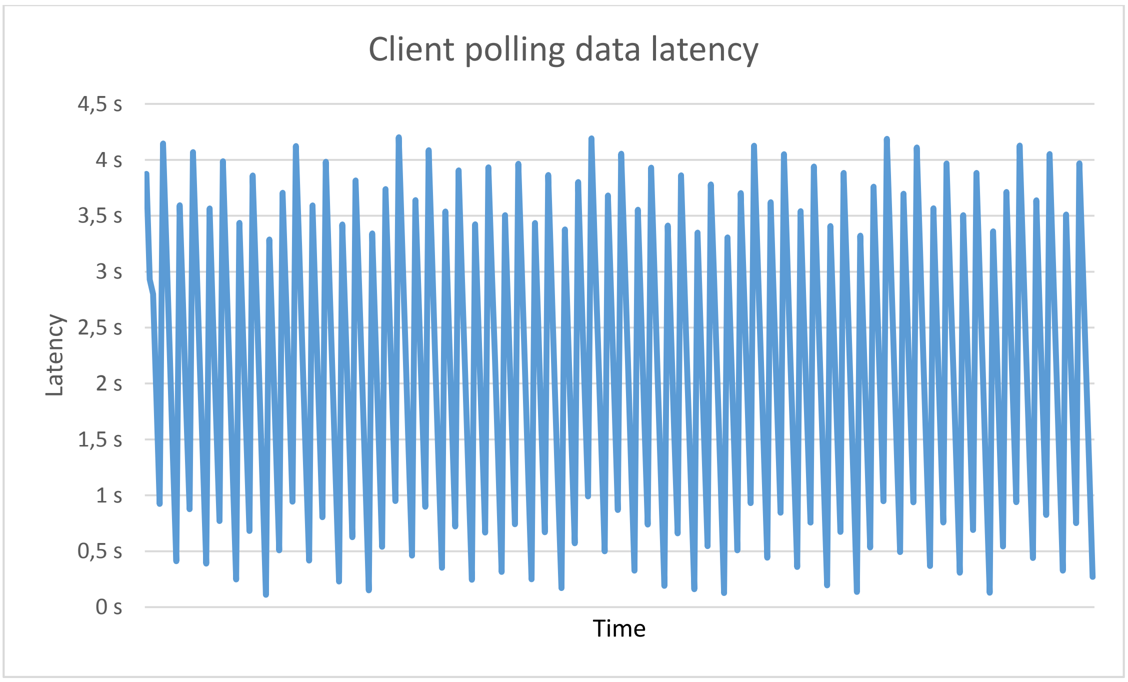


Figure 38 – Client polling data latency

4.3.2. Long Polling

For the long polling test I cannot measure the time between the HTTP request and response on the client side due to the characteristics of the technology. To measure the latency I used the demo application on an Android simulator and WireShark to capture the network packages. I sent the sensor data to the Amazon server from the same machine using the server's rest interface. This way I was able to determine the delay between the 2 different HTTP packets. Of course, this also includes the POST request's latency. Due to this I used similar method to examine the latency for the WebSocket messages as well.

For this test I sent 286 sensor data to the server while the application was connected to the server with long polling technology and the results were similar to what I measured in the client polling test. The first request took 0.48 seconds; 2 times more than in the HTTP polling case. However, the average latency was 68 ms, so basically there is no difference compared to the request-response latency in the client polling implementation.

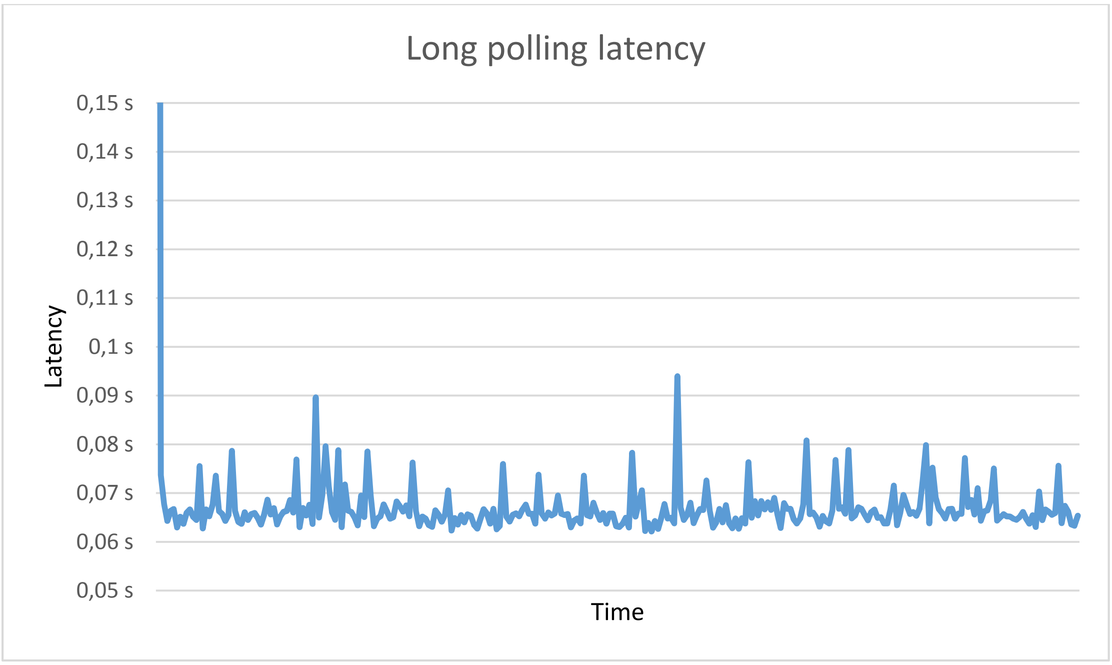


Figure 39 – Long polling latency

4.3.3. WebSocket

WebSocket's latency is generally considered lower than the long polling implementations [47]; however, my experiments show similar latency for these 2 technologies. I run the test the same way as in the long polling case using an Android simulator and I calculated the difference between the POST request that sent the sensor data to the server and the arrived WebSocket message. The first message's latency was 0.58 seconds and the average delay was 70 ms during the experiments.

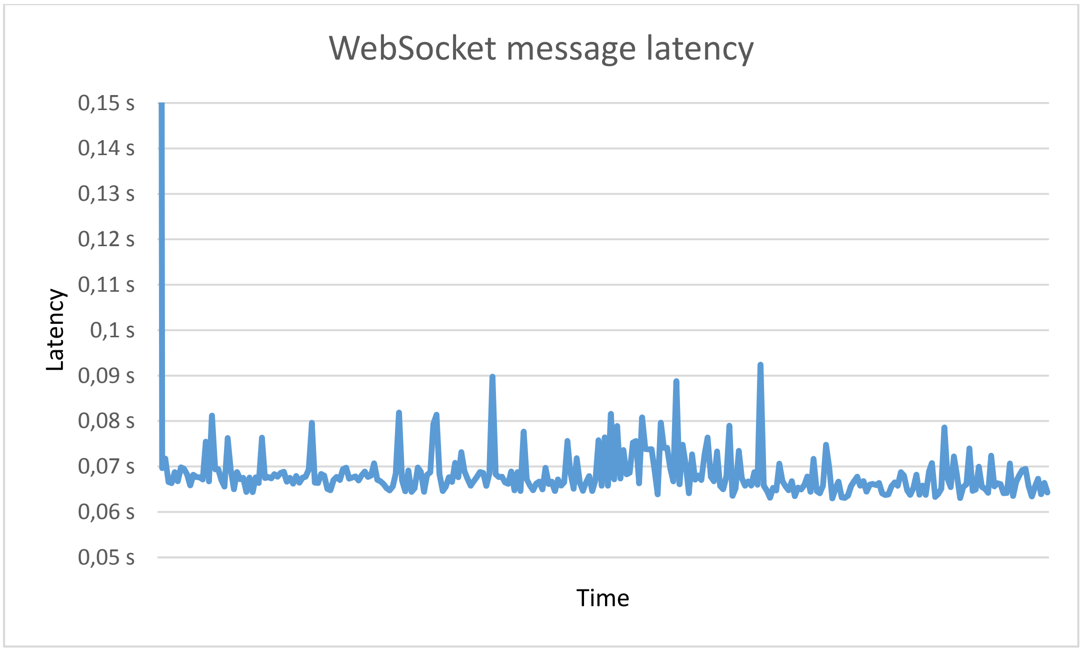


Figure 40 – WebSocket message latency

Of course, this latency does not show purely the WebSocket messages latency, that's why I made another experiment, in which I calculated the time difference between the WebSocket ping and pong messages. I run the application for 50 minutes and sent the ping messages from the client in every 5 seconds. In this case the latency between the Amazon server and smartphone application was only 50ms which is only 2 ms higher than the standard IMPC ping messages [48].

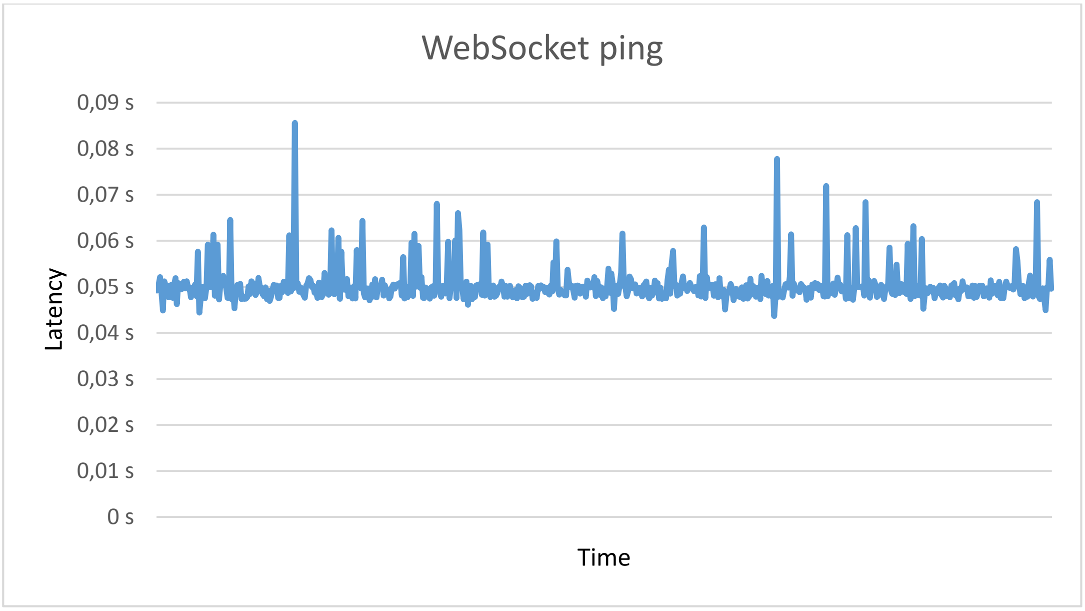


Figure 41 – WebSocket ping latency

In this case the first ping message's latency was not higher like in the previous example, because this experiment did not include the REST calls which generated higher initial latency.

4.3.4. Overview

Because all the three technologies had a much higher initial latency I assume that it is connected with the first POST request and its network routing which I used for sending the JSON messages to the application server. According to this I calculated the average and standard deviation of the latency for the rest of the messages. These values are presented in Table 9.

	Average	Deviation
Client polling request latency	65 ms	5.1 ms
Client polling data latency	2,15 s	1,19 s
Long polling	67 ms	4.1 ms
WebSocket messages	68 ms	4.4 ms
WebSocket ping	50 ms	3.7 ms

Table 9 – Average and standard deviation of the latency

4.4. Load Test

An efficient communication technology needs to serve multiple clients connections at the same time; furthermore it should behave in a predictable way while the server load increases.

For this reason I planned to perform a load test on the server to see how these 3 technologies affect the performance in terms of CPU, memory and network load as well as how many simultaneous clients can the server handle. To do the tests I connected to the server with an Android device in order to measure the latency of the messages and recorded the server load. In order to connect multiple clients I made a simple program which was able to simulate multiple client connections from the same computer. During the test I continuously sent sensor data to the server in every 5 seconds.

For the first time I gradually increased the number of connections. Until approximately 4000 connections I haven't experienced increase in the latency of the messages (Figure 42) and also the server's CPU load was between 10-20%. When the number of connections reached about 5000 simultaneous users the server crashed unexpectedly. When I checked the log files I noticed this was due to the system settings which limited the number of open files and IO connections.

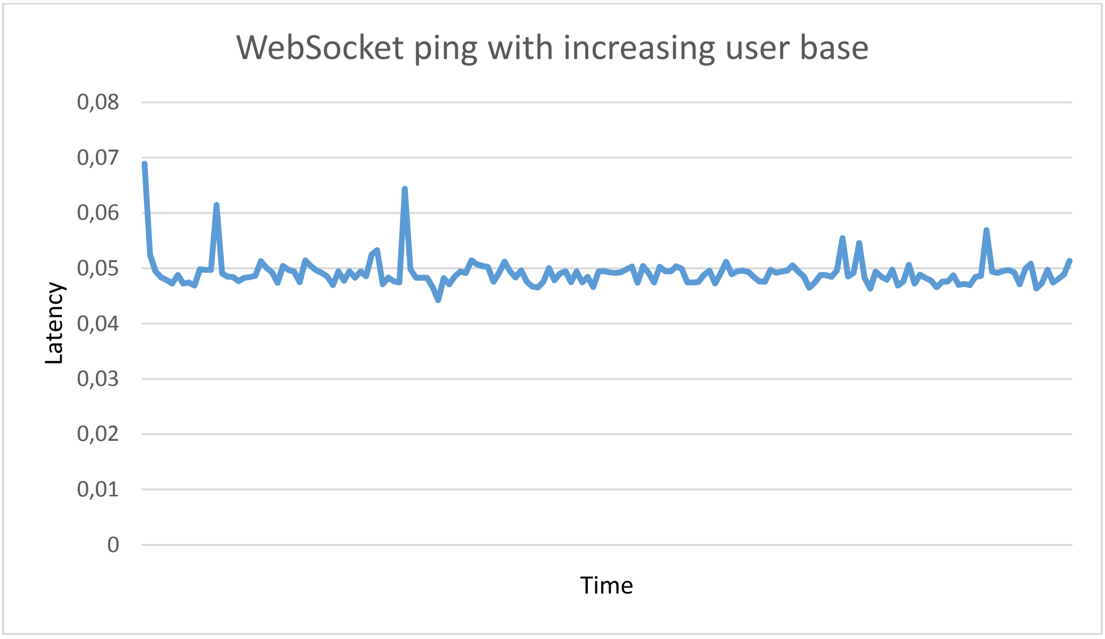


Figure 42 – WebSocket ping with increasing user number up to 5000

After I modified the settings I haven't experienced similar problem. I performed the same experiment; however I was not able to detect increase in the message latency with any of the 3 technologies when I connected to the server with a total of 15000 connections from 2 computers.

Unfortunately, with my available devices I was not able to perform a valuable load test on the server. However, I believe that it would have been really interesting to see how the 3 different implementations react to high user number.

A performance research [30] of WebSocket says the following. "Monitoring the server CPU utilization in real time reveals that it very rarely goes past 25% on a quad core system, effectively using one CPU core 100%. The sequential nature of message processing is a probable reason for this. The system can never be faster than the initial processing of the messages (which is sequential). The sending of the messages to the clients is not very CPU intensive. Only when a client is disconnected and the messages have to be merged in the outgoing client queue it will show in CPU utilization. This is why we can see a clear increase in CPU utilization when clients begin to disconnect in the final sequence of the tests."

A paper written by Engin Bozdag, Ali Mesbah and Arie van Deursen [49] claims that long polling and similar technologies brings some scalability issues, in their experiment the CPU usage was 7 times higher than in the client polling case around 350-500 simultaneous users.

4.5. Security

Using web services has become an important part of our everyday life. Since the spread of smartphone and tablet devices many applications moved to provide web-based services instead of the traditional, offline applications. We use email, online banking, financial applications, social websites, voice-over IP (VoIP) and many other services on our mobile devices that require secured connections. Therefore, it is required form a competitive communication protocol to support secured connections.

There are some security concerns about WebSocket. An article written by Jussi-Pekka Erkkilä [50] discusses possible security threats. For example the Origin Policy from HTTP is not used, instead a mechanism called verified-origin. The main problem could be arises from the lack of official standards.

4.5.1. TLS

The Transport Layer Security (TLS) – previously Secured Socket Layer (SSL) is a cryptographic protocol which is designed to provide privacy and data integrity between two communicating applications typically over the internet using a combination of public-key and symmetric-key encryption. "One advantage of TLS is that it is application protocol independent. Higher-level protocols can layer on top of the TLS protocol transparently. The TLS standard, however, does not specify how protocols add security with TLS; the decisions on how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left to the judgment of the designers and implementers of protocols that run on top of TLS." [51]

SSL or TLS Handshake

In order to have secured communication using SSL or TLS, the client and server has to establish a handshake. The handshake allows the server to authenticate itself to the client by using public-key techniques, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server [52].

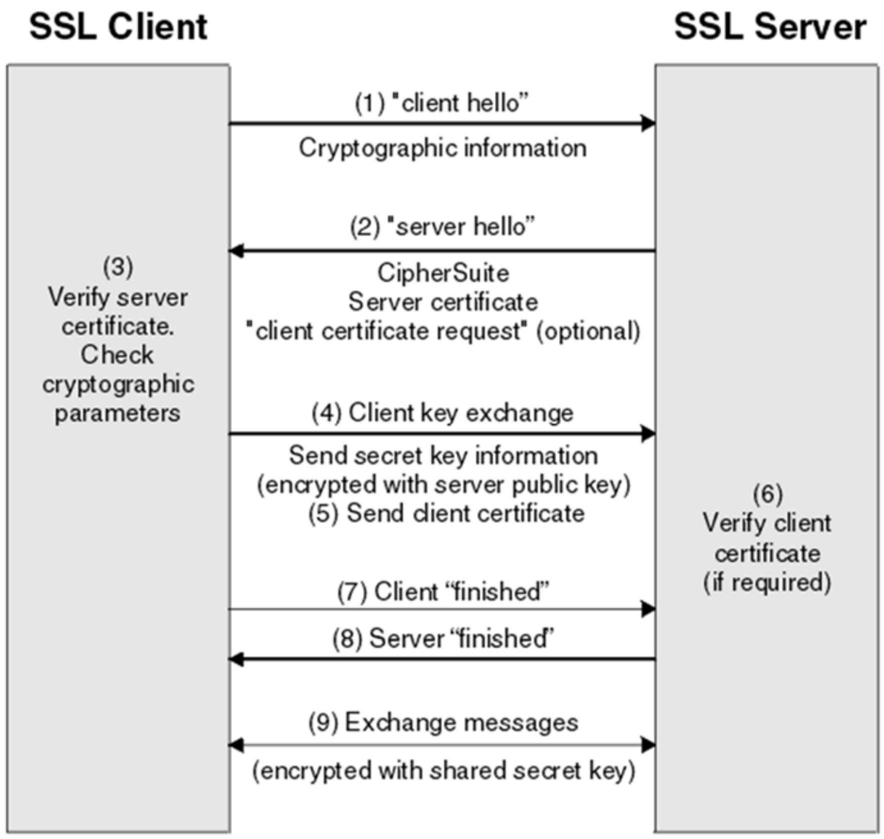

SSL Client
SSL Server
(1) "client hello"
Cryptographic information
(2) "server hello"
CipherSuite
Server certificate
"client certificate request" (optional)
(3) Verify server certificate. Check cryptographic parameters
(4) Client key exchange
Send secret key information (encrypted with server public key)
(5) Send client certificate
(6) Verify client certificate (if required)
(7) Client "finished"
(8) Server "finished"
(9) Exchange messages
(encrypted with shared secret key)

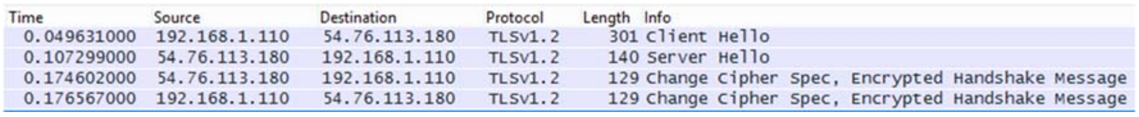
Figure 43 – SSL/TLS handshake workflow

Source:
http://pic.dhe.ibm.com/infocenter/wmqv7/v7r5/index.jsp?topic=%2Fcom.ibm.mq.sec.doc%2Fq009930_.htm

1. The SSL or TLS client sends a client hello message that lists cryptographic information such as the SSL or TLS version and, in the client's order of preference, the CipherSuites supported by the client. The message also contains a random byte string that is used in subsequent computations. The protocol allows for the client hello to include the data compression methods supported by the client.
2. The SSL or TLS server responds with a server hello message that contains the CipherSuite chosen by the server from the list provided by the client, the session ID, and another random byte string. The server also sends its digital certificate. If the server requires a digital certificate for client authentication, the server sends a client certificate request that includes a list of the types of certificates supported and the Distinguished Names of acceptable Certification Authorities (CAs).

3. The SSL or TLS client verifies the server's digital certificate. For more information, see How SSL and TLS provide identification, authentication, confidentiality, and integrity.
4. The SSL or TLS client sends the random byte string that enables both the client and the server to compute the secret key to be used for encrypting subsequent message data. The random byte string itself is encrypted with the server's public key.
5. If the SSL or TLS server sent a client certificate request, the client sends a random byte string encrypted with the client's private key, together with the client's digital certificate, or a no digital certificate alert. This alert is only a warning, but with some implementations the handshake fails if client authentication is mandatory.
6. The SSL or TLS server verifies the client's certificate. For more information, see How SSL and TLS provide identification, authentication, confidentiality, and integrity.
7. The SSL or TLS client sends the server a finished message, which is encrypted with the secret key, indicating that the client part of the handshake is complete.
8. The SSL or TLS server sends the client a finished message, which is encrypted with the secret key, indicating that the server part of the handshake is complete.
9. For the duration of the SSL or TLS session, the server and client can now exchange messages that are symmetrically encrypted with the shared secret key.

Source: IBM [53]



```
Time         Source          Destination     Protocol  Length Info
 0.049631000 192.168.1.110   54.76.113.180   TLSv1.2      301 Client Hello
 0.107299000 54.76.113.180   192.168.1.110   TLSv1.2      140 Server Hello
 0.174602000 54.76.113.180   192.168.1.110   TLSv1.2      129 Change Cipher Spec, Encrypted Handshake Message
 0.176567000 192.168.1.110   54.76.113.180   TLSv1.2      129 Change Cipher Spec, Encrypted Handshake Message
```

Figure 44 – TLS handshake before the communication

The 3 different communication types that I analyze use TLS protocol to encrypt the messages. The TLS handshake packages are the same for all the 3 protocols which is visible in the following WireShark screenshot. The total size of the handshake is 699 bytes, however, due to the secured connection the framing of the messages increase which I will analyze in later.

4.5.2. Generating Self-Signed Certificate

The SSL or TLS Certificates are data files that digitally associate the cryptographic key to an organization's details. When this certificate is installed on a web server, it enables the clients and the server to use HTTPS and other secured connection that is based on SSL or TLS security protocol. Typically, secured connections are used to secure credit card transactions, data transfer and logins, and more recently is becoming the norm when securing browsing of social media sites [54].

In order to install an SSL certificate for the WildFly web-server that I used during the testing, I generated a self-signed certificate using keytool software with the command in Figure 45:

```
> keytool -genkey -alias wildfly -keyalg RSA
```

Figure 45 – Command to generate certificate

After this I had to configure WildFly server to use the generated certificate which can be done in the used configuration file. For the standalone.xml file I added a new security-realm element which is displayed in Figure 46 [55].

```
<security-realms>
  ...
  <security-realm name="UndertowRealm">
    <server-identities>
      <ssl>
        <keystore path="my.keystore" relative-to="jboss.server.config.dir"
            keystore-password="my-pass" />
      </ssl>
    </server-identities>
  </security-realm>
</security-realms>
```

Figure 46 – Configuration for the generated certificate file for WildFly

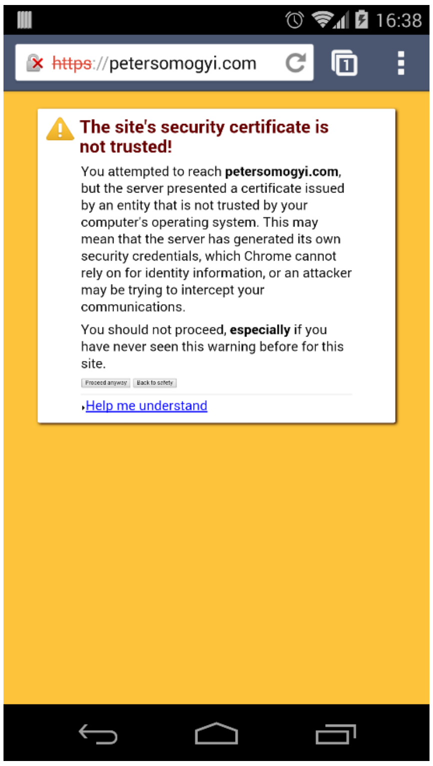
Furthermore, I had to add an HTTPS listener under the *undertow* subsystem. After this, the web-server is able to serve HTTPS requests as well as to establish Secure WebSocket connections with the *wss://* protocol.

```xml
<subsystem xmlns="urn:jboss:domain:undertow:1.0">
  ...
  <server name="default-server">
    <http-listener name="default" socket-binding="http" />
    <https-listener name="https" socket-binding="https"
         security-realm="UndertowRealm" />
    <host name="default-host" alias="localhost">
      <location name="/" handler="welcome-content" />
      <filter-ref name="server-header" />
      <filter-ref name="x-powered-by-header" />
    </host>
  </server>
  ...
</subsystem>
```

Figure 47 – Required HTTPS listener for WildFly configuration file

This self-signed certificate is not equivalent with an official certificate. In order to obtain a valid SSL/TLS certificate, I would need to register it from a trusted authority, so I only used the self-signed version. Unfortunately, I was not able to do complete tests using the 3 selected technologies with TLS protocol due to the limitations of the self-signed certificate.

Opening the website on the mobile device the visitor receives a notification that the server's certificate is not trusted, which is visible on the screenshot in Figure 48. In case the user accepts the certificate he or she is able to use the secured connection, only the address bar will indicate the warning. Using the application I made for the testing I was not able to configure the device to use this secured connection from the native application with untrusted certificates. However, I was able to use the website from the Chrome browser and perform tests to analyze the HTTPS and Secure WebSocket connection. In order to do a trustworthy comparison, I used the same JSON objects in the messages.

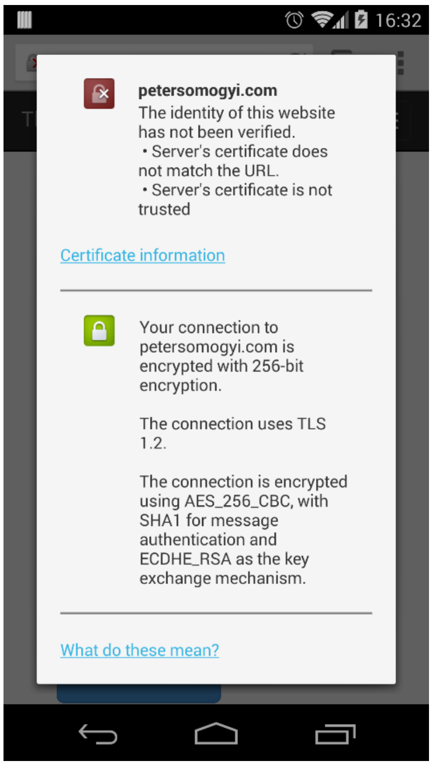


The site's security certificate is not trusted!

You attempted to reach **petersomogyi.com**, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Figure 48 – Android Chrome's warning for untrusted certificate



petersomogyi.com
The identity of this website has not been verified.
• Server's certificate does not match the URL.
• Server's certificate is not trusted

Your connection to petersomogyi.com is encrypted with 256-bit encryption.

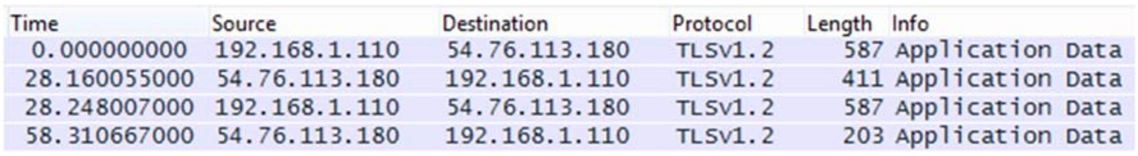
The connection uses TLS 1.2.

The connection is encrypted using AES_256_CBC, with SHA1 for message authentication and ECDHE_RSA as the key exchange mechanism.

Figure 49 – Android Chrome's certificate information

4.5.3. Message Size

Due to the encryption, the size of the messages increased regardless the used communication technology. As I explained in Chapter 4.1, the structure and size of the client polling and long polling messages are relatively the same. Without using any encryption, the request messages were 285 and 263 bytes sent from the Android application. When I used the same web-service with HTTPS over TLS protocol, the size of the requests were 587 bytes in the client polling and long polling case as well. At the same time, the server sent responses with the following JSON content were 411 bytes and HTTP 204 NO CONTENT responses were 203 bytes. Previously, these messages were 366 and 206 bytes which contained 134 byes of JSON data in the payload. It means to send the same messages over a secured connection the TLS protocol will add an extra 300 bytes overhead just to the requests, however, the response only increased with 45 bytes; moreover the HTTP 204 NO CONTENT answer was even smaller using the TLS protocol.



```
Time          Source          Destination     Protocol  Length Info
  0.000000000 192.168.1.110   54.76.113.180   TLSv1.2      587 Application Data
 28.160055000 54.76.113.180   192.168.1.110   TLSv1.2      411 Application Data
 28.248007000 192.168.1.110   54.76.113.180   TLSv1.2      587 Application Data
 58.310667000 54.76.113.180   192.168.1.110   TLSv1.2      203 Application Data
```

Figure 50 – Long polling TLS messages

The WireShark capture on Figure 50 shows the transferred messages for long polling. The first and third lines are HTTP requests, the second is a reply from the server with the JSON payload and the final one is a response with no content which was sent after the 30 second timeout.

```
{"carId":1,"timestamp":1398112134064,"gpsLatitude":
 47.53398,"gpsLongitude":19.079723,"speed":1.25,
 "rpm":1250,"coolantTemperature":101}
```

Figure 51 – Used JSON payload

Using WebSocket the size of the messages increased from 192 bytes to 235 bytes which is only 22% increase. This is the lowest increase comparing these 3 technologies; the increase caused by the secured connections can be found in Table 10.

Technology	Unsecure	Secure	Increase
Client polling with JSON (request + response)	651 B	998 B	53%
Client polling without JSON (request + response)	491 B	790 B	61%

Technology	Unsecure	Secure	Increase
Long polling with JSON (request + response)	629 B	998 B	59%
Long polling without JSON (request + response)	469 B	790 B	68%
WebSocket	192 B	235 B	22%

Table 10 – Comparison of message size increase with security

4.5.4. Latency

Before I made tests for latency using secured connections I expected that the increase will be similar since all the technologies I analyze use TLS protocol. However, the increase for WebSocket was much higher compared to the long polling example.

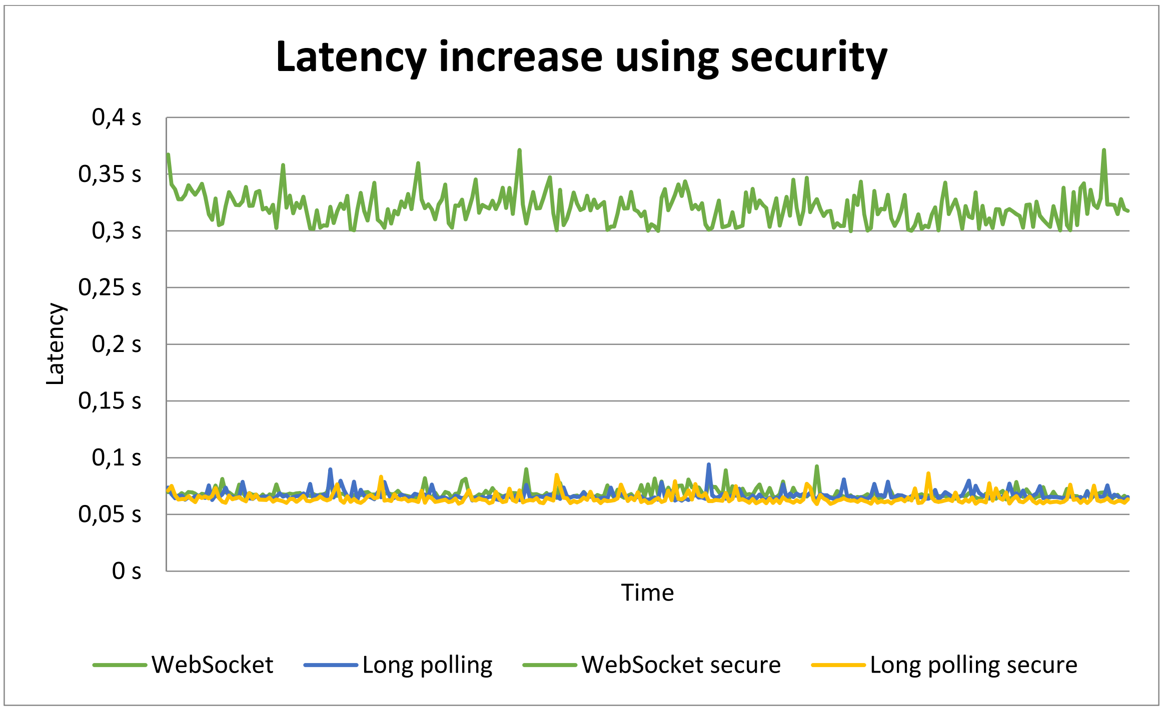


Figure 52 – Latency comparison of unsecured and secured connections

As Figure 52 shows long polling with and without using security have the same latency as pure WebSocket connection. However, using Secure WebSocket connection radically increased it which was unexpected. The exact values can be found in Table 11. This radical increase was unexpected, and I assume it is arising from the used WebSocket library.

Technology	Average	Standard deviation
Long polling	66 ms	4.1 ms
Secured long polling	64 ms	4.3 ms
WebSocket	68 ms	4.4 ms
Secured WebSocket	320 ms	13.1 ms

Table 11 – Latency comparison of unsecured and secured connections

Note: The unsecured long polling request's latency was 2 ms higher than the secured one which is unlikely. Since the 2 experiment was done on different days I assume it is due to the change in the network's latency.

5. Conclusion

This chapter will summarize my results of the client polling, long polling and WebSocket technologies. The main goal of this thesis was to analyze these three communication technologies focusing on smartphone and server communications with specific requirements. I will also discuss some possibilities for future work in this area.

5.1. My Results

My objective was to find the most suitable communication form to deliver real-time data from a server to smartphone devices. For this I selected 3 communication technologies, namely client polling, long polling and HTML5 WebSocket. I selected 5 specific requirements to compare these technologies.

The first I examined was message size. Regarding this I compared JSON and XML representation of data and stated that JSON is much more efficient in terms of size and parsing speed [34]. Therefore I selected the JSON representation and worked with this during the assessments. As it was expected, the most efficient technology in terms of message size among this three is WebSocket, which is not surprising, because one of the main principles of the technology was to reduce message framing [39]. The HTTP based technologies require bigger message framing to deliver the same message. Although long polling technology aims to reduce the unnecessary request that client polling has, but my estimated network traffic for 24 hours with the used CarCare example was still 2.5 times bigger than in the WebSocket case.

The second requirement I had was reliability. Generally all the technologies I analyzed can be considered reliable since all of them use TCP connection to transfer messages. Due to this the network problems are handled in the Transport Layer of the OSI model [56], such as retransmission in case of lost packets. Apart from this, I noticed reliability problems which concern all of these technologies; however, I was able to solve these problems. With long polling and client polling I found that using proxy servers can cause caching problems. Generally, this can be easily solved with using specific headers in the HTTP messages; however, in my case it does not solve the problem possible due to the used proxy server's configurations. A workaround I finally used was to pass a randomly generated query parameter which solved this problem.

A reliability issue I found affects WebSocket. After long inactivity on the network neither the server nor the client was able to send messages to the other party and the lost connection was not recognized. This can be solved by sending ping messages. My experiment shows that sending ping in every 5 minutes solved this problem. However, it is still a major reliability issue, because some libraries, such as AutoBahn or JavaScript from browser, do not enable to send ping messages and in this case the connection can be lost. Since WebSocket API is still in a Candidate Recommendation stage the issue with lost connection could be fixed later on, but until now this problem can case big reliability issue. A paper called A Real-Time Group Communication Architecture Based on WebSocket [57] claims that WebSocket can be used for real-time communication, however, currently it is not fully supported and different browsers support different portions of the specification.

Another requirement was low latency. As I discussed previously, client polling is significantly worse in delivering real-time data to the client compared to the other technologies due to its implementation. Of course it can be reduced by using more frequent polling interval; however it would radically increase the network traffic. As I reviewed related studies I found out that WebSocket's latency is generally lower than long polling implementation, however, my experiments show that this 2 technology have similar latency and also the standard deviation was almost the same. [49]

Load testing would have been a really important and interesting experiment, but unfortunately I was not able to perform this test due to limitations that I explained in Chapter 4.4.

Security is a major requirement for a competitive communication technology. Therefore, all the three analyzed technology supports secured connection over TLS protocol. The only required configuration was to install security certificate for the application server and changing to secured protocols in the URIs was the only modification needed for the clients. When I analyzed the secured connections I noticed that the client polling and long polling messages' size increased by 60%, however, for the secured WebSocket connection this growth was only 20-30%. On the other hand, latency radically increased in the WebSocket case. As my experiments show, it increased from 70 ms to 300 ms while the long polling latency did not change using encrypted connection. This huge increase might be caused by the used library and not the Secured WebSocket connection's characteristic.

As it was expected both long polling and WebSocket technologies outperformed client polling in terms of message size and latency. The main benefit of WebSocket is the reduced message size and the bi-directional communication. However, currently WebSocket is not necessarily provides a reliable connection and the users have to face higher latency when encrypted connection is used.

5.2. FUTURE WORKS

While this thesis has covered the major requirements for an efficient communicating technology for real-time server and smartphone communication, many opportunities for extending the scope of this thesis remain. This part covers some of the possible directions for future works.

5.2.1. PROOF THE RESULTS

All of my experiments were conducted on the same server and Android library. Redoing the same tests on other platform might end up with different results or it can strengthen my findings. Since the aim of the thesis is to cover server and smartphone communication, the most common smartphone operating systems needs to be tested which are currently Android, iOS and Windows Phone

8. Furthermore, alternative servers should be tested, because latency and scalability can highly depends on the server-side implementation.

5.2.2. Scalability and Load Test

As I discussed previously I was not able to accomplish the load testing of the server. Therefore, it is definitely an area where this research can go forward. My initial plan was to monitor the server's CPU, memory and network load while the user number progressively increases. Another modification could be that the clients are not connected from multiple geographical locations to simulate a real production environment.

Generally, a big system uses multiple servers which are located behind a load balancer for two main purposes. The first reason is to provide a secondary server which can take over the clients in case the primary server breaks down. The other purpose is to use multiple servers simultaneously. In this way the server load can be reduced by redirecting the incoming connections to different back ends. HTTP and HTTPS traffic are generally supported by load balancers, however, it might not be the case with WebSocket connection.

5.2.3. Other Technologies

This thesis focused on client polling, long polling and WebSocket technologies. Of course there are other possibilities that can be suitable for server and smartphone communication that I did not cover but could show interesting results. One of these is also an HTML5 technology, called Server-Sent Events [58] which is still in Candidate Recommendation state but shows great opportunities for HTML-based applications. Another possibility is to use simple socket connection; however it would require implementation of different interfaces for smartphone and web applications. Cloud messaging [59] [60] is another interesting field which is mainly used for delivering notifications to the mobile devices. A drawback of this technology is the underlying UDP protocol and due to this it is not guaranteed that the messages arrive to the clients.

5.2.4. Other Message Types

The main concept of my thesis was about transferring real-time data from the server to the mobile devices. This is an interesting field due to the increasing popularity of device-to-device communication and Internet of Things. However, there is a need for different kinds of communications as well.

The traditional request and response based messages are still needed and widely used communication pattern. It can be used to retrieve information that does not need to be sent continuously, for example to fuel level or static information about the car. Sending large data files can be considered as request-response communication; however, other kind of communication technologies might be more efficient. Notification is another kind of message type which is basically a message that needs to be sent to the client based on an event or time. The trigger can be an accident when the device can notify the authorities or another notification could be sent in case the check engine lamp lights up. As these examples show there are many different communication types for which the analyzed three technologies can be also suitable but the requirements does not necessarily meets the ones I analyzed.

6. Appendices

6.1. Appendix A

This appendix covers the message size estimations for 24 hours interval with 50-60 minutes travelling time. The messages are transferred in every 5 minutes when the car is moving. Table 12 covers the client polling estimation, Table 13 has the results for long polling and Table 14 shows the message size calculations for WebSocket.

Client polling	One message	24 hours
Request	263B	21 600 * 263B = 5547.65KiB
Response with JSON	366B	700 * 366B = 250.19KiB
Response with no content	206B	20 900 * 206B = 4204.5KiB

Table 12 – Client polling 24h estimation

Long polling	One message	24 hours
Request	285B	746 * 285B = 207.62KiB
Response with JSON	366B	700 * 366B = 250.19KiB
Response with no content	206B	46 * 206B = 9.25KiB

Table 13 – Long polling 24h estimation

WebSocket	One message	24 hours
Opening handshake	234B + 357B	591B

Closing handshake	62B + 60B + 62B	184B
Messages	192B	700 * 192B = 131.25KiB
Ping / Pong	60B + 60B	24 * 12 * 120B = 33,75KiB
Sum	1.06KiB	165.75KiB

Table 14 – WebSocket 24h estimation

7. Bibliography

[1] Ericsson, "10 Hot Consumer Trends 2014," 2014.

[2] M. Rouse and I. Wigmore, "Internet of Things (IoT)," 2013. [Online]. Available: http://whatis.techtarget.com/definition/Internet-of-Things. [Accessed: 12-May-2014].

[3] Business Insider, "The Internet of Everything: 2014," 2014. [Online]. Available: http://www.businessinsider.com/the-internet-of-everything-2014-slide-deck-sai-2014-2. [Accessed: 12-May-2014].

[4] Waze, "Free Community-based Mapping, Traffic & Navigation App." [Online]. Available: https://www.waze.com/. [Accessed: 08-Jun-2014].

[5] Ford.com, "SYNC and SYNC with MyFord Touch." [Online]. Available: http://www.ford.com/technology/sync/. [Accessed: 10-Jun-2014].

[6] P. Sawers, "Bill Ford Outlines His 'Blueprint for Mobility' in Cars," 2012. [Online]. Available: http://thenextweb.com/mwc/2012/02/28/bill-ford-the-automotive-and-telecom-industries-are-at-a-historic-crossroads/. [Accessed: 12-May-2014].

[7] G. Templeton, "Google's chance to dominate the robot car market is quickly slipping away | ExtremeTech," 2014. [Online]. Available: http://www.extremetech.com/extreme/181982-googles-chance-to-dominate-the-robo-car-market-is-quickly-slipping-away. [Accessed: 12-May-2014].

[8] Apple, "CarPlay." [Online]. Available: http://www.apple.com/ios/carplay/. [Accessed: 10-Jun-2014].

[9] B&B Electronics, "OBD-II Background Information," 2011. [Online]. Available: http://www.obdii.com/background.html. [Accessed: 16-May-2014].

[10] Wikipedia, "On-board diagnostics," 2014. [Online]. Available: https://en.wikipedia.org/wiki/On-board_diagnostics. [Accessed: 11-Jun-2014].

[11] "The HTTP Protocol As Implemented In W3," 1991. [Online]. Available: http://www.w3.org/Protocols/HTTP/AsImplemented.html. [Accessed: 14-May-2014].

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and P. Leach, "Hypertext Transfer Protocol -- HTTP/1.1," 1999. [Online]. Available:

http://www.w3.org/Protocols/rfc2616/rfc2616.html. [Accessed: 16-May-2014].

[13] J. J. Garrett, “Ajax: A New Approach to Web Applications | Adaptive Path,” 2005. [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/. [Accessed: 22-Apr-2014].

[14] Jotorres, “What is AJAX? - CodeProject,” 2013. [Online]. Available: http://www.codeproject.com/Articles/534632/WhatplusisplusAJAX-f. [Accessed: 14-May-2014].

[15] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. [Accessed: 14-May-2014].

[16] G. Mulligan and D. Gracanin, “A Comparison of Soap and Rest Implementations of a Service Based Interaction Independence Middleware Framework,” pp. 1423–1432, 2009.

[17] S. Tilkov, “A Brief Introduction to REST,” 2007. [Online]. Available: http://www.infoq.com/articles/rest-introduction. [Accessed: 14-May-2014].

[18] Wikipedia, “Create, read, update and delete,” 2014. [Online]. Available: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. [Accessed: 08-Jun-2014].

[19] P. Lubbers and F. Greco, “HTML5 Web Sockets: A Quantum Leap in Scalability for the Web | SOA World Magazine,” 2010. [Online]. Available: http://soa.sys-con.com/node/1315473. [Accessed: 22-Apr-2014].

[20] R. Gravelle, “Comet Programming: Using Ajax to Simulate Server Push,” 2009. [Online]. Available: http://www.webreference.com/programming/javascript/rg28/index.html. [Accessed: 15-May-2014].

[21] A. Alinone, “Comet and Push Technology,” 2007. [Online]. Available: http://cometdaily.com/2007/10/19/comet-and-push-technology/. [Accessed: 15-May-2014].

[22] G. Wilkins, S. Salsano, S. Loreto, and P. Saint-Andre, “Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP,” 2011.

[23] V. Wang, F. Salim, and P. Moskovits, *The Definitive Guide to HTML5 WebSocket*, 1st ed. Apress, 2013.

[24] WebSocket.org, “About WebSocket.” [Online]. Available: http://www.websocket.org/aboutwebsocket.html. [Accessed: 29-Apr-2014].

[25] H. Hämäläinen, “HTML5 : WebSockets,” pp. 1–9.

[26] XMPP.org, “The XMPP Standards Foundation.” [Online]. Available: http://xmpp.org/. [Accessed: 16-May-2014].

[27] RedHat, “WildFly,” 2014. [Online]. Available: http://www.wildfly.org/. [Accessed: 12-Jun-2014].

[28] Oracle Corporation, “Jersey,” 2014. [Online]. Available: https://jersey.java.net/. [Accessed: 10-Jun-2014].

[29] Oracle Corporation, “WebSocket API Endpoints, Sessions and MessageHandlers.” [Online]. Available: https://tyrus.java.net/documentation/1.4/index/websocket-api.html. [Accessed: 10-Jun-2014].

[30] N. Qveflander, “Pushing real time data using HTML5 Web Sockets,” 2010.

[31] Amazon, “AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting.” [Online]. Available: http://aws.amazon.com/ec2/. [Accessed: 28-May-2014].

[32] Davidiusdadi, “Java-WebSocket.” [Online]. Available: http://java-websocket.org/. [Accessed: 05-May-2014].

[33] Autobahn, “Autobahn|Android Documentation — AutobahnAndroid 0.5.2 documentation.” [Online]. Available: http://autobahn.ws/android/. [Accessed: 28-May-2014].

[34] G. Wang, “Improving data transmission in web applications via the translation between XML and JSON,” in *Proceedings - 2011 3rd International Conference on Communications and Mobile Computing, CMC 2011*, 2011, pp. 182–185.

[35] W3C, “Extensible Markup Language (XML) 1.0 (Fourth Edition),” 2006. [Online]. Available: http://www.w3.org/TR/2006/REC-xml-20060816/. [Accessed: 24-May-2014].

[36] E. B. Sept, “The myth of self-describing XML,” 2003.

[37] E. International, “ECMA-262 ECMAScript Language Specification,” no. June, 2011.

[38] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, “Comparison of JSON and XML Data Interchange Formats : A Case Study,” p. 6, 2009.

[39] I. Fette and A. Melnikov, “The WebSocket Protocol,” 2011. [Online]. Available: http://tools.ietf.org/html/rfc6455. [Accessed: 22-Apr-2014].

[40] W. Cox, “The Transportation Politics of Envy: The United States & Europe,” 2011. [Online]. Available: http://www.newgeography.com/content/002217-the-transportation-politics-envy-the-united-states-europe. [Accessed: 19-May-2014].

[41] FIA Foundation, “The Automobile and Society.”

[42] Information Sciences Institute, “Transmission Control Protocol,” 1981. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt.

[43] WebSocket.org, “Echo Test.” [Online]. Available: http://www.websocket.org/echo.html. [Accessed: 18-May-2014].

[44] Android Developers, “Determining and Monitoring the Connectivity Status | Android Developers.” [Online]. Available: http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html. [Accessed: 28-May-2014].

[45] A. Luotonen, “Proxy server caching mechanism that provides a file directory structure and a mapping mechanism within the file directory structure,” 1999.

[46] D. Parzych, “Caching Behavior of Web Browsers,” pp. 1–10, 2007.

[47] V. Pimentel and B. G. Nickerson, “Communicating and Displaying Real-Time Data with WebSocket,” pp. 45–53, 2012.

[48] Wikipedia, “Ping (networking utility),” 2014. [Online]. Available: http://en.wikipedia.org/wiki/Ping_(networking_utility)#ICMP_packet. [Accessed: 10-Jun-2014].

[49] E. Bozdag, A. Mesbah, and A. van Deursen, “A Comparison of Push and Pull Techniques for AJAX,” *2007 9th IEEE Int. Work. Web Site Evol.*, pp. 15–22, Oct. 2007.

[50] J. Erkkilä, “WebSocket Security Analysis,” 2012.

[51] T. Dierks, “The Transport Layer Security (TLS) Protocol Version 1.2,” 2008.

[52] Microsoft, “Description of the Secure Sockets Layer (SSL) Handshake.” [Online]. Available: http://support.microsoft.com/kb/257591. [Accessed: 20-May-2014].

[53] IBM, "An overview of the SSL or TLS handshake," 2014. [Online]. Available: http://pic.dhe.ibm.com/infocenter/wmqv7/v7r5/index.jsp?topic=/com.ib m.mq.sec.doc/q009930_.htm. [Accessed: 20-May-2014].

[54] GlobalSign, "What is an SSL Certificate?" [Online]. Available: https://www.globalsign.com/ssl-information-center/what-is-an-ssl-certificate.html. [Accessed: 31-May-2014].

[55] JBoss, "SSL Configuration HOW-TO," 2011. [Online]. Available: http://docs.jboss.org/jbossweb/7.0.x/ssl-howto.html. [Accessed: 18-May-2014].

[56] Wikipedia, "OSI model," 2014. [Online]. Available: http://en.wikipedia.org/wiki/OSI_model. [Accessed: 12-Jun-2014].

[57] Y. Zhangling and D. Mao, "A Real-Time Group Communication Architecture Based on WebSocket," vol. 1, no. 4, 2012.

[58] I. Hickson, "Server-Sent Events," 2012. [Online]. Available: http://www.w3.org/TR/eventsource/. [Accessed: 29-May-2014].

[59] Google, "Google Cloud Messaging for Android." [Online]. Available: http://developer.android.com/google/gcm/index.html. [Accessed: 01-Jun-2014].

[60] S. Zhao, P. P. C. Lee, J. C. S. Lui, X. Guan, X. Ma, and J. Tao, "Cloud-based Push-Styled Mobile Botnets : A Case Study of Exploiting the Cloud to Device Messaging Service," *ACSAC '12 Proc. 28th Annu. Comput. Secur. Appl. Conf.*, pp. 119–128, 2012.