Evaluating WebSocket and WebRTC in the Context of a Mobile Internet of Things Gateway

GUNAY MERT KARADOGAN

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Evaluating WebSocket and WebRTC in the Context of a Mobile Internet of Things Gateway

Gunay Mert Karadogan

Master of Science Thesis

Embedded Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

12 January 2013

Examiner: Professor Gerald Q. Maguire Jr.

Abstract

This thesis project explores two well-known real-time web technologies: WebSocket and WebRTC. It explores the use of a mobile phone as a gateway to connect wireless devices with short range of radio links to the Internet in order to foster an Internet of Things (IoT).

This thesis project aims to solve the problem of how to collect real-time data from an IoT device, using the Earl toolkit. With this thesis project an Earl device is able to send real-time data to Internet connected devices and to other Earl devices via a mobile phone acting as a gateway. This thesis project facilitates the use of Earl in design projects for IoT devices.

IoT enables communication with many different kinds of "things" such as cars, fridges, refrigerators, light bulbs, etc. The benefits of IoT range from financial savings due to saving energy to monitoring the heart activity of a patient with heart problems. There are many approaches to connect devices in order to create an IoT. One of these approaches is to use a mobile phone as a gateway, i.e., to act as a router, between IoT and the Internet.

The WebSocket protocol provides efficient communication sessions between web servers and clients by reducing communication overhead. The WebRTC project aims to provide standards for real-time communications technology. WebRTC is important because it is the first real-time communications standard which is being built into browsers.

This thesis evaluates the benefits which these two protocols offer when using a mobile phone as a gateway between an IoT and Internet. This thesis project implemented several test beds, collected data concerning the scalability of the protocols and the latency of traffic passing through the gateway, and presents a numerical analysis of the measurement results. Moreover, an LED module was built as a peripheral for an Earl device. The conclusion of the thesis is that WebSocket and WebRTC can be utilized to connect IoT devices to Internet.

Sammanfattning

I detta examensarbete utforskas två välkända realtidsteknologier på internet: WebSocket och WebRTC. Det utforskar användandet av en mobiltelefon som gateway för att ansluta trådlösa enheter - med kort räckvidd - till Internet för att skapa ett Internet of Things (IoT).

Det här examensarbetet försöker med hjälp av verktyget Earl lösa problemet med hur insamlandet av realtidsdata från en IoT-enhet skall genomföras. I det här examensprojektet kan en Earl-enhet skicka data i realtid till enheter med Internetanslutning, samt till andra Earl-enheter, med hjälp av en mobiltelefon som gateway. Detta projektarbete förenklar användandet av Earl i design-projekt ör IoT-enheter.

IoT tillåter kommunikation mellan olika sorters enheter, så som bilar, kyl- och frysskåp, glödlampor etc. Fördelarna med IoT kan vara allt från ekonomiska - tack vare minskad energiförbrukning - till medicinska i form av övervakning av puls hos patienter med hjärtproblem. Det finns många olika tillvägagångssätt för att sammankoppla enheter till ett IoT. Ett av dessa är att använda en mobiltelefon som en gateway, dvs en router mellan IoT och internet.

WebSocket-protokollet erbjuder effektiv kommunikation mellan web-servrar och klienter tack vare minskad överflödig dataöverföring. WebRTC-projektet vill erbjuda standarder för realtidskommunikation. WebRTC är viktigt då det är den första sådana standarden som inkluderas i webläsare.

Det här examensarbetet utvärderar fördelarna dessa två protokoll erbjuder i det fallet då en mobiltelefon används som gateway mellan ett IoT och Internet. I det här examensprojektet implementerades ett flertal testmiljöer, protokollens skalbarhet och fördröjningen av trafiken genom mobiltelefonen (gateway) undersöktes. Detta presenteras i en numerisk analys av mätresultaten. Dessutom byggdes en LED-modul som tillbehör till en Earl-enhet. Slutsatsen av examensarbetet är att WebSocket och WebRTC kan användas till att ansluta IoT-enheter till Internet.

Acknowledgements

I would like to acknowledge my gratitude to my examiner Professor Gerald Q. Maguire Jr. for his helpful comments and continuous suggestions during the development of this thesis.

I thank my classmate Deniz Akkaya for letting me extend his thesis project by taking a different approach.

Last but not the least, I would like to thank my family for supporting me throughout my life.

Contents

List of Figures

List of Tables

List of listings

List of Acronyms and Abbreviations

6LoWPAN	IPv6 over Low Power Wireless Area Network
AMI	Amazon Machine Images
AWS	Amazon Web Services
API	Application Programming Interface
AJAX	Asynchronous JavaScript and XML
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
DTLS	Datagram Transport Layer Protocol
DNS	Domain Name System
EC2	Elastic Compute Cloud
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I^2C	Inter-Integrated Circuit
IAAS	Infrastructure as a Service
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation

LED	Light-emitting Diode
MCU	Multipoint Control Unit
NAT	Network Address Translation
PCB	Printed Circuit Board
P2P	Peer to Peer
RTC	Real-Time Communication
RTP	Real-time Transport Protocol
SIP	Session Initiation Protocol
SMD	Surface-Mount Device
SSH	Secure Shell
SSL	Secure Sockets Layer
STUN	Session Traversal Utilities for NAT
SCTP	Stream Control Transport Protocol
TCP	Transmission Control Protocol
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
WebRTC	Web Real-Time Communication
WSS	WebSocket Secure
W3C	World Wide Web Consortium
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

The Internet of Things (IoT) is an evolving concept that enables interaction with objects around us through sensors, actuators, mobile devices, and so forth. It is a broad vision that assumes that there will be sensors on many different kinds of things, and that these sensors are connected to the Internet or to other computer systems. This potentially large amount of sensor data may help us to understand the world around us. Additionally, these things may have actuators - so that we can act on the physical world. This vision has been known by different names over the past decades: pervasive computing, ubiquitous computing, smart objects, and now IoT.

According to Cisco's Internet Business Solutions Group, in 2010 the number of devices connected to the Internet was 12.5 billion, while the world's human population was 6.8 billion, thus the average number of connected devices per person was 1.84 [1]. Looking to the future, it is predicted that by 2015 there will be 25 billion devices connected to the Internet and by 2020 this number will be 50 billion [1]. This expected growth is due to both smartphones and IoT.

IoT can be seen as a set of systems that collect data, process this data, and then allow us to react to this data (in many cases potentially by acting on something in the real-world). IoT utilizes many computer and communication technologies. Having such large numbers of devices introduces a number of new requirements, such as analyzing big data, new server architectures, larger networking address spaces (such as provided by IPv6), and new mobile experiences. These technologies frequently utilize real-time web protocols, data management, in-memory computing, and hardware toolkits. This thesis project was driven by the introduction in another thesis project (described in the next section) of a new hardware toolkit. This thesis project will focus on real-time web protocols.

1.1 Problem Description

With the evolving technology, many toolkits have been introduced into the market, such as Arduino [2], mBed [3], and littleBits [4]. Interaction designers at the ICT Sweden Mobile Life research institute [5] wanted to have their own toolkit. Mobile Life is a research institute with a focus on mobile services involving researchers from computer science and interaction design. In this institute a master's student, Deniz Akkaya, implemented a toolkit called Earl [6]. Earl is a library of sensors and actuators that can be connected to the Internet through a mobile phone or a tablet. An Earl instance is implemented with a low power wireless network interface, such as ANT+ [7] or Bluetooth low energy (BLE) [8], and a low power microcontroller such as Texas Instruments' MSP430 [9]. Such a device sends sensor data to a mobile phone via a wireless communication link - typically BLE. This data is forwarded to one or more web services by a mobile phone. Earl aims to provide the user with a means to connect different types of sensors or actuators and to see how these different sensors impact users' interactions. Earl enables designers to create proof of concept systems faster than implementing a system from scratch.

The Mobile Life research institute was asked to conduct a project for ABB Sweden[10] in order to prevent boredom of workers working with ABB's control systems in factories. Ethnographic studies were carried out to understand the reason for this boredom in control rooms. However, these studies did not help to decide on a feasible product to help the control room workers avoid boredom. Moreover, a step-by-step design process would not work because there was not enough time or a predefined problem where a product was specified. Therefore, it was decided to use a co-designing method using technology probes [11]. Technology probes are simple and flexible devices used in a design process aiming to understand the needs and desires of users. Technology probes involve the users actively in the design process and inspire users and designers to formulate new product ideas. In the case of this project for ABB Sweden, the main aim was to let the workers find out what would entertain them as they worked.

Some of the important features of technology probes are that:

- Technology probes usually have only one function and they are easy to interact with.
- Technology probes should collect data about these users in order to help designers generate new product ideas.
- Technology probes should be open-ended in order to enable the data that is collected to be reinterpreted and they should be open for various combinations of usage.

A variety of technology probes were discussed before development began, and Earl (with a mobile phone) was selected to implement the technology probes. Since data collection is important when using technology probes, Earl needed a connection oriented web service that it could push its collected data to in real-time. This web service was needed to extend the usage of Earl for further design projects which were thought to require real-time data collection. The requirements of this web service were:

- For the technology probes of ABB project, the delay in collecting data from Earl needed to be under a second on average. However, for future probes the web service should be analyzed in terms of minimizing any additional delay.
- The web service should support hundreds of Earl connections simultaneously and in the future the service should be able to scale up in order to support thousands of connections.

Moreover, one of the technology probes was an arm band built using Earl with a motor to cause vibrations. This probe (called the arm-probe) enables workers to communicate using arm movement and vibration. The arm-probe senses movement of the arm and sends information about this movement to other Earl devices as a message which triggers vibration. The requirements for this probe are listed below:

- Data should be transferred quickly and the user should not perceive the delay. One way delay should be less than half second.
- Since there are 10 workers in a shift, each mobile phone acting as a gateway should be able to communicate with 9 other mobile phones simultaneously (if a full mesh architecture is used).

In addition to the requirements above, another technology probe was a small ball with LEDs that could be controlled by an Earl device. This probe (called the ball-probe) was designed to change color and shine based on the interactions of the user. It should be able to adjust each LED's brightness and color based upon Earl commands. Thus, Earl needed an LED module which could provide different colors and different outputs to cause the LEDs to have different visual effects, such as blinking and fading.

This thesis project investigates the usage of real-time web technologies to provide solutions to support web services for IoT toolkits, such as Earl. A number of test beds have been implemented, data has been collected about WebSocket (for the case of a web service collecting data) and WebRTC (for the case of the arm-probe). The analysis of this data showed how these real-time web protocols

can provide real-time access to data emitted by IoT devices. Moreover, this thesis project developed an LED module for Earl. A software library has been written and a printed circuit board (PCB) was made for this LED module.

1.2 Problem Context

A family of standards, architectures, and wireless technology, called IP version 6 over Low Power Wireless Area Network (6LoWPAN) [12], enables the latest Internet protocols to be used in low power embedded devices by adapting IP version 6 (IPv6) [13] to suit these constrained devices. It is expected that most future embedded devices will be seamlessly integrated into the Internet. However, the past decade shows that there may not be a single technical standard for IoT due to the fact that IoT spans a broad technological domain.

Two basic approaches have been used to connect things to the Internet: embedding web servers in smart things and using smart gateways [14]. The first approach embeds a lightweight web server into each device and enables access to the embedded device through a web application [15, 16]. In this approach, embedded devices implement TCP/IP and HTTP stacks and an embedded web server. Typically such devices are equipped with low-power Wi-Fi modules. For instance, the OpenPicus project produced a low-cost system on a module, called FlyPort, with embedded Internet connectivity [17]. FlyPort comes with full TCP/IP support and a web server making it easy to seamlessly integrate such a device into the Internet.

While devices with embedded web servers are likely to continue to be popular, the second approach uses intermediate gateways rather than an embedded web server [16, 18]. These gateways receive requests from the Internet and forward the requests to embedded devices via low-power link layer communication protocols, such as ANT+ or BLE. These gateways abstract the details of the underlying wireless link layer communication protocols. Earl utilizes this gateway approach and the gateway runs on a mobile phone. Web requests and responses sent via the gateway are used to control wirelessly connected sensors and actuators. The main advantage of this approach is that it connects low power constrained devices to the Internet without requiring these devices to implement web protocols nor even implement TCP. An Earl mobile application contains an embedded web page within it to be used by the browser. Earl provides a JavaScript API. This thesis project builds upon Earl's gateway approach, but focuses on those web technologies which connect the mobile phone to the Internet rather than focusing on the wireless link technologies connecting an Earl device to the mobile phone. Moreover, Earl was designed to work with peripheral components such as LED modules, vibration motors, etc. This thesis also builds an LED module for Earl

which uses Inter-Integrated Circuit (I^2C). General overview of this architecture is depicted in Figure 1.1 below.

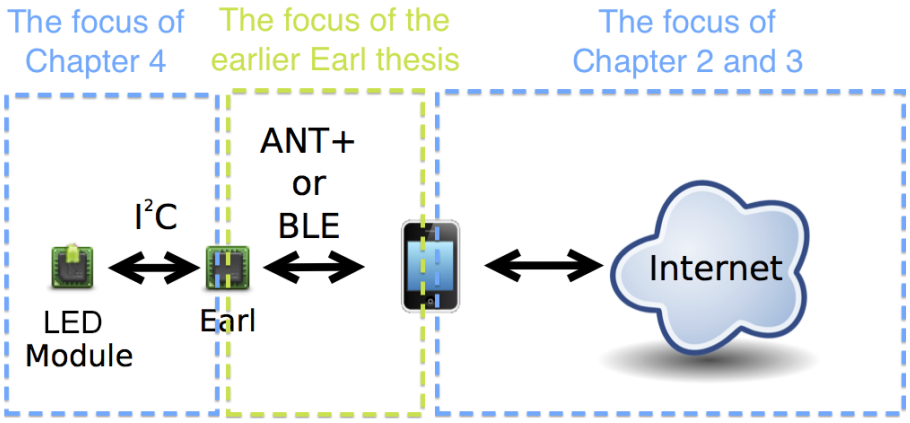


Figure 1.1: The relationships between LED module, Earl, mobile phone, and Internet for this thesis project. This figure also illustrates the relationship of this project to the earlier Earl thesis project [6]

1.3 Goal

The main two goals of this master's thesis project are to provide a server architecture that will transfer real-time data collected by Earl and to provide a P2P WebRTC architecture to realize the arm-probe technology probe. Another goal of this thesis project is to evaluate today's real-time web protocols, specifically WebSocket and WebRTC, in terms of scalability and their compliance with IoT toolkits using the gateway approach. Tests of these protocols should provide numerical data to enable an analysis of the latency and scalability of these technologies. These tests will assume that an Earl device is already communicating with a mobile phone, hence this thesis project will examine the communication from when a mobile phone initiates communication with a web server (in the case of WebSocket) or a peer (in the case of WebRTC). In addition to these goals, an LED module was implemented for Earl. The following activities were defined as the project's deliverables (and they can be used as indicators of the success of the project):

- Performance measurements of the implemented WebSocket architecture with various number of connected clients. Analysis of the results to assess the limits of this architecture.

- Performance measurements of the WebRTC architecture to enable the arm-probe. Analysis of the results to assess this architecture.
- Software implementation and PCB design of an LED module which is compatible with Earl. This effort should reveal the ease of implementing modules for Earl.

1.4 Thesis Structure

Chapter 1 gave a brief introduction to the problem. Chapter 2 describes the tests that will be conducted to evaluate WebSocket, while Chapter 3 will focus on those tests that will be conducted to evaluate WebRTC. In these two chapters, there will be detailed information about the background of the protocols, the tools, data collection process, analysis of data, and discussion of the results. Chapter 4 will describe the work to develop the LED module for Earl. Finally, Chapter 5 will present conclusions and suggest possible future work. Chapter 5 will also discuss the economic, social, and ethical issues associated with this thesis project.

1.5 Methodology

This thesis project utilizes quantitative research methods. It uses an empirical approach to measure the performance of the protocols because quantitative data must be collected to achieve the goals of the project. This thesis project defines performance metrics and conducts experiments to collect data about these metrics. Then, the collected data is analyzed with respect to the project's requirements. In this thesis project, qualitative research methods are not used since the experiments focus on the results of experiments yielding numeric data rather than qualitative data.

Chapter 2

WebSocket Experiments

This chapter describes the design, implementation, and evaluation of a web service that an Earl device can push its collected data to in real-time. We have used the WebSocket protocol due to its reduction in HTTP overhead and low network latency, while building upon existing web protocols. The chapter examines scaling of WebSocket connections when multiple servers are behind a load balancer. The chapter starts with a background presentation of Internet technologies, and then reviews related work. This is followed by a description of the tools that will be used and the experimental setup created to perform the experiments. Next the experiments are described step by step as we build up the architecture shown in Figure 2.1. The chapter concludes with an analysis of the results of these experiments.

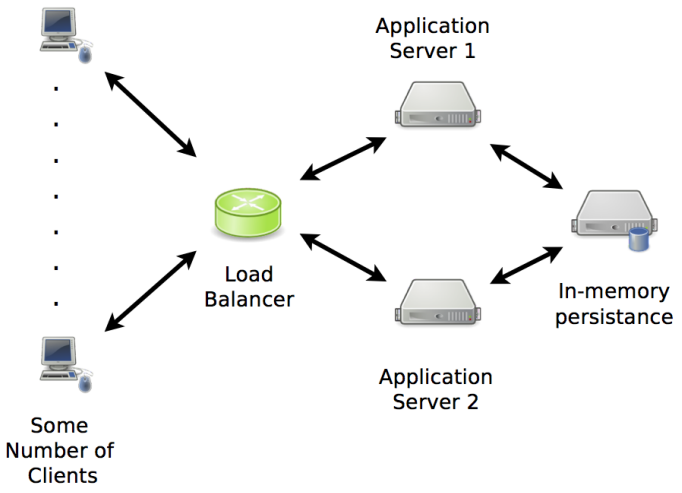


Figure 2.1: Overall view of the architecture for the WebSocket tests

2.1 Background

HTTP [19] was designed for a request and response paradigm for Web access, where a user requests a web page and gets some content in a response. With the need for more interactive web pages, AJAX [20] was introduced to make asynchronous requests without refreshing the whole current web page. However, AJAX was still based upon requests being sent by the client. HTTP was originally designed for document sharing, rather than for interactive applications. Techniques that attempt to provide real-time web applications include polling, long-polling, and Comet [21]. Long polling is an approach where the client keeps a connection to the server open until getting a response or a timeout occurs. Polling methods provide almost real-time communication. Polling and long polling are compared in Figure 2.2. However, the problem with polling is that this approach is not suitable for low latency applications because of the overhead of HTTP's header data. In addition, the client must wait for responses to return before it can send a new request which increases the latency. Hence, some new protocols have been introduced to overcome this limitation. This chapter explores one of these, specifically WebSocket.

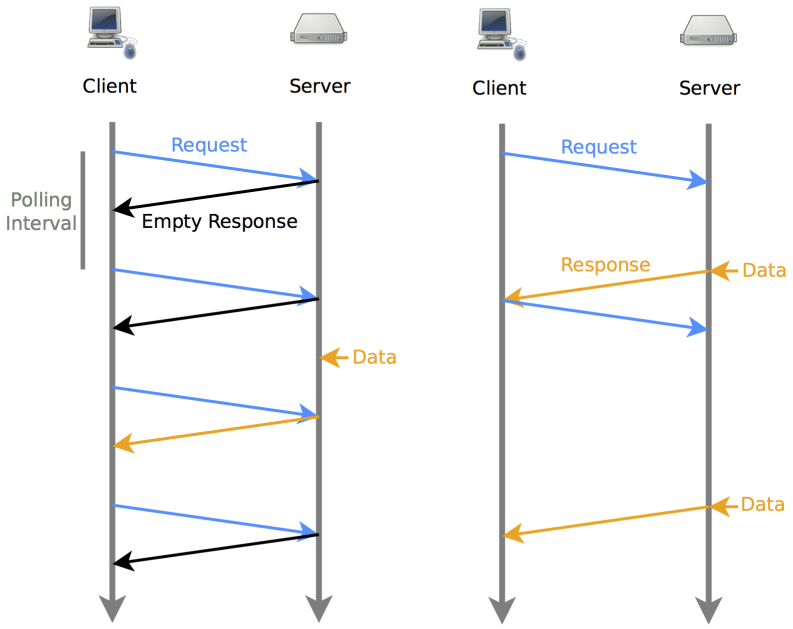


Figure 2.2: Polling (shown above on the left) versus long polling (shown above on the right)

WebSocket is a technology which provides a full-duplex channel over a single TCP socket. A WebSocket is a persistent connection over which the server and client can send data at any time. A single request is sent to open a connection and this connection is reused for all subsequent communication. The WebSocket protocol has been standardized by the Internet Engineering Task Force (IETF) in RFC 6455 [22] and the WebSocket API [23] is being standardized by the World Wide Web Consortium (W3C). Modern browsers support the WebSocket API. The WebSocket API is event-driven, hence there is no need to poll the server. This means that the server can push data to the client (and vice versa) at any time as depicted in Figure 2.3.

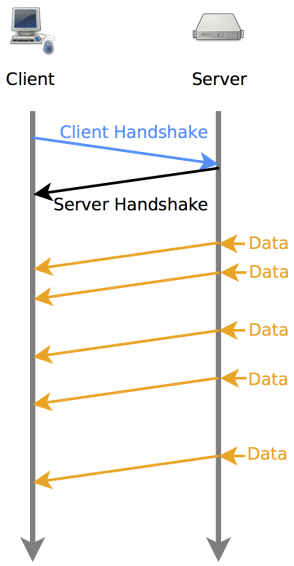


Figure 2.3: WebSocket traffic

WebSocket utilizes a TCP connection. The protocol begins with a HTTP request which is upgraded to the WebSocket protocol during the initial handshake as shown in Listing 2.1. The `Upgrade` header indicates that this connection should be changed to use the WebSocket protocol. The response code, `101`, (shown in Listing 2.2) indicates that upgrade of the connection was successful. The `Sec-*` headers are part of the handshake to confirm that the server understands the WebSocket protocol. After the upgrade, WebSocket messages can be sent with a data-framing format. This data-framing format is necessary because the TCP connection does not have message markers, but rather simply transports a stream of bytes which are delivered in order. To terminate the connection, an

endpoint that wants to close the connection sends a numerical code representing the reason for termination. The details of this protocol can be found in the WebSocket Protocol specification [22].

Listing 2.1: Request to server

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

Listing 2.2: Response from server

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

WebSocket supports sending encrypted traffic over Transport Layer Security (TLS). This is called WebSocket Secure (WSS). TLS is also used in HTTPS, to protect data confidentially and to verify its authenticity.

Today many companies providing real-time web solutions take advantage of WebSocket [24, 25, 26]. For instance, Xively, which is a secure, scalable platform as a service that connects devices with applications, provides WebSocket support to provide real-time control and data storage [25]. In his doctoral dissertation, Dominique Guinard proposes to add support for WebSockets to the IoT architecture in order to offer a web based real-time eventing mechanism to communicate with IoT [14]. Additional related work about WebSocket is discussed in Chapter 2.

2.1.1 Scaling WebSocket Connections

To support a large number of IoT devices, WebSocket servers need to be scaled either out or up. Load balancing solves issues of scalability and availability. Load balancing has been used since the early days of the Internet. In practice, there are two ways of load balancing: hardware or software load balancing. In this thesis project, I use software load balancing due to the ease of its deployment, while its performance is similar to that of hardware load balancing. Software load

balancing can be provided by software (bundled as part of an operating system or software installed as an add-on such as HAProxy). HAProxy will be discussed in Section 2.4.4.

Load balancers can be thought as reverse proxy servers. A proxy server is a server acting on behalf of other computers. A reverse proxy is a proxy that prevents direct access to a website by forcing clients to go through the proxy in order to communicate with the website. The operation of a proxy server is transparent to the client and the client can only see the proxy's IP address(es). A forward proxy is a proxy on the client's side and is placed between the client and Internet. The difference between reverse and forward proxies can be seen in the Figure 2.4.

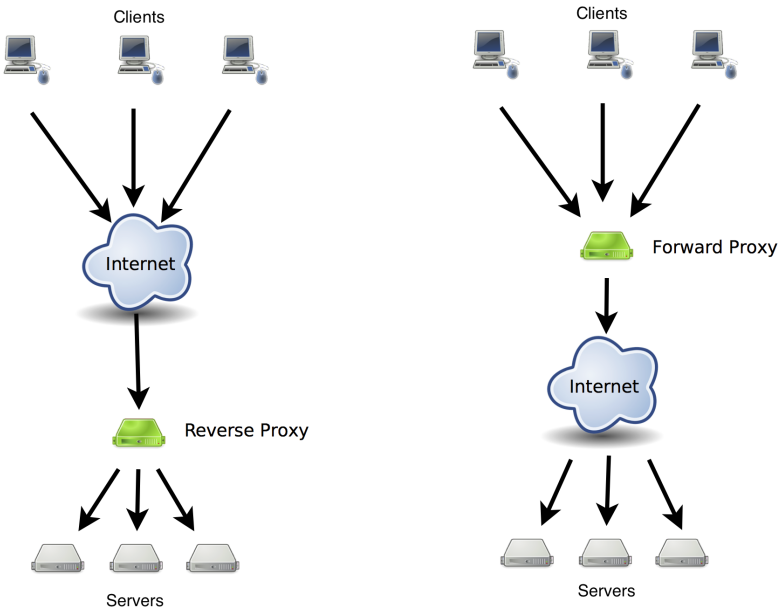


Figure 2.4: Forward proxy acting on behalf of the client. Reverse proxy acting on behalf of server.

Load balancers can make routing decisions based on layer 2, layers 3-4 (IP, TCP/UDP), or layer 7 (mainly HTTP). As described in the previous section, a WebSocket utilizes two protocols: HTTP for setting up the connection and TCP for the actual data exchange (and possibly TLS for security). Therefore, the load balancer needs to forward the TCP traffic to an appropriate server *without* breaking the TCP connection. The TCP connection from the client is to the proxy and not directly to the end server. Hence a new TCP connection is created between the proxy and the server. Therefore the proxy is stateful and must be involved in forwarding all of the traffic flowing from the client to the server. Load balancers

can utilize one of several algorithms to choose a server to route the TCP traffic to. Some algorithms, such as round robin or random choice, are not deterministic and they do not use client side information (such as cookies or IP addresses) to pick the server. There are also deterministic algorithms, such as sticky sessions, that use client side information and choose a given server based on this information.

Load balancing can also offer many additional benefits, such as providing TCP buffering, acting as a firewall, and TLS/SSL acceleration. In the context of this thesis, the main aim of load balancing will be to suitably distribute loads across a number of servers.

2.2 Related Work

This section gives a brief overview of previous projects concerning the WebSocket's performance.

Gutwin, Lippold, and Graham [27] carried out a study to compare the performance of web-based networking methods used in groupware applications. They describe several requirements for different kinds of real-time systems. They show that WebSocket can support most groupware systems, and suggest that the browser should be used by groupware developers. Their studies show that WebSockets can perform better than plug-in approaches, such as Java applets. However, they note that the use of TCP rather than UDP may have latency problems in real-world situations, due to limited bandwidth and varying traffic patterns.

Greco and Lubbers [28] compare the performance of WebSocket and polling. They showed that WebSocket reduces HTTP header traffic and has lower network latency when compared to polling. Their experiments showed a 500 to 1 reduction in overhead and a 3 to 1 reduction in latency. This latency reduction does not reduce the propagation latency of data, as the propagation latency is the same for polling and WebSocket. However, the queueing latency (the time that a server waits before sending a message) is different. In the case of WebSocket, the server can send a message as soon as the message becomes available. However, in the case of polling, the polling interval determines how long the server must wait for a client's request. All of the reduction in latency results from removing the queuing latency.

Puranik, Feiock, and Hill [29] quantitatively compared AJAX and WebSocket by integrating them into a distributed real-time embedded system. They show that WebSocket provides higher throughput (215.44% more) and better network performance (while using 50% less bandwidth). However, these results are for their example application and there is no guarantee that other applications would experience the same performance.

Jomier and Marion [30] present a real-time collaborative visualization tool based on WebSocket and WebGL. Their results show that using WebSocket compared to AJAX has better performance in terms of lower latency.

Autobahn Testsuite [31] is a project to test the correctness of WebSocket protocol implementations and is based upon over 300 test cases. Autobahn includes some tests to measure the round trip time for different message sizes and fragment sizes.

These projects all show that WebSocket performs better than polling in terms of providing lower latency and greater user data bandwidth in most cases. Building on these results, this thesis project uses WebSocket to implement a gateway architecture for IoT. This thesis project describes tests that have been performed to measure the scalability and the latency of this architecture in terms of the following metrics: throughput, message service time (including event loop latency), response latency, and CPU usage. The choice of these scalability metrics are mainly based on Greg Barish's book, *Building Scalable and High-Performance Java Web Applications* [32]. An article about WebSocket test by Cubeia (a software development company focusing on scalability solutions) also leads to these same metrics [33].

2.3 Goal

The goal of these experiments are to stress test the potential WebSocket architecture that has been proposed to support IoT communication. Some benchmarks have already been used to investigate the scalability of WebSockets [34, 35, 36, 37]. Some of these benchmarks use a server that simply echoes incoming messages [35, 36], while some use a server which broadcasts each message to all connected clients [34, 37].

Instead of echoing or broadcasting all messages from all connected clients, in the tests described here the clients are separated into two types: publishers and subscribers. In this way, we can simulate a number of IoT devices which act as publishers sending data to a number of clients that act as subscribers. Figure 2.5 shows the case of a publisher and several subscribers. Subscribers receive messages published to those topics to which they subscribe. The tests will focus on the costs of sending messages from the publishers to the subscribers using the metrics described in the previous section.

The tests follow a quantitative research method with a numerical data analysis and investigate the following:

- How many WebSocket connections can a server support?

- How can load balancers be exploited to scale up the numbers of WebSocket connections that can be supported by a WebSocket architecture?
- What operations create a bottleneck? Is opening connections, holding connections open simultaneously, or sending messages the limitation of a WebSocket server?
- How does the CPU usage of servers change during WebSocket communication?

This thesis project does not give a comparative analysis of different technologies for server side programming languages or load balancers. (A comprehensive project comparing the performance of different server side technologies has been done as part of the TechEmpower web application framework benchmarks project [38].) Instead this thesis investigates a proposal for a scalable architecture that can support a number of IoT devices by presenting data showing the limits of this architecture.

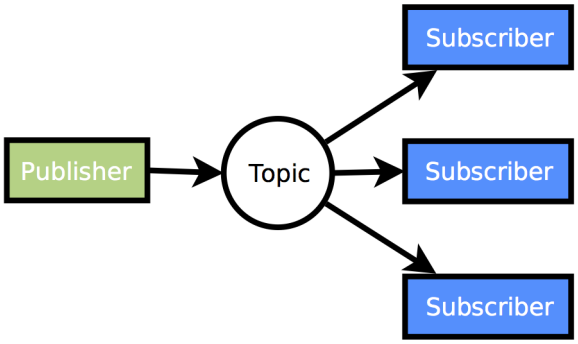


Figure 2.5: Publishers and subscribers

2.4 Tools and Experimental Setup

This section presents the tools used in the tests. The section begins with a description of a number of a useful command line tools, then describes the other software that was used for testing. This section also describes the steps taken to setup an experimental environment in order to perform these tests.

2.4.1 Command Line Tools

A number of command line tools were used for the tests, specifically: SSH, sar, kSar, git, ntp, and gnuplot. Each of these is briefly described below. The

specification of the local computer and the servers used for the test environment can be found in Table 2.1 below and Table 2.3 on page 32.

Table 2.1: Hardware specification of the local computer

Model	MacBook Pro
Operating System	OSX 10.8.3
CPU	2.2 GHz Intel Core i7
Memory	4 GB DDR3

2.4.1.1 SSH

SSH is an acronym for secure shell. SSH is a secure network protocol to connect to a remote machine over an unsecured network. SSH is commonly used to access shell accounts on Unix/Linux systems (in this case a Ubuntu Linux system). Some examples of its usage are:

```
# Connecting to shell server of KTH
$ ssh gunayk@shell.it.kth.se
# Connecting to my Amazon instance with a key file created
   for this instance.
$ ssh -i ./gunay-amazon-s1.pem \
   ubuntu@ec2-54-216-102-8.eu-west-1.compute.amazonaws.com
```

2.4.1.2 Sar

Sar is a tool to monitor performance statistics including: CPU usage, memory consumption, network traffic, etc. Sar was used to collect performance data about the servers during the tests. Sar is part of the linux `systat` package which can be installed with the command *:

```
$ sudo apt-get install sysstat
```

Here are some examples of its usage:

```
#print CPU usage every one second
$ sar -u 1
#print info about sockets in use for IPv4
$ sar -n SOCK 1
#print to file
```

* All of the command examples given are for a Ubuntu linux system.

```
$ sar -n SOCK -o results.sar 1
#read the file to extract info about sockets
$ sar -n SOCK -f results.sar
#read the file to see memory cpu utilization
$ sar -f results.sar
#put the sar data into file
$ sar -o results.sar 1 >/dev/null 2>&1 &
```

2.4.1.3 kSar

kSar [39] is a Java based tool to graph the output of sar. It can export the graph in a variety of formats, such as PDF, JPEG, etc. This tool can be used as shown with the following commands:

```
#print sar data to a text file by using C locale
$  LC_ALL=C sar -f ~/datafile -A > sar.txt
#execute kSar with the sar.txt file given as input and
   sar.pdf as output
$  java -jar kSar.jar -input sar.txt -outputPDF sar.pdf
```

2.4.1.4 Git

Git is an open source distributed version control system developed by Linus Torvalds. The purpose of Git is to store all versions of the source code and easily access any past version. While Git can be used locally, a remote (centralized) repository is needed to protect against data loss. GitHub is a popular service for Git repositories [40]. Git and GitHub were used to provide a backup and to maintain a complete history of the software development for this project. Git was also used to push the local code to the test servers. All the code for these experiments can be found in the public GitHub repository: https://github.com/gmertk [41]. The most frequently used commands during this project were:

```
# to initialize a Git repository which is a hidden
   directory in the folder where Git is executed
$ git init
# to add and commit changed files into the repository.
$ git commit -am "message"
# to add a remote repository to push the local repository
$ git remote add origin
   "https://github.com/gmertk/websocket-test"
```

```
# to push the local changes to the remote repository
   called origin
$ git push -u origin master
# to check for changes on the GitHub repository and pull
   any new changes
$ git pull
```

2.4.1.5 NTP

To synchronize the clocks of each of the computers the Network Time Protocol (NTP) [42] was used. The NTP daemon synchronizes the local system's time with a remote server.

To install this NTP daemon the following command can be used:

```
$ sudo aptitude install ntp
```

After installation, the daemon can be started or stopped with the following commands (as usual for Linux services):

```
$ sudo /etc/init.d/ntp start
$ sudo /etc/init.d/ntp stop
```

In the tests, one of the servers was chosen as master NTP server, and the other servers synchronize to this master to guarantee that all of the servers have their time set to the same value. The advantages of this setup are a reduced number of outgoing connections [43]. The master NTP server was configured to synchronize with a stratum 1 server (nist1-sj.WiTime.net). `/etc/ntp.conf` file was configured to set the server addresses. Before the tests, the time difference between the servers was below 1 ms as established from the output of the `ntpq -p` command.

2.4.1.6 Gnuplot

Gnuplot [44] is a widely used and powerful tool to generate plots of data. It can export the plot in a variety of formats, such as PNG, JPEG, etc. The test results, presented in Section 2.6, were plotted using gnuplot.

2.4.2 Node.JS Server

Node.js is an event-driven, non-blocking infrastructure for building highly concurrent software. It is used by many large companies to create fast and scalable networked

services [45]. Node.js is written in JavaScript and it provides elegant APIs and has a large number of third-party modules available for it.

Event-driven programming is a programming model where events determine the flow of the program. Events are handled by callbacks which are functions that are invoked when an event happens, such as when a connection occurs or when a database query produces a result. The example below shows how event-driven programming is different from traditional blocking input/output (I/O) programming [46]. In traditional blocking I/O programming, a database query stops the current process until the database processing is completed:

```
result = query('SELECT * FROM users WHERE name = "mert" ');
log(result);
```

In event-driven systems, logging this query would be written as:

```
queryFinished = function(result) {
      log(result);
}
query('SELECT * FROM users WHERE name = "mert"',
   queryFinished);
```

In the event-driven approach when the query has completed, the callback `queryFinished` function will be called. This model of programming is called event-driven or asynchronous programming. Event-driven programming is one of the defining features of Node.js. JavaScript is well suited for event-driven programming, because it supports closures and functions as arguments.

Event-driven programming is realized by having an event loop which detects events and invokes callbacks when a specific type of event happens. An event loop is simply a thread running inside a process, thus when an event happens the event handler runs without requiring an interrupt. In this model at most one event handler is running in a process at any given time, thus the programmer does not have to consider concurrent threads of execution changing the shared memory state as they would have to do in multi-threaded programming. However, this also is a limitation since only one handler is running and it runs to completion - therefore other events have to wait until this event handler has voluntarily given up the CPU.

Node.js can be installed on Ubuntu via the package manager `apt-get` by entering the following commands:

```
$ sudo apt-get install -y python-software-properties
#this command installs the latest stable version from
   Chris Lea's Ubuntu Personal Package Archives (PPA).
#node v0.10.5 was the stable version when I performed the
```

```
   experiments.
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
#npm is the node package manager to install third-party
   modules
$ sudo apt-get install -y nodejs npm
```

Node.js has many packaged modules which can be installed via the npm package manager. The modules that have been used in our testing are Optimist, Gauss, Socket.io, Websocket-worlize, Forever, and Node-redis. Each of them is described in more detail below.

Optimist – Optimist is a Node.js module for simple option parsing. With optimist, options are similar to a dictionary. An example from the optimist website [47] is shown below. The file short.js contains:

```
#!/usr/bin/env node
var argv = require('optimist').argv;
console.log('(%d,%d)', argv.x, argv.y);
```

```
$ ./short.js -x 10 -y 21
```

To install:

```
$ npm install optimist
```

Gauss – Gauss makes it easy to calculate and explore collected data by providing functions for data analysis such as standard deviation and arithmetic mean [48].

To install:

```
$ npm install gauss
```

Socket.io – Socket.io is a popular JavaScript library which simplifies the usage of WebSocket and provides fail-overs to other protocols such as Adobe Flashsockets, JSONP polling, and AJAX long polling. It also offers heartbeats, timeouts, and disconnection support, none of which the WebSocket API provides directly. Initially the testing used Socket.io because I thought Socket.io was better because of its support of old browsers and it provided a better abstraction for native WebSocket API. However, I subsequently realized that the Socket.io library has a significant problem which causes it to disconnect clients when there was a large number of connections [49]. Unfortunately, this issue is not fixed yet, so I changed to another WebSocket library (Websocket-worlize - described next).

Websocket-worlize – Websocket-worlize is a pure JavaScript implementation of WebSocket for Node.JS. It provides both a WebSocket server and client library. I chose to use this library because of its pure design and interoperability with load balancers.

To install:

```
$ npm install websocket
```

The following commented example for an echo server is modified from the module's documentation [50]:

```
#!/usr/bin/env node
var WebSocketServer = require('websocket').server;
var http = require('http');

var server = http.createServer(function(request, response) {
    // console.log prints on the command line.
    console.log((new Date()) + ' Received request for ' + request.url);
    response.writeHead(404);
    response.end();
});
server.listen(8080); // Start the server listening on TCP port 8080

wsServer = new WebSocketServer({
    httpServer: server
});

// Here we can see a clear example of event-driven programming.
// When a request occurs, call the function to open a WebSocket.
wsServer.on('request', function(request) {
    var connection = request.accept('echo-protocol', request.origin);
    console.log((new Date()) + ' Connection accepted.');

    // When a message arrives, call the function to echo it.
    connection.on('message', function(message) {
        // Message type can be text or binary
        if (message.type === 'utf8') {
            console.log('Received Message: ' +
```

```
                message.utf8Data);
28          connection.sendUTF(message.utf8Data);
29      }
30      else if (message.type === 'binary') {
31          console.log('Received Binary Message of ' +
                message.binaryData.length + ' bytes');
32          connection.sendBytes(message.binaryData);
33      }
34  });
35
36  // When the socket is closed, call the following
        function.
37  connection.on('close', function(reasonCode,
        description) {
38      console.log((new Date()) + ' Peer ' +
            connection.remoteAddress + ' disconnected.');
39  });
40 });
```

Forever – Simply running Node.js scripts with the `node` command, the server will start as a process that will block the current shell until the process crashes or is killed. To run the server in the background, a forever module is used. Its purpose is simply to run the server script continuously. If a flag is set when the module is invoked, then the forever module can also automatically restart the script in case of an exception.

To install:

```
1 $ sudo npm install forever -g
```

An example of using the forever module is:

```
1 #in case of crash, restart the script a maximum of 3 times
2 $ forever -m 3 simple.js
```

Node-redis – Redis [51], a key-value store, will be discussed in section 2.4.5. This is a complete Redis client for Node.js which supports all Redis commands. Installation:

```
1 $ npm install redis
```

The publisher-subscriber (pub-sub) architecture was introduced in the beginning of this chapter. A simple example of a publisher-subscriber service is:

```
1   var redis = require("redis"), client1 =
```

```
    redis.createClient(), client2 = redis.createClient();

// When client1 subscribes to the channel, client2
    starts to publish messages to this channel.
client1.on("subscribe", function (channel, count) {
    client2.publish("a channel", "a message");
});

// When client1 gets a message from the channel, it
    logs those messages.
client1.on("message", function (channel, message) {
    console.log("client1 channel " + channel + " " +
        message);
});

// Client1 subscribes to the channel
client1.subscribe("a channel");
```

Node-toobusy [52] – Node-toobusy module checks the Node.js event loop and keeps track of event loop latency, which is the amount of time that the event queue is behind. Node.js used to have libev which is a high-performance event loop [53]. Since libev ran only on Unix, libuv which is an abstraction around libev was used to support Windows platforms. The resulting libuv [54] is a multi-platform library with a focus on asynchronous I/O. As libuv became a standalone library, libev was removed in the node-v0.9.0 version of libuv [55]. The current libuv has an event loop and callback based notifications, thus, the asynchronous style of Node.js is provided by libuv.

Node-toobusy was used in the tests to measure how much lag the event queue has. The time returned is the maximum delay in the event queue. Node-toobusy provides a threshold `maxLag` in order to stop processing more requests. In this project, the lag was always checked by the test script and the `maxLag` was not used to stop processing.

To install:

```
$ npm install toobusy
```

Event loop latency can be received with:

```
var toobusy = require('toobusy'),
var lag = toobusy.lag();
```

2.4.3 Amazon Web Services

This section describes how the Amazon instances were created for running the tests described in section 2.5. I chose to use Amazon Web Services (AWS) due to its affordable prices and popularity. AWS is an infrastructure-as-a-service (IAAS) provider, where a virtualized computer is rented. Virtualization enables a single computer with eight processors to seem like eight independent computers or even more - since the hypervisor (Xen in AWS) can be running many virtual machines per physical processor.

There are many AWS terms that the reader should be familiar with before going further in this thesis. These include:

EC2 – Amazon Elastic Compute Cloud (EC2) is a web service to launch and manage Linux/UNIX and Windows server instances in one of Amazon's data centers.

AMI – Amazon Machine Image (AMI) is a snapshot of a (virtual) computer's root drive. These images contain the operating system, installed software, and configuration files. When a virtual machine instance is launched, it is launched from an AMI.

Security groups – A security group defines a set of protocols, ports, and IP address ranges to define firewall rules for instances. Several instances can use the same security group. For the purpose of the tests described in this thesis a security group was created that allows SSH, HTTP, and NTP connections.

Key pair – A key pair is essential for communicating with a fresh instance when it is launched. A SSH key pair with a public key is stored in the instance, while the private key is stored in my local computer. More information about public key cryptography can be found in the book *"SSH, The Secure Shell: The Definitive Guide"* [56].

Regions and availability zone – AWS resources are located in different regions (such as EU West - Ireland, US West - California, etc.). A region comprises at least two availability zones which are distinct locations within a region in order to make the infrastructure more resilient to power failures and network outages.

2.4.3.1 Problems with Virtualization

While the virtual computers have reasonable performance, their performance is worse than that of native hardware. Wang and Ng carried out a study to analyze the impact of virtualization on the network performance of Amazon EC2 [57]. Their measurement shows that virtualization can cause throughput instability and abnormal delay variations, especially on EC2 small instances due to processor sharing. In the virtualized EC2, the delay caused by virtual machine scheduling

can be much larger than the other delay factors such as propagation delay and router queuing delay. Their analysis concludes that virtualization and processor sharing cause unstable network performance.

Another issue is that AWS does not guarantee that the instance is not placed with a noisy neighbor. The virtual machines that work on the same hardware may steal other virtual machines' share of the physical CPU. This is called the noisy neighbor problem and can be measured with "Steal Time" in CPU statistics. Noisy neighbors may also cause abnormal and variable performance.

Although AWS is subject to these kinds of problems, this thesis project uses AWS due to its affordable prices and ease of scaling up and out. High capacity instances were chosen to reduce the problems and in the tests, thus the "Steal Time" was measured to always be under %2 for the instances that were used in the measurements described later in this thesis.

2.4.3.2 Setting Up Servers

To start using an AWS cloud computer is pretty straightforward. One goes to the AWS website (aws.amazon.com) and creates an account. Then, using the management console the user launches a new EC2 instance, for example an instance of Ubuntu Server 12.04.2 LTS. (The next section will give a detailed specification of the instances that were used for testing). Next using the management console, the user creates a key pair for use with SSH. One connects to the instance using SSH and installs the software that he or she wants to run, in this case the required tools for the tests were installed. After installing these tools, an image of this instance (i.e. an AMI) can be created in order to easily instantiate further instances. Four custom AMIs were created with different configurations and tools: one for application servers, one for client servers, one for a load balancer, and another for a persistence (i.e. stable storage) layer.

Application Server and Client Server instances each included `Git`, `sar`, `Node.js`, and the Node.js modules introduced above. Some customizations were done on these servers. For example, each TCP connection has a file descriptor open in the file operating system. It is important to set the limit of the number of such open files to a value larger than what will be needed during testing. By default Ubuntu only allows 1024 file descriptors to be open at any one time. This number of open file descriptors is insufficient for stress testing purposes. To set this limit permanently in Ubuntu, the file `/etc/security/limits.conf` is modified to include the lines shown below [58]. The value has been set to a very large number so that these limits will not be a bottleneck in the tests. However, this is for stress testing purposes and may not suitable for a production server. Because this number of sockets will use a large amount of memory which is not swappable, this large number of sockets may decrease the actual performance of

the server with respect to an optimally tuned server.

```
* soft nofile 999999
* hard nofile 999999
```

After restarting the instance the new limit can be displayed with the commands below:

```
$ ulimit -a
$ ulimit -n
```

The **ulimit -a** command reports all limits such as maximum file size, maximum size of virtual memory, etc. The **ulimit -n** command only reports the maximum number of file descriptors.

To avoid limitations of the operating system's default settings and to improve general performance, some other system options are tuned by editing the `/etc/sysctl.conf` file. These options are used for the purpose of stress testing [59] and can be found in Appendix A.

The **load balancer** instance included `sar` and `HAProxy`. Further details of HAProxy will be given later in section 2.4.4.

The **Redis layer** instance in addition to `sar`, has `Redis` installed.

To simulate a large number of clients on one machine is not a realistic approach. Because, that machine's limitations in terms of memory, CPU, or network throughput will limit the load that it can generate. The Bees with Machine Guns tool was used to create a number of instances to simulate clients.

Bees with Machine Guns [60] is a tool to create and control EC2 instances remotely. It enables the user to run any number of clients (bees) immediately and execute commands on them. The tool is implemented as a python program and can be installed by the python package manager `pip`:

```
$ pip install beeswithmachineguns
```

A typical bees usage is:

```
# Create 10 instances in the region called us-east-1a.
# Use the default security group, the pem file called
   beeskeypair, and the AMI called ami-621b630b
# Login with the user name ubuntu.
$ bees up    -s 10 \
            -g default \
            -k beeskeypair \
            -i ami-621b630b \
            -z us-east-1a \
```

```
            -l ubuntu \
# Execute the command, node load.js, in all of the
   instances and prints the output to response.txt
$ bees exec -o "response.txt" - "node load.js"
# Terminate all of the instances
$ bees down
```

2.4.4 Load Balancer: HAProxy

HAProxy [61] is a high-performance software based TCP and HTTP load balancer. It has been used by the biggest companies in the industry, such as Twitter, Instagram, etc. [62]. HAProxy is very configurable and provides many load balancing algorithms, including `roundrobin`, `leastconn`, and more [63]. Detailed documentation about this load balancer can be found on HAProxy's website [64]. HAProxy was installed on the load balancer instance with the following commands:

```
# get HAProxy version 1.5
$ wget
   http://haproxy.1wt.eu/download/1.5/src/haproxy-1.5-dev18.tar.gz
$ tar -zxf haproxy-1.5-dev18.tar.gz
$ cd haproxy-1.5-dev18
# build and install
$ make
$ cp haproxy /usr/sbin/haproxy
```

The configuration file `haproxy.cfg` was modified and the HAProxy was restarted:

```
$ sudo vim /etc/haproxy/haproxy.cfg
# to restart
$ sudo service haproxy start
# or
$ sudo haproxy -f /etc/haproxy/haproxy.cfg
```

The configuration file used in the tests can be found in Appendix B. This file has several sections that are of interest to us:

global Parameters in this section are process wide and global settings for the configuration.

defaults This is the section where common parameters are set for all other sections.

frontend Incoming client connections are listened to in frontend sections.

backend Backend is a section which is used to define a list of servers to be used with a load balancing algorithm, health check configurations, etc. Incoming connections are forwarded to these servers.

listen Listen sections are a combination of frontend and backend sections.

A simple configuration file from the HAProxy wiki [65] is shown below:

```
# An HTTP proxy listening on port 80 in all interfaces
# and forwards requests to a single backend called "servers" with a
# single instance called "server1" listening on 127.0.0.1:8000
global
   daemon
   maxconn 256 #is the maximum number of concurrent connections. The number is chosen randomly for this example.
frontend http-in
   bind *:80
   default_backend servers
backend servers
   server server1 127.0.0.1:8000 maxconn 256
```

As shown in the previous chapter, a WebSocket starts as a HTTP request such as the one shown below:

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Host: localhost:8080
Sec-WebSocket-Protocol: echo-protocol
```

The `Connection:Upgrade` and `Upgrade:websocket` headers tell the server to change to the WebSocket protocol. When the server acknowledges with a status code `101`, the TCP connection used for the HTTP request is re-used for the WebSocket data exchange. The HAProxy is able to switch a connection from HTTP to TCP without breaking the TCP connection and HAProxy implements timeouts for both protocols. During the first request and response, the HAProxy acts as a HTTP server. When the HAProxy detects the `Connection:Upgrade`

header line, it switches to being a TCP proxy when the server responds with a success code. From that point on, the HAProxy simply operates in tunnel mode and does not analyze or interact with the coming TCP data stream, it simply forwards the data stream.

This behavior is configured in the config file by using the lines below. The access control list (`acl`) keyword defines a test criteria with sets of values and the proxy only performs actions if the criteria is true.

```
acl ws_upgrade hdr(Upgrade) -i WebSocket
use_backend ws if ws_upgrade
```

There are several other ways that the HAProxy can proxy WebSocket connections [66], specifically URI based and sub-domain based.

URI based – HAProxy can direct the traffic based on the URI such as `example.com/websocket`. This is actually a path based means of identifying which traffic to proxy. An example is:

```
acl ws_uri path_beg -i /websocket
use_backend ws if ws_uri
```

Sub-domain based – If the server for WebSocket connections is in a separate sub-domain, then HAProxy can direct connections for this specific sub-domain to a server. An example for the domain `websocket.example.com` is:

```
acl ws_subdomain hdr_end(host) -i websocket.example.com
use_backend ws if ws_subdomain
```

For the purposes of our tests the WebSocket detection method was configured. This approach was selected because I did not have any subdomains or specific URIs to use for the test setup.

Another well-known load balancer is Nginx. HAProxy and Nginx are similar in terms of performance [67]. However, Nginx only started to support WebSockets in its later versions [68].

2.4.5 Redis Layer

In order to reliably send messages to users across servers, a layer was needed to allow asynchronous messaging. Redis was used as a pub-sub service as it natively supports pub-sub operations. Redis is a key-value in-memory persistent data store. While Redis holds all the data in memory, it can write this data to disk for true persistence. Based on how many keys have changed, Redis writes the memory to disk. In addition to a pub-sub service, Redis actually provides five data structures:

strings, sets, lists, hashes, and sorted sets. Further details can be found in the Redis reference documentation [69].

Redis has two kinds of persistence modes. Snapshotting mode and append only mode. Snapshotting mode produces snapshots of the data when some configured conditions occur. For example, Redis can be configured to create a snapshot every N seconds if there are at least M changes in the data. Snapshots are created as `.rdb` files, thus this mode is also known as RDB (Redis database). The append only mode logs every write operation to an append-only file (AOF) and this log is re-played when Redis is restarted. Both of these modes have their own advantages and disadvantages. For more information, see the Redis persistence documentation [70]. In the tests described in this thesis, Redis is used only as in-memory database and the persistence was disabled by removing the `save` lines in the configuration file `/etc/redis/redis.conf`. This is because Redis was used only for its pub-sub service, i.e., without any persistence.

In order to install and use Redis, one enters the following commands into a persistence layer instance:

```
1 #In the tests Redis v2.6.14 was used
2 $ wget
    http://download.redis.io/releases/redis-2.6.14.tar.gz
3 $ tar xvzf redis-stable.tar.gz
4 $ cd redis-stable
5 $ make
6 #Starts Redis server
7 $ redis-server
```

During the tests, the Redis instance was monitored by redis-stat tool [71]. This tool is is based on Redis's `INFO` command [72], and does not affect the performance of the Redis instance.

2.5 Data Collection

This section explains how WebSocket connections were tested and how the data was collected.

2.5.1 Test Scripts

There were two Node.js scripts: `load.js` and `app.js`. Each of these scripts will be described in detail in the paragraphs below.

2.5.1.1 load.js

The load script is designed to simulate an IoT device which generates messages and publishes these messages at a configurable frequency to a subscriber. The script creates a number of publishers and a subscriber per publisher. The script sends messages via each publisher's connection and messages are received via each subscriber's connection.

The script is executed with a command, such as:

```
$ node load.js -t 2 -n 100 -h localhost -p 8080
```

The command above creates 100 publishers and one subscriber per publisher, where each individual publisher's connection publishes a message every 2 seconds. After each connection is established, the publishers starts publishing messages. In this example, these numbers were chosen simply to explain the parameters of the script. The `-h` and `-p` options specify the server's IP address and port number to use for initiating a connection. The script calculates the latency of receiving messages from the publisher. This latency is the time for a message to propagate from a publisher to a subscriber. Each subscriber extracts a timestamp from the message and calculates the time difference between this timestamp and the time when this message was received.

The messages sent by publishers are JSON objects (converted to a string by the `JSON.stringify` method). These messages represent information that could be sent to IoT devices. The message format used in the tests is simple and contains only a timestamp and a subject of the publisher. However, Activity Streams [73] could also be used as a generic JSON format to describe an IoT activity in order to provide more data. The Activity Streams format can utilize information from an IoT for other web services and this format has been adopted by major companies such as Google, MySpace, etc. [73].

The JSON message format used in the tests is:

```
{
      type: "publisher",
       subject: "subject00001",  //Each publisher has a
          different subject. The 0's are inserted to have
          a constant message size (70 bytes).
       published: +new Date()
}
```

An example for Activity Streams is:

```
{
    "published": +new Date(), //Unix Timestamp
    "actor": {
      "objectType" : "sensor",
      "id": "1376118068325",
      "displayName": "Temperature Sensor"
    },
    "verb": "post",
    "object" : {
      "value": 25,
      "unit": "C"
    }
}
```

2.5.1.2 app.js

The server script is the main application which is executed in the application servers. The script receives publisher messages and forwards these messages to subscribers with the help of Redis. The script logs the event timestamps to calculate the message service time and loop latency. The IP address and port number of the Redis server are the only parameters.

```
$ node app.js -h localhost -p 6379
```

The parameters of all of the scripts are listed in Table 2.2.

Table 2.2: Parameters of the scripts

load.js	-t	Message period in seconds
	-n	Number of publishers
	-h	Address of application server
	-p	Port of application server
app.js	-h	Address of Redis server
	-p	Port of Redis server

2.5.2 Test Servers

The scripts are executed in EC2 instances having the specifications shown in Table 2.3. The details of the HAProxy and Redis servers are also presented in this table. More information about these types of instances can be found in the EC2 documentation [74]. According to Amazon, “The amount of CPU that is

allocated to a particular instance is expressed in terms of these EC2 Compute Units (ECU). One ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor." [74]. The reason for locating the clients in another site was to increase the communication delay to a value that would be more representative of an actual usage case. It would be unrealistic to limit the location of the IoT devices to being within a single cloud data center. The client instances were chosen larger than the application instances to ensure that the tests are not limited by the number of client instances which generate the load. Two client instances were used and in Section 2.6.6 it can be seen that this number of client instances were sufficient to generate the target load.

Table 2.3: Specification of the Instances

Instance	Family	Type	ECU	Memory (GiB)	Region
Client instances	General purpose	m3.2xlarge	26	30	US East
Application instance	General purpose	m1.large	4	7.5	US West
HAProxy instance	General purpose	m1.large	4	7.5	US West
Redis instance	Memory optimized	m2.xlarge	6.5	17.1	US West

Operating system of the all instances is Ubuntu Server 12.04.2 LTS

2.5.3 Testing

An IoT device might be anything: an accelerometer, a gyroscope, a thermostat, a light bulb, etc. Depending on the aim of the device, the message rate of the device may vary a lot. The IoT Strategic Research Roadmap report [75] anticipates that IoT devices could be generating a large number of updates, such as 10000 messages per person per day. Priyantha et al. [76] prototyped an application for home energy management and they observed that sensor states for motion sensors and power sensors change only 2 to 20 times a day, that is one message every 100 minutes to 10 minutes. Xively [25], a cloud service for IoT, offers pricing plans with different limits of update rates varying from 10 to 0.1 updates per minute. These projects show that IoT devices' domain is very large and that message rates depend on the usage domain of the device. In the tests done for this thesis, the message rate is 4 Hz per publisher (as specified in the article of Pimentel and Nickerson, "Communicating and Displaying Real-Time Data with WebSocket" [77]).

Testing started with one application server. When this single server hit its CPU limit, a load balancer was added and the same tests were repeated with two servers. This pair of tests enabled us to examine the effects of scaling the number of servers

up from one to two. Four performance metrics were generated: throughput (in terms of number of processed messages), message service time (including event loop latency), response time, and CPU usage.

During these tests, the server(s) were monitored with the sar tool to examine the CPU usage and to know whether the server was overloaded by the arriving and departing messages.

To execute all test cases, a small bash script (test.sh) was written. This bash script executes load.js script with a different number of publishers during the test. When all of the publishers and subscribers are connected, the publishers start to send messages. Latency data is collected for 5 minutes as suggested in Nottingham's blog post [78], and then all connections are closed before starting a new test with an increased number of connections. The time between each test case was 1 minute in order to have no TCP connections in a timewait state. The test was stopped when the application servers had no idle CPU capacity.

To execute the test shell into the client servers which create the load the following command is used:

```
1 #Server to be loaded is given as a parameter to test.sh
2 $ bees exec -o "response.txt" - "./test.sh
    ec2-54-228-107-60.eu-west-1.compute.amazonaws.com"
```

2.6 Data Analysis

This section presents the data that was collected and analyzes the results of the tests that were carried out. The presentation is in six subsections corresponding to the four performance metrics, number of commands processed by the Redis server, and number of events created by the clients. The four performance metrics are throughput, message service time (with event loop latency), response time, and CPU usage. The number of commands processed by Redis server was collected by using redis-stat tool (mentioned in Section 2.4.5). Each subsection compares two cases: (case 1) the case without a load balancer (i.e., a single server) and (case 2) the case with a load balancer and two servers.

2.6.1 Throughput

The load was created by the load script with each publisher sending four messages (corresponding to four message events in the application server) per second. For example, for 1000 publishers there will theoretically be a maximum of 4000 message events per second. Figure 2.6 shows the measured number of events during the test of a single server; while Figure 2.7 shows the case with a load balancer and two servers. It can be seen that a single server starts to diverge from an ideal system at roughly 800 publishers, while the load balanced servers start to diverge at 1600 publishers. The deviation in number of events starts to increase as the number of publishers increases. These critical points should be considered for the other metrics that are described in the following sections. Note that the x-axis of each of these two figures shows the time in seconds since the start of the test.sh script. As stated in section 2.5.3 each test ran for 5 minutes (i.e., 300 seconds) and was separated from the next test by 1 minute (i.e. 60 seconds).

The statistics of the results are presented in Tables 2.4 and 2.5. For the single server, when the number of publisher increased, the difference between mean and expected number of events increases (except for the case for 1000 publishers which has some outliers as seen in Figure 2.6). The difference between the average of actual events and expected events is at most 13. However, the standard deviation increases up to 1387 for 1400 publishers. This increase in deviation will result in an unstable user experience. For the load balanced servers, the difference between the expected and mean number of events is less than 3 until reaching 1800 publishers. After this point, the difference starts to increase. If the load balanced servers are compared to a single server for the same number of publishers, it can be seen that load balanced servers provide better performance in terms of the number of events that are successfully being processed. Apparently it is due to the sharing of the load between the two servers.

As seen in Figures 2.6 and 2.7, there are cases with more than the expected

numbers of events. When the events in the event queue of Node.js starts to build up (because the arrival rate is faster than the processing rate), then the events in the queue may be processed in the event loop's next cycle where there may be less work to do. This unstable number of events in the event queue is because of the overloaded CPU.

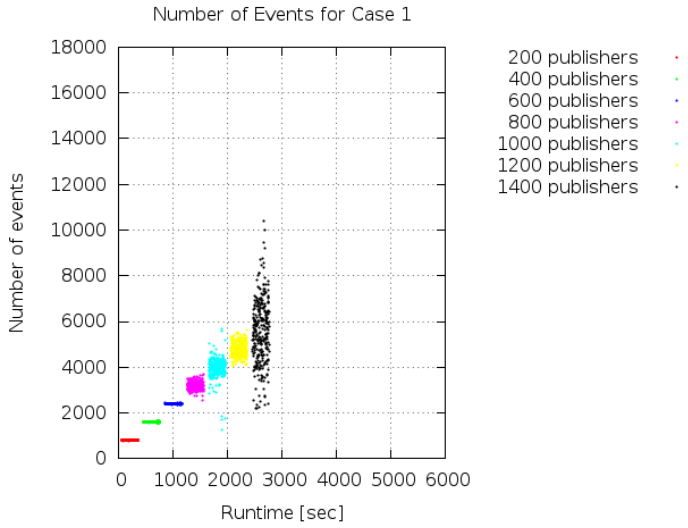


Figure 2.6: Message events for case 1

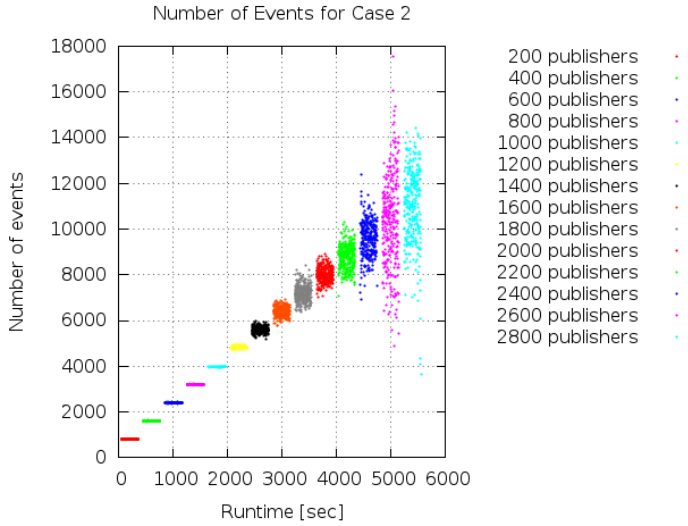


Figure 2.7: Message events for case 2

Table 2.4: Number of events per second for case 1

Publishers	Expected Events	Actual Events
200	800	800
400	1600	1599 ± 11
600	2400	2399 ± 8
800	3200	3198 ± 68
1000	4000	3987 ± 435
1200	4800	4793 ± 309
1400	5600	5587 ± 1387

Table 2.5: Number of events per second for case 2

Publishers	Expected Events	Actual Events
200	800	800
400	1600	1599 ± 5
600	2400	2399 ± 10
800	3200	3199 ± 38
1000	4000	3999 ± 19
1200	4800	4798 ± 85
1400	5600	5598 ± 118
1600	6400	6399 ± 205
1800	7200	7197 ± 355
2000	8000	7993 ± 331
2200	8800	8784 ± 532
2400	9600	9582 ± 779
2600	10400	10357 ± 1993
2800	11200	10974 ± 1681

2.6.2 Message Service Time and Event Loop Lag

Figure 2.8 and 2.9 show the mean and median of the message service time, which includes the queueing time and the time spent in the Redis layer. For a single server, message service time starts to increase at 800 publishers. The difference between mean and median also increases due to large outliers. An exponential growth is seen after 1200 publishers. For two servers the service time starts to increase at 1600 publishers because of the shared load. As seen in the previous throughput metric, the critical point is 800 publishers for the single server and 1600 publishers for two servers. At these critical points the message service time deviates from a constant value. Figures 2.10 and 2.10 show the event loop lag

measured by the node-toobusy module. Until 1000 publishers the loop lag does not show a large growth. At 1200 publishers the mean loop lag increases up to 500 ms and later it increases exponentially. This increase in loop lag effects the message service time as seen in Figures 2.8 and 2.9.

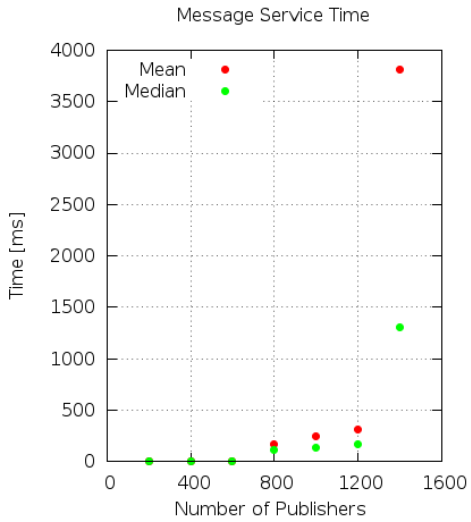


Figure 2.8: Message service time for case 1

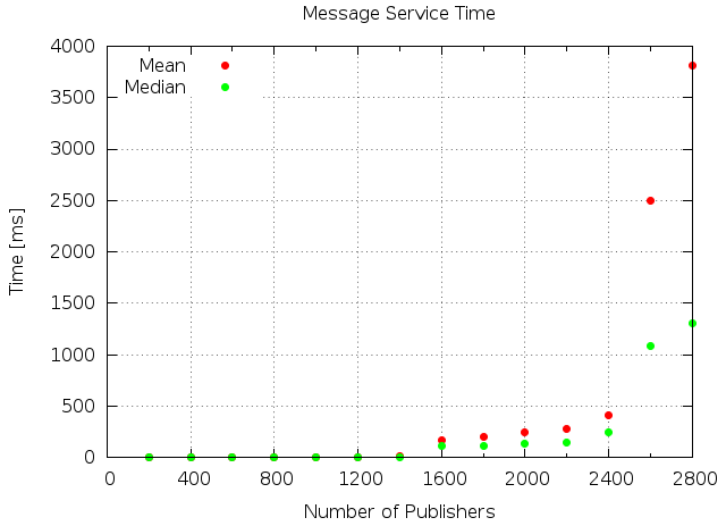


Figure 2.9: Message service time for case 2

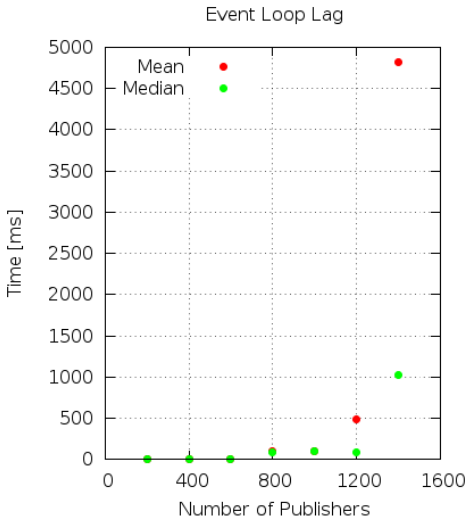


Figure 2.10: Event loop lag for case 1

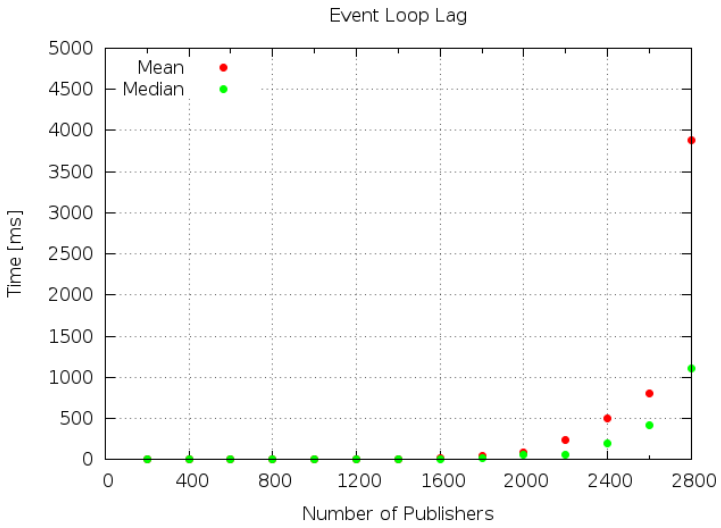


Figure 2.11: Event loop lag for case 2

Tables 2.6 and 2.7 present the statistics of message service time for each test case. With single server the message service time is around 1 ms until 800 publishers. At 800 publishers the mean time of message service time reaches to 166 ms from 2.5 ms. This increase is also observed in standard deviation and median. This is because the servers starts to be overloaded. At 1400, the mean message service time is reaching to around 4 second, which is already beyond to

the requirement of 1 second response delay. There is a large standard deviation after the server is overloaded. This implies that the delay will be unstable. For two servers, the results generates a similar statistics, but with the doubled number of publishers.

Table 2.6: Message service time statistics for case 1

Publishers	**Average (ms)**	**Median (ms)**
200	1.47 ± 0.60	1
400	2.20 ± 1.21	1
600	2.53 ± 1.35	1
800	165.85 ± 96.41	119
1000	249.84 ± 156.41	139
1200	314.16 ± 267.19	174
1400	3806.55 ± 1822.16	1311

Table 2.7: Message service time statistics for case 2

Publishers	**Average (ms)**	**Median (ms)**
200	0.87 ± 0.31	1
400	1.34 ± 0.50	1
600	1.84 ± 1.09	1
800	2.03 ± 1.21	1
1000	2.35 ± 1.20	1
1200	2.60 ± 1.34	1
1400	19.47 ± 10.66	10
1600	174.53 ± 94.12	112
1800	200.44 ± 98.93	118
2000	251.54 ± 147.24	132
2200	279.13 ± 206.45	151
2400	418.06 ± 283.85	244
2600	2503.13 ± 1890.19	1091
2800	3906.06 ± 2013.42	1391

The statistics of the event loop lag are presented in Tables 2.8 and 2.9. For a single server no event loop lag is observed at 200 and 400 publishers. At 600 publishers, there is a negligible lag of 0.22 ms of average. However, at 800 publishers the mean loop lag shows a large increase of up to 100 ms. This is the critical point as seen in the previous metrics. This increased loop lag and also the increased message service time will cause an increase in the response time.

For two servers, as expected there is no loop lag until 1000 publishers. At 1000 and 1200 publishers the loop lag is less than half a millisecond. Then, the loop lag starts to increase as in the case of a single server. At 2800 publishers a large exponential increase is observed up to around 4 seconds.

Table 2.8: Event loop lag statistics for case 1

Publishers	**Average (ms)**	**Median (ms)**
200	0 ± 0	0
400	0 ± 0	0
600	0.22 ± 0.12	0
800	99.39 ± 92.15	93
1000	104.41 ± 101.83	99
1200	490.03 ± 408.52	85
1400	4810.12 ± 3708.52	1023

Table 2.9: Event loop lag statistics for case 2

Publishers	**Average (ms)**	**Median (ms)**
200	0 ± 0	0
400	0 ± 0	0
600	0 ± 0	0
800	0 ± 0	0
1000	0.28 ± 0.25	0
1200	0.38 ± 0.18	0
1400	3.94 ± 1.23	0
1600	30.51 ± 23.68	10
1800	40.41 ± 39.31	25
2000	94.61 ± 71.37	57
2200	236.20 ± 254.12	65
2400	502.16 ± 405.62	201
2600	810.10 ± 606.35	423
2800	3887.89 ± 3401.53	1108

Node.js is single-threaded and scheduling of tasks are handled with a queue. The latency increases with the number of messages waiting in the server's queue to be processed when messages arrive at a faster rate than the processor can process these messages. To calculate the queue size at a certain rate Little's Law can be used. Little's Law says that the average number of items in a queuing system

equals the average rate at which items arrive multiplied by the average time that an item spends in the system [79].

$$L = \lambda W \tag{2.1}$$

L Average number of items in the queuing system

W Average service time in the system for an item

λ Average number of items arriving per unit time

Little's Law can be used to estimate the average queue length for the single server case as shown for some sample message arrival rates below:

$$4 \times 1000 \times 0.249 = 996 \tag{2.2}$$

$$4 \times 1200 \times 0.314 = 1507.2 \tag{2.3}$$

$$4 \times 1400 \times 3.806 = 21313.6 \tag{2.4}$$

The number of messages waiting in the queue increases as the server falls further and further behind. As expected, the service time increases as the number of clients increases. After the critical points, the event loop in the Node.js server is receiving events faster than it can service them and this results in an increase in service time. Unless the sustained rate of message arrivals is below a given threshold (related to the average message service rate), the latency will increase. It should be obvious that since messages are being generated at 4 HZ (i.e., one message per publisher every 250 ms), that when the average service time for a message exceeds 250 ms, the queues will begin to grow.

2.6.3 Response Time

Figure 2.12 and 2.13 show the mean and median of the response time. For a single server, the latency is under 1 second until there are 1200 publishers. It is seen that the difference between the median and mean of response time has increased when the number of publishers is 1400. Thus, there are large outliers that are increasing the mean value. For two servers and the load balancer, the number of publishers is doubled while keeping a similar latency. It can be seen that once the mean response time exceeds 250 ms (at 800 publishers for single server and at 1600 publishers for two servers), the mean response time starts to exponentially increase (which occurs since requests are arriving faster than they can be processed).

Statistics of response time are given in Tables 2.10 and 2.10. As the number of publisher increases, the mean and standard deviation of response time increases. For a single server, it can be seen that the mean response time is under 1 second when the number of publishers is below 1200 publishers. However, at 1000 publishers the mean is 639 ms and the standard deviation is 414 ms, these results imply there can be responses which may take more than 1 second. As expected for two servers, the number of publishers is doubled to 2400 for a mean response time of 1 second.

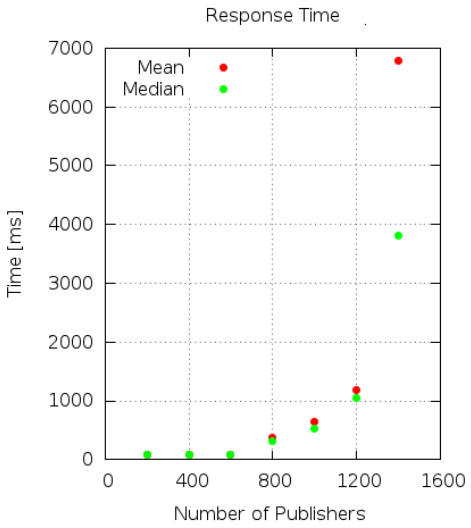


Figure 2.12: Response time for case 1

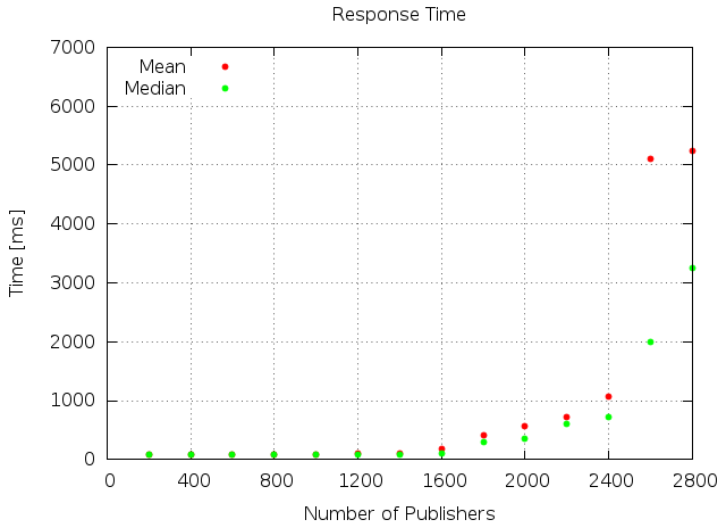


Figure 2.13: Response time for case 2

Table 2.10: Response time statistics for case 1

Publishers	**Average (ms)**	**Median (ms)**
200	83.96 ± 1.94	84
400	87.63 ± 7.57	85
600	86.37 ± 7.24	85
800	372.44 ± 164.03	309
1000	639.25 ± 414.13	520
1200	1179.14 ± 602.81	1054
1400	6787.76 ± 3085.75	3818

Table 2.11: Response time statistics for case 2

Publishers	**Average (ms)**	**Median (ms)**
200	82.95 ± 4.97	83
400	83.56 ± 4.25	84
600	85.32 ± 8.75	84
800	86.72 ± 8.36	84
1000	86.30 ± 9.23	85
1200	95.14 ± 31.52	86
1400	102.91 ± 46.38	87
1600	274.16 ± 194.03	102
1800	405.49 ± 244.61	300
2000	602.29 ± 387.50	400
2200	714.61 ± 549.10	607
2400	1074.33 ± 612.16	727
2600	5103.38 ± 2017.19	1996
2800	5240.46 ± 3515.25	3244

2.6.4 CPU Usage

Figures 2.14, 2.15, 2.16, and 2.17 show the CPU usage as monitored during the tests. Each peak corresponds to a test case with a given number of publishers. In Figure 2.14 for a single server the CPU usage reaches 100% (with around 80% of user load and around 20% of the load being due to processing system calls) at 1200 publishers. Figure 2.15 shows the CPU usage of one of the load balanced servers where the CPU usage for 1200 publishers is around 50% (this occurs because of the fact that the two servers share the load).

Figure 2.16 shows the CPU usage of Redis in the case of two servers. The load increases linearly as the number of publishers increases. At 2800 publishers the Redis server has reached approximately 30% CPU usage. Figure 2.17 shows the CPU usage of the load balancer in the case of two servers. The load balancer reaches 30% CPU usage with a load of 2800 publishers. It can be seen that the load balancer's load and Redis's load increase linearly with the offered load. Thus, the system can support 4 more application servers.

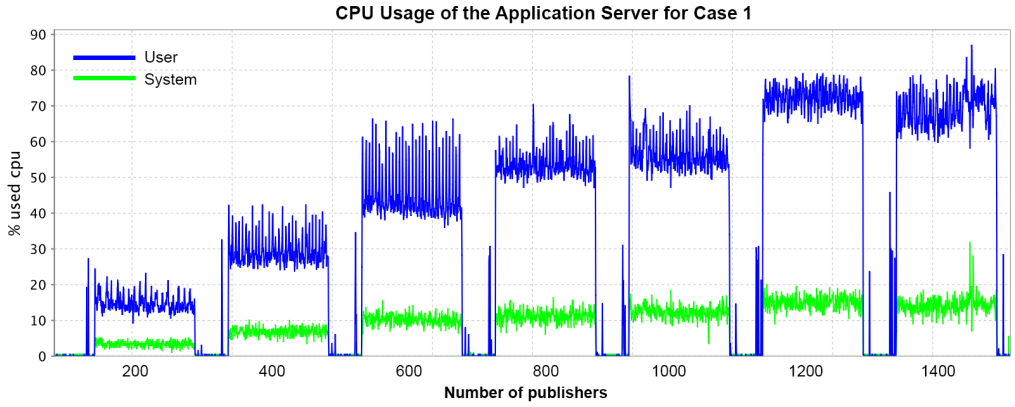
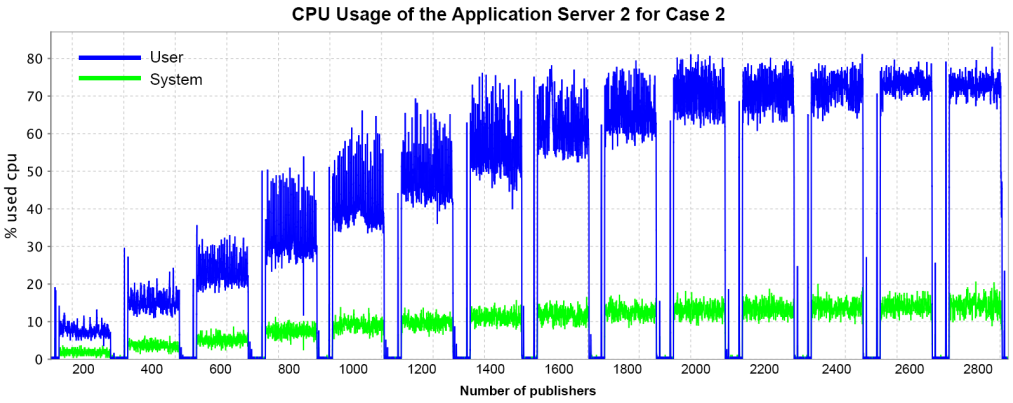
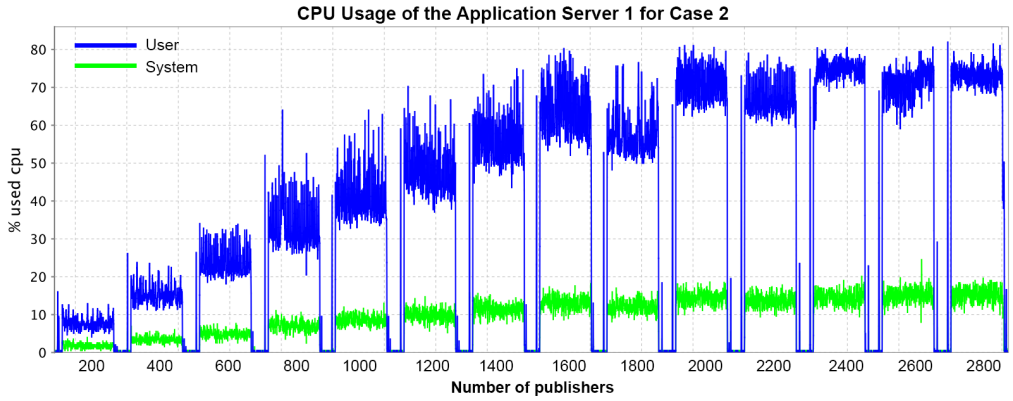


Figure 2.14: CPU usage of the application server for case 1

These figures also provide a comparison of the cost of opening connections, holding connections open, terminating connections, and sending messages. In each of these previous figures, short peaks *before* the publishers start to send messages, correspond to the time when the scripts open WebSocket connections. Then, they start to send messages. This is more easily seen in Figure 2.18 where the parts of Figure 2.14 when messages are being sent are eliminated. The figure only shows the processing associated with opening and terminating connections. It can be seen that at 1400 publishers opening connections consumes up to 45% of the CPU's capacity, while terminating connections consumes up to %25. After the connections are opened, the CPU usage is around 2% while holding these connections open.



(a) Server 1



(b) Server 2

Figure 2.15: CPU usage of the application servers for case 2

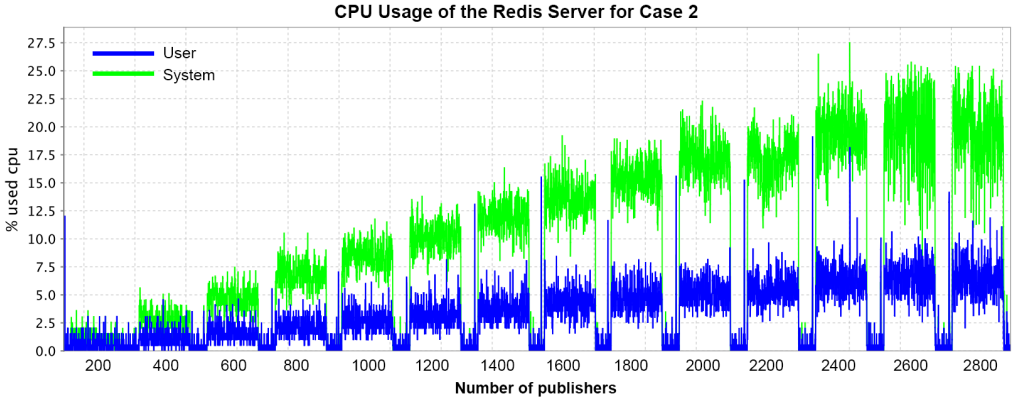


Figure 2.16: CPU usage of the Redis server for case 2

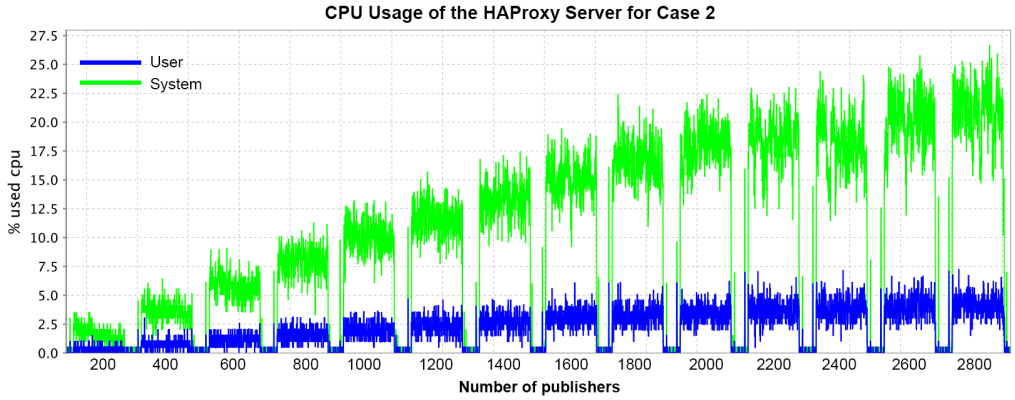


Figure 2.17: CPU usage of the HAProxy server for case 2

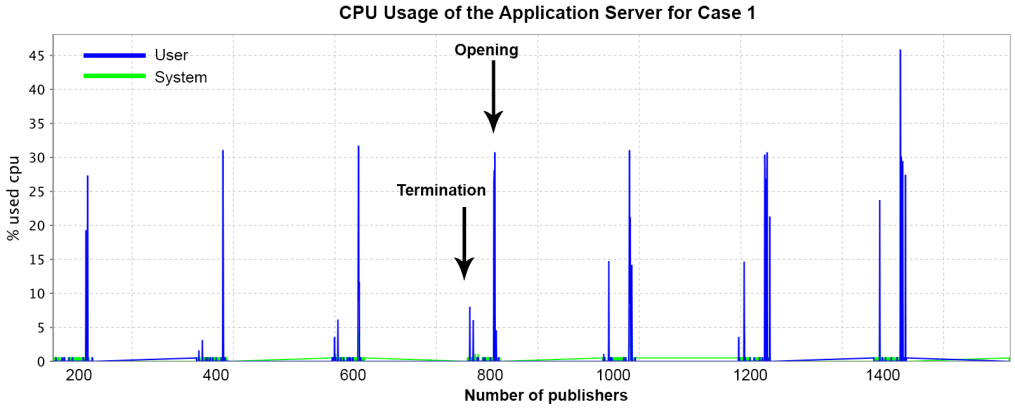


Figure 2.18: CPU Usage of connections for case 1

2.6.5 Commands processed by Redis

The Redis-stat tool collected data about the number of commands processed per second in the Redis server. The results are the same as the first performance metric (i.e., the message events that occurred in the application servers). Thus, Redis was able to handle all of the requests sent by the application servers.

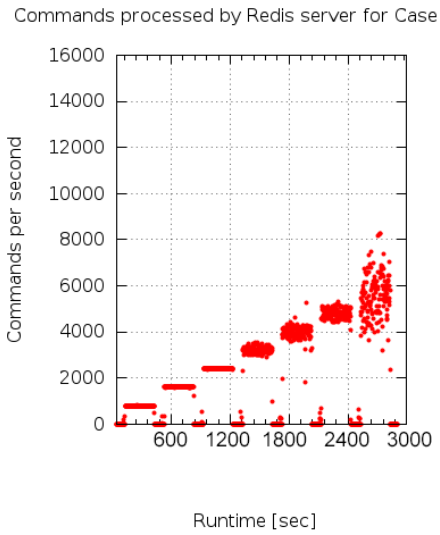


Figure 2.19: Commands processed by the Redis server for case 1

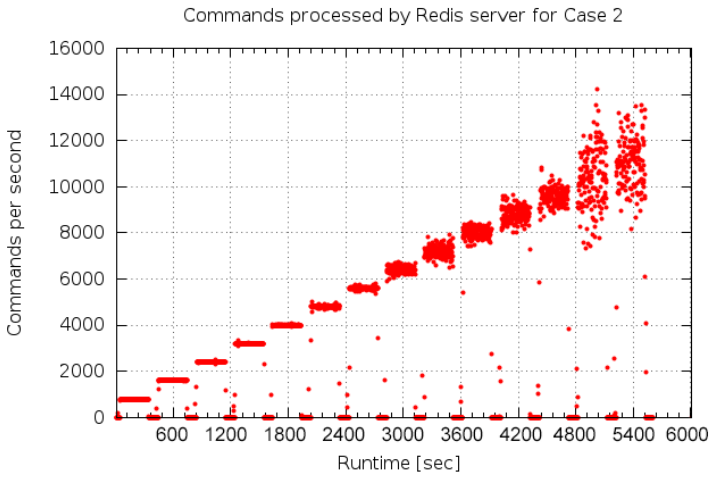


Figure 2.20: Commands processed by the Redis server for case 2

2.6.6 Events in Client Instances

To make sure that the client instances were not overloaded while generating the load on the application servers, the message events of the clients are plotted in Figure 2.21. It can be seen that the number of events do not deviate from the expected number of events, unlike the situation for the application servers. Thus, we are sure that the tests are not limited by performance of the client instances.

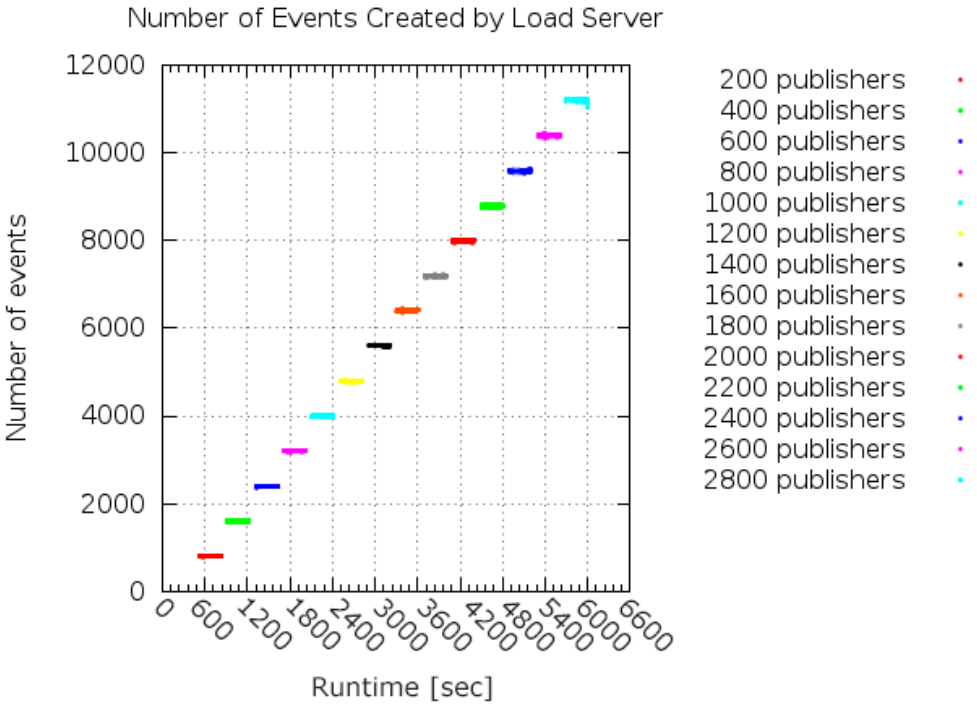


Figure 2.21: Number of message events created by the client servers as they generated load on the application servers

2.7 Conclusion

For the technology probes of ABB project, the response delay needed to be under a second on average and hundreds of simultaneous connections were supposed to be supported. In the experiments, a number of WebSocket connections from EC2 servers were produced in order to simulate Earl connections via mobile phones. The WebSocket protocol was used because of its high throughput and network performance due to reduced HTTP header traffic. The experiments showed that how a load balancer can be exploited to scale up the numbers of WebSocket connections. When each publisher sends four messages per second, the architecture (with two servers and a load balancer) was able to support 1600 publishers and 1600 subscribers simultaneously before the processing of events starts to deviate from the expected performance. The mean response time was under 1 second for these numbers of publisher and subscribers. These numbers can be useful for someone thinking about using EC2 instances to realize such an architecture.

For future projects, a thousands of technology probes were supposed to be supported. It was shown that the load balancer used up to 30% of its CPU while balancing the offered load over two servers and that the load on the load balancer increased linearly with the offered load from the load generators. Thus, we expect that this load balancer could support additional servers and hence the number of the simultaneous clients could be increased by adding more application servers. Furthermore, when this load balancer hits its limits, then multiple load balancers could be used. DNS load balancing with a round-robin algorithm could be used in order to distribute the traffic over these multiple load balancers.

Moreover, the measurements made in these tests depend on the server hardware. For the cost and ease of scaling up servers, EC2 was used in the tests. More powerful native hardware could increase the number of concurrent WebSocket connections.

The results of these experiments also showed that the most costly operation in terms of CPU usage is sending messages, followed by opening connections and then terminating connections. Holding connections open does not use a significant amount of CPU compared to the other operations. From the server's point of view, it is better that IoT devices keep the connection open rather than opening and terminating connections frequently. Note that there will be a problem with this due to the limited number of TCP ports which a given IP address at each host (or proxy) can support.

Future work and required reflections are given in the last Chapter of this thesis.

Chapter 3

WebRTC Experiments

This chapter presents the experiments conducted to test the WebRTC performance on a mobile phone. While the WebSocket tests described in the previous chapter focused on servers, the WebRTC tests described in this chapter focus on testing a mobile phone as a peer in a P2P architecture. To implement the arm-probes which should be able to communicate with each other, a WebRTC DataChannel is used. Stress tests were conducted to see how many peers can be supported by a mobile phone. The results are presented in terms of the number of peers, one-way delay, throughput, CPU usage, and battery power consumption of the mobile phone. The chapter starts with background information. This is followed by a review of related work. Then, the goal of these tests is given. Following this, the WebRTC API is explained. This chapter concludes with a discussion of the data collection and an analysis of the collected data. It should be noted that WebRTC is still under development and Internet browsers are trying to implement draft versions of the WebRTC standards at the time of writing this thesis. The statements made in this thesis may differ from the details of future releases of WebRTC.

3.1 Background

The Web Real-Time Communications Protocol (WebRTC) is an ongoing effort to address the need for real-time communications between web browsers. The WebRTC protocol implementations provides buit-in real-time audio and video functions to browsers without requiring any plug-ins. The WebRTC standards are being developed by both W3C and IETF.

In most web applications, communications occur between a browser and a web server as was the case in the WebSocket protocol. In WebRTC, the communication occurs between two browsers, directly from one browser to another browser, as in Skype [80] or Google Hangouts [81]. This is peer-to-peer (P2P) communication.

WebRTC uses P2P streaming of data. However, a server is required to coordinate the P2P communication between the browsers. This server provides the signaling that is needed to initialize, manage, and close sessions. This signaling can be implemented with any full-duplex channel, such as one running over HTTP or WebSocket. Either a standardized signaling mechanism (such as Extensible Messaging and Presence Protocol (XMPP) [82, 83, 84], the Session Initiation Protocol (SIP) [85]), or a custom signaling mechanism can be used [86]. The WebRTC standards do not define an implementation of the signaling protocol. A simple architecture for a WebRTC application with signaling is depicted in Figure 3.1.

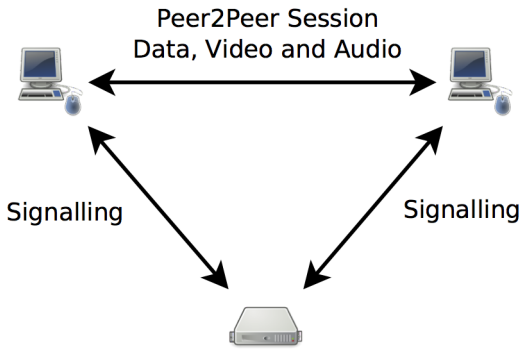


Figure 3.1: Overview of the WebRTC architecture

3.1.1 WebRTC Protocols

This section will give a brief overview of the underlying protocols utilized by WebRTC.

WebRTC can transport audio-video packets and arbitrary non-media data. This thesis will focus on the transport of arbitrary data via WebRTC data channels. These data channels use the Stream Control Transport Protocol (SCTP) [87] over the Datagram Transport Layer Protocol (DTLS) [88] over the User Datagram Protocol (UDP) [89]. This proposed data channel stack is said to provide Network Address Translation (NAT) traversal, authentication, confidentiality, and reliable transport of multiple streams [90]. SCTP is a transport layer protocol that provides reliable transport over an IP network while offering congestion control and multiple streams in a single association. While SCTP is a transport protocol, it cannot be used in many settings directly on top of IP due to the need for NAT traversal. Instead, the higher layer protocol traffic is tunneled over UDP so that NAT traversal may occur. DTLS provides security that prevents data

eavesdropping. UDP transports datagrams and is frequently used for real-time media. The Domain Name System (DNS) [91, 92] is probably the most well known user of UDP, but Real-time Transport Protocol (RTP) [93] is also built on UDP. UDP does not provide delivery guarantees or failure notifications of data transport, which means that unlike TCP there are no acknowledgements, retransmissions, or congestion control. Omitting these features in UDP facilitates data exchange. In this proposed stack, UDP traverses NATs by using Interactive Connectivity Establishment (ICE) [94].

NAT enables devices in a private network to share one public IP version 4 (IPv4) address [95] in order to connect to the Internet. NAT was introduced to delay the problems caused by the limited number of IPv4 address. NAT devices, often implemented within routers, provide a mapping of a local IP address and ports combination to a public IP and port combination. Some IP ranges are reserved to be used as local IP addresses, thus allowing reuse of IP addresses for devices that are going to be connected to a private network, while minimizing the use of the limited number of public IPv4 addresses. This is shown in Figure 3.2. However, NAT causes some difficulties for P2P communication (as a peer acts as both a client and server). One of the issues caused by NAT is that communication initiated by a peer will be blocked by the NAT of the other peer, since there is no suitable mapping created yet between the exterior and interior address and port pairs. Another issue caused by NAT is that the peer only knows its local IP address - hence it does not know its public IP, but yet it needs to provide the other peer with this public IP address. Some prearrangement needs to be done to establish such peer-to-peer communications between devices when both devices are behind NATs. WebRTC uses the ICE protocol which combines a set of methods in order to exchange data across devices behind NATs. ICE utilizes two other protocols: STUN and TURN.

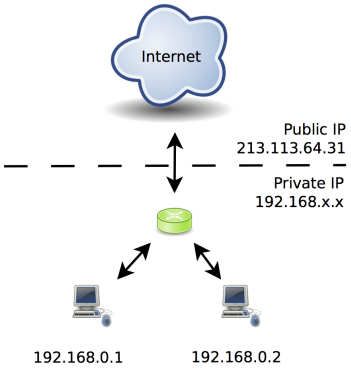


Figure 3.2: Network Address Translation

Session Traversal Utilities for NAT (STUN) [96] is a protocol used to help with NAT traversal. In WebRTC, STUN packets are sent before initiating a P2P session to discover if the peer is behind a NAT and to obtain the IP address and port mapping. The protocol requires a STUN server connected to a globally routable IP address. This STUN servers's IP address must be provided to peers. Peers send a request to a STUN server to discover their own public IP and port. Then they can use the discovered public IP address and port to connect to each other. Unfortunately, STUN works for all but one class of NATs.

Traversal Using Relays around NAT (TURN) [97] is an extension to the STUN protocol that provides a relay for shuttling data between peers as shown in Figure 3.3. TURN requires the relay server to be sufficiently powerful to service all data flows in both direction simultaneously. TURN is a fallback when STUN fails.

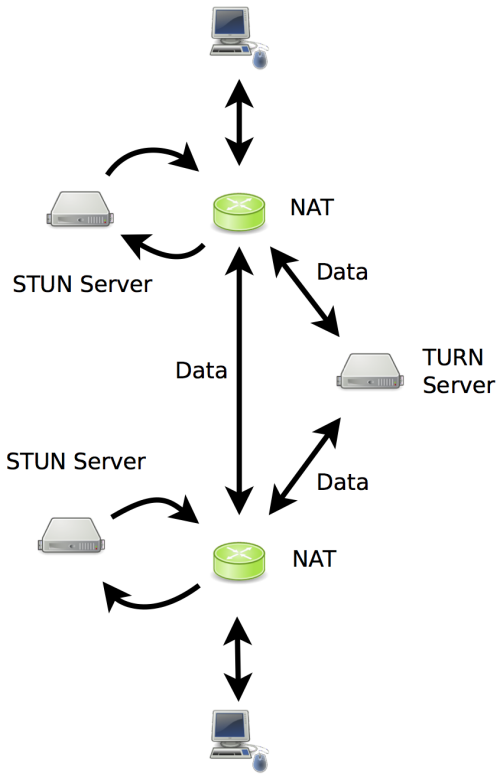


Figure 3.3: TURN and STUN

ICE can be used over IPv6 or IPv4. Lazar states in his blog post "Does WebRTC support IPv6" [98] that if peers are IPv4 or IPv6 capable, a dual-

stack can allow IPv4 session establishment or a client may request IPv6 via the WebRTC's internal API [99]. If one peer is IPv4 and the other is IPv6, a gateway is required to translate the different versions of IP addresses. Adoption of IPv6 continues slowly, and not all Internet Service Providers (ISPs) currently support it.

The WebRTC standards are still in draft form. However, both the Chrome and Firefox web browser development teams started to implement WebRTC before approval of standards [100, 101].

3.1.2 Possible Architectures for WebRTC

WebRTC only supports P2P sessions, but this includes conferencing involving multiple peers. Several P2P architectures for using WebRTC are listed below:

One to One

In the simplest architecture, there is simply one P2P session.

Full Mesh

In a full mesh, every peer communicates directly with every other peer. This is a simple architecture. No server is needed to coordinate the peers (except for the initial signaling). This architecture has the advantage of not requiring a server. However, here every peer sends the same data to every other peer, at some cost in terms of the load on the CPU and network interface, as well as requiring duplicate information to be sent across the communication links. These costs limit the usefulness of this architecture in terms of the maximum number of peers that are feasible. The full mesh architecture is shown in Figure 3.4.

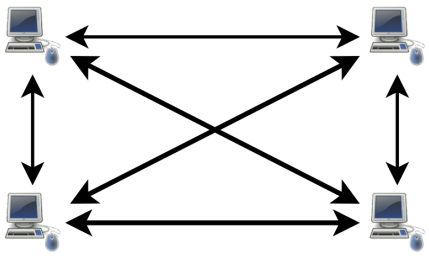


Figure 3.4: Full mesh

Star

Another approach is a star architecture where the most powerful device receives data from all other devices and forwards this data to all of the other peers. The star architecture is shown in Figure 3.5.

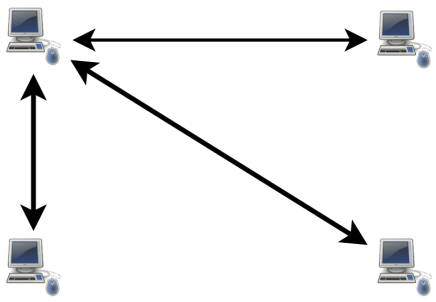


Figure 3.5: Star

MCU

A multipoint control unit (MCU) is a customized server for relaying data. Such a server can provide a more robust architecture. Each peer sends their data to this server, then this server relays this data to the other peers. When new peers join a session, new sessions do not need to be established to the existing peers. A new peer simply establishes a session with the server, while the existing peers continue to utilize the already established session to/from the server. This minimizes the CPU usage by the peers, but increases the load on the MCU and requires the network to and from the MCU to carry all of the traffic N times. The MCU architecture is shown in Figure 3.6.

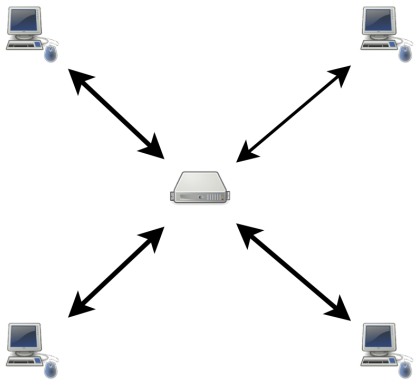


Figure 3.6: MCU

While the MCU is robust for scaling up to a moderate number of peers, the actual scalability depends on the power of the MCU and its network connectivity. In this thesis we will explore the limits of the number of peers that a mobile phone can support in a full mesh network where peers connect to each other directly. The reason for doing this is to measure the limits when using a mobile phone as a gateway in terms of CPU, battery power consumption, and network interface. There will not be a central cloud server acting as MCU. The disadvantage of using a full mesh is that this requires that the same data be repeatedly transmitted across the wireless link from/to the mobile phone. Thus, the tests includes metrics for CPU usage and battery power.

The mobile phone is used as a gateway as it is assumed to forward data received via its wireless link once, and send this data to another connected devices via WebRTC.

3.2 Related Work

Johnston, Yoakum, and Singh [102] discuss the enterprise usage of WebRTC in the context of security and compliance. P2P communication is a challenge to enterprise firewalls. Session Border Controllers (SBCs) [103] are widely used to traverse enterprise firewalls. SBC is a gateway which includes signaling and media communication. However, WebRTC is designed without a standardized signaling protocol and without sessions, thus it is not compatible with SBC. The authors point out some ways to assist WebRTC in enterprise firewall traversal,

such as detecting an ICE exchange in order to distinguish a WebRTC media flow or using a media relay to permit media flows through this relay. The conclusion is that today's WebRTC is difficult to use in enterprises, but there are a number of potential approaches to solve the existing issues.

Nurminen et al. [104] examine WebRTC for a video-on-demand (VoD) service by evaluating performance challenges and possible solutions. They propose an experimental system for a P2P VoD service, while noting that there can be limitations in current browser implementations and performance issues that may limit the functionality or lead to poor performance. The performance of the Data Channel API and video decoding in the browser are their main concern. However, the suggested system was not (yet) implemented and tested.

Singh, Lozano, and Ott [105] evaluate the performance of WebRTC video calls with varying amounts of network traffic over different topologies. Their experiments show that Chrome's congestion control algorithm works well in low delay networks, however the sending data rate under-utilizes the network when competing with TCP cross traffic.

3.3 Goal

Real-time communication (RTC) technology is already used by many applications, such as Google Hangouts [81]. However, these applications require plugins which must be installed. WebRTC eliminates these plugins which disrupt the user's experience. Moreover, WebRTC has standardized JavaScript APIs for developers who were previously limited to complicated plugins. WebRTC enables applications such as video chat, file sharing, and gaming within a browser by using the JavaScript APIs. In the tests described in this chapter, a mobile phone was used as a gateway to/from IoT devices. This chapter demonstrates:

- Usage of WebRTC to build a real-time application that can be used by a mobile phone (which acts as a gateway in the context of IoT).
- WebRTC performance in a mobile phone which is communicating with some number of peers.
- Limits of a mobile phone in terms of the maximum number of connected peers.
- Usage of a library, called PeerJS, to abstract the existing WebRTC API.

While the WebSocket tests described in the previous chapter focused on servers, the WebRTC tests described in this chapter focus on testing a mobile phone as a peer in a P2P architecture. The reason for testing a mobile phone is

to understand the limits of a mobile phone as a gateway. The final conclusion is based upon an analysis of data collected during the performance tests.

3.4 WebRTC API

The WebRTC API is a work in progress and is not yet standardized. It is a Javascript API which abstracts the complexities of P2P communication. The major components of the WebRTC API are `RTCPeerConnection`, `RTCSessionDescription`, `RTCIceCandidate`, `RTCDataChannel`, and `Media`. This section describes these components (except the Media component which is not relevant to this thesis).The tests described in this chapter focus on the use of the data channels to exchange arbitrary data, rather than video and audio.

Before going through the four components, it is good to know that both the Chrome and Firefox browsers implement this API each using their own prefixes, specifically Firefox uses `moz` and Chrome uses `webkit` or no prefix at all. For example,
`RTCPeerConnection` is called `mozRTCPeerConnection` in Firefox and `webkitRTCPeerConnection` in Chrome. The following descriptions will avoid using any prefixes for clarity and implementation independence.

3.4.1 RTCPeerConnection

RTCPeerConnection is the base of WebRTC. This object abstracts all the internal mechanisms of real-time data transfer. RTCPeerConnection:

- controls ICE states to traverse NAT,
- sends keepalive messages between peers,
- keeps track of local and remote streams, and
- provides methods for a developer to control the connection by sending an offer and an answer.

As shown in the code below, RTCPeerConnection accepts STUN and TURN server information, along with some options.

```
1 var server = {iceServers: [
2       {url: "stun:stun.l.google.com:19302"}, // Google's
          public STUN server
3       {url: "turn:numb.viagenie.ca", credential: "mypass",
          username: "gunayk@kth.se"} // Free TURN server
          operated by a firm called Viagenie
```

```
    ]
};
var options = { optional: [
      {RtpDataChannels: true} // This is a flag required
         for data channels for Chrome versions below 31.
    ]
};

var pc = new RTCPeerConnection(server, options);
```

3.4.2 RTCSessionDescription

WebRTC uses the Session Description Protocol (SDP) [106] for formatting signaling messages. These message contain network information collected by NAT traversal mechanisms, types of data to be transferred between peers, and CODECs to be used (in the case of media transmission). Once the RTCPeerConnection is created and a stream is added, such as audio or a data channel, then an offer can be sent to the other peer by calling the `createOffer()` method to create an SDP description. In `createOffer`'s callback, the SDP description is sent to the other peer over the signaling channel and saved as the peer's own local description. The peer receiving the offer creates its answer and sends this answer back while setting up its own local and remote endpoints according to the SDP descriptions. Figure 3.7 depicts this scenario. After a session description is set, the ICE workflow starts in the background to collect and exchange candidate IP address and port combinations. In the case of DataChannels, an SDP message is shown as below [107], in which the `m-line` indicates that the data channels will run over DTLS over SCTP.

```
  ...
  ...
   m=application 54111 DTLS/SCTP 5000
   c=IN IP4 79.97.215.79
   a=sctpmap:5000 webrtc-datachannel 16
  ...
  ...
```

The `createOffer()` method of RTCPeerConnection can be used as follows:

```
pc.createOffer(function (desc) {
   pc.setLocalDescription(desc);
   // Send the offer and wait for answer.
```

```
    signalingChannel.send(JSON.stringify({"offer": desc}));
});
```

When the offer is received, the peer saves the offer and sends its own description as follows:

```
signaling.onmessage = function(msg) {
   if (msg.offer) {
      pc.setRemoteDescription(JSON.parse(msg.offer));
      pc.createAnswer(function (answer) {
         pc.setLocalDescription(answer);
         signalingChannel.send(JSON.stringify({"answer":
            answer}));
      });
   } else if (msg.candidate) {
      pc.addIceCandidate(msg.candidate);
   }
}
```

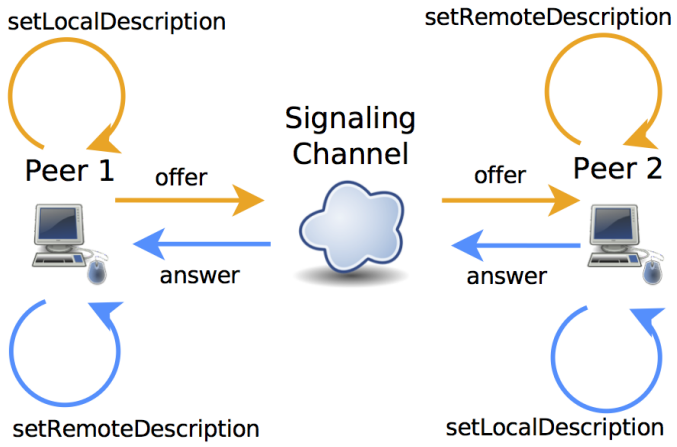


Figure 3.7: SDP flow in WebRTC

3.4.3 RTCIceCandidate

As introduced in Chapter 1, the ICE framework is a suite composed of STUN and TURN. The STUN and TURN servers help the peer to learn its own candidate IP addresses. These addresses can potentially be used by another peer to connect to it. When a candidate IP address is found, the peer sends this address to the other

peer over the signaling channel in SDP messages. An example of sending such a candidate address is shown below.

```
pc.onicecandidate = function (evt) {
   // Candidate exists in evt.candidate
   if (evt.candidate) {
      // Send it to the other peer over signaling channel.
      signalingSend(JSON.stringify({"candidate":
         evt.candidate}));
   }
};
```

3.4.4 RTCDataChannel

The RTCDataChannel can be used to exchange binary or textual data once a RTCPeerConnection is established. The RTCDataChannel API is similar to WebSocket API with its own events and callbacks. However, there are many important differences due to the fact that WebSocket runs over TCP, while RTCDataChannel runs over SCTP over DTLS over UDP. Table 3.1 presents some differences between WebSocket and RTCDataChannel APIs.

Table 3.1: Comparison of WebSocket and RTCDataChannel

	RTCDataChannel	**WebSocket**
Reliability	Reliable or Unreliable	Reliable
Delivery	Ordered or Unordered	Ordered
Encryption	Encrypted	Encrypted or Unencrypted

DataChannel supports in order or out of order, and reliable or unreliable message delivery. Unreliable delivery has two options: retransmit and timeout. The retransmit option restricts the number of retransmissions, while the timeout option stops the retransmission after a configured time. While ordered and reliable delivery is similar to TCP, unordered and unreliable delivery with zero retransmission behaves similar to UDP.

The constructor accepts a channel name and options. When data is received, the `onmessage` callback is called. The code below shows an example of how to create a data channel.

```
var name = "channel";
var options = {reliable: false}; // The W3C Draft of
   WebRTC specifies additional options.
var channel = pc.createDataChannel(name, options);
```

```
pc.ondatachannel = function (evt) {
   evt.channel.onmessage = function () {
     console.log(evt.data);
   };
};
channel.onmessage = function (evt) {
   console.log(evt.data);
   channel.send("Received your message!");
}
channel.onerror = function (err) {
   console.error(err);
};
channel.onclose = function (evt) {
}
```

The code pieces are combined to create an application. A complete code example of using the WebRTC DataChannel API can be found in Appendix C.

3.4.5 PeerJS

The WebRTC API does not specify how to exchange SDP and ICE candidates over a signaling channel. Since signaling is not part of the API, this means that developers can chose any suitable signaling mechanism. The PeerJS library fills this gap and provides a easy to use API *including* the signaling channel [108]. PeerJS deals with the WebRTC handshake and allows connections to be established to a peer based upon an identifier (these identifiers can be any string). Identifiers must be exchanged between peers, but how this is done is left to the developer. In the tests used in this thesis project the identifier of the mobile phone is hardcoded and known in advance by all the other peers.

PeerJS has two components: the client side library (which needs to be included in a web page) and the server side library which handles the signaling.

3.4.5.1 PeerJS Client

After adding the PeerJS client library to the page, a peer object can be created and used as shown in the code below. The PeerJS API is simple and further details of this API can be found in [109].

```
<html>
<head>
<script src="http://cdn.peerjs.com/0/peer.min.js"></script>
<script>
```

```
5       // Create a peer with an id. Give the address and
           port of PeerServer
6       var peer = new Peer('some-id', {host: 'localhost',
           port: 'some-port'});
7
8       // Listen for incoming connections
9       peer.on('connection', function(conn){
10        // Listen for incoming data
11        conn.on('data', function(data) {
12          console.log('Got data:', data);
13        });
14      });
15
16      // Connect to a peer.
17      var conn = peer.connect('other-id');
18      // When the connection is open, send a message.
19      conn.on('open', function() {
20        conn.send('Hello world!');
21      });
22  </script>
23  </head>
24  <body></body>
25  </html>
```

3.4.5.2 PeerServer

PeerServer provides an abstraction of the process of SDP and ICE candidate exchange. A PeerServer helps to establish a connection between peers, and afterwards the data flows directly peer to peer.

The PeerServer builds on Node.js and can installed by the command:

```
1  $ npm install peer
```

Once the PeerServer module is installed, it is simple to create a PeerServer as shown below:

```
1  var PeerServer = require('peer').PeerServer;
2  var server = new PeerServer({ port: 'some-port' });
```

PeerJS was used in the tests because it wraps all of the WebRTC API into a nice and simple API and also proves a signaling server.

3.5 Experimental Setup

To test the WebRTC P2P architecture, an EC2 instance was used as a signaling server to broker the connections. A mobile phone was used as a gateway between two laptop computers. One of the laptop computers was used to produce and send data through the mobile phone to the other laptop computer. The specifications of all of these devices can be found in Table 3.2.

Table 3.2: Hardware Specification of WebRTC tests

Signaling Server	Amazon Micro Instance
Mobile phone	Android Nexus 4 Operating System: Android 4.3, Jelly Bean CPU: Qualcomm Snapdragon S4 Pro CPU Memory: 2 GB RAM Battery: 2100 mAh 3.8V battery which is 8 Wh Browser: Firefox for Android version 26 Internet: Connected to 3G network of Tele2's 3G network. Download data rate: 10 Mbps. Upload data rate: 1.10 Mbps.
Laptop Computers	MacBook Pro Operating System: OSX 10.8.3 CPU: 2.2 GHz Intel Core i7 Memory: 4 GB DDR3 Browser: Firefox version 26 Internet: Connected to Wi-Fi network of eduroam at KTH. Download data rate: 20 Mbps. Upload data rate: 20 Mbps.

Download and upload data rates are determined by Ookla's Speedtest [110].

In addition, as WebRTC uses the ICE framework, both STUN and TURN servers are needed. A TURN server is needed when two users are behind symmetric NATs. In the case of two symmetric NATs, NAT traversal using STUN is impossible, hence a TURN server is used to relay the data. The test configuration that was used did not require a TURN server, thus only a STUN server was needed. Fortunately, a number of STUN servers are hosted by Google. A list of these STUN servers can be found in Appendix D. For all of the tests a single STUN server was used, specifically: `stun.l.google.com:19302`.

The EC2 instance runs a PeerServer in a Node.JS program and it serves webpages containing test scripts. To run the tests, the mobile phone and the laptop computers access the webpages. The scripts will be described in detail in next section.

The WebRTC Data Channel is supported in Chrome 26+ and Firefox 22+.

Unfortunately, Chrome and Firefox DataChannels are not able to connect with each other. In the tests, Firefox running on the laptop computer and Firefox Android running on an Android mobile phone were used. Firefox DataChannels seem to be more stable than Chrome's implementation at the time of writing this thesis, hence this thesis used the Firefox DataChannels. Chrome has incompatibility between their releases of Android and desktop since their first DataChannel implementation used RTP instead of SCTP. Firefox implemented SCTP from the beginning. WebRTC is evolving rapidly and it is probable that Firefox, Firefox Android, Chrome, and Chrome Android will be compatible by the time that this thesis project is completed.

In the mobile phone an application called Trepn Profiler [111] was used. Trepn profiles the performance of Android applications and provides CPU usage and battery power statistics.

As will be explained in the next section, the mobile phone is used for "tethering" in order to share an Internet connection with the phone acting as a gateway. The Anroid-wifi-tether application [112] was used in order to enable *ad-hoc* mode usage of the Wi-Fi (i.e., IEEE 802.11) wireless local area network interface of the mobile phone. Android-wifi-tether exploits the 'iwconfig' command for configuring the wireless local area network interface. In order to use android-wifi-tether, the phone should be rooted. Rooting implies modifying the constrained access rights of the operating system in order to run applications which make use of some of the features of the device. More information about the rooting procedure of an Android phone can be found in the related sections of the XDA Forums [113]. Android-wifi-tether enabled the mobile phone to be used as a gateway in these tests.

In order to collect network packets, Wireshark [114] was running on the laptop computers and Shark For Root [115] was running on the mobile phone. During the tests, these programs capture the network traffic between the two laptop computers passing through the Android phone. The collected data was analyzed in terms of one-way delay and throughput.

To synchronize the clocks of the computers and the mobile phone the Network Time Protocol (NTP) [42] was used. The NTP daemon synchronizes the local system's time with a remote server (ntp3.sptime.se). In the mobile phone an application called ClockSync [116] was used to implement NTP.

3.6 Data Collection

Because this thesis aims to use a mobile phone as a gateway, several possible ways of implementing this gateway were considered. Some of these approaches to convert a mobile phone into a gateway are:

- Configure the Linux kernel to set up a bridge (as link layer forwarding),
- Use IP forwarding (as network layer forwarding),
- Implement a native application with networking capabilities (to perform network or application layer forwarding),
- Implement a WebRTC application in an Internet browser (to perform application layer forwarding).

In an Android phone the minimum cost of building a gateway would be provided by using the Linux kernel. A mobile phone can forward packets from one network interface to another if the mobile phone's kernel sets up a bridge between network interfaces. The `brctl` command can be used to set up a bridge configuration in the Linux kernel. The `brctl` command configures the kernel to forward packets based on an Ethernet address (i.e., an address at the link layer - layer 2). All protocols can be forwarded transparently via this bridge. However, bridging in Linux Kernel of Android is not enabled by default. To be able to set up a kernel bridge, the kernel should be compiled with support for bridging [117].

This thesis takes the approaches of using IP forwarding and implements an application in an Internet browser to perform this forwarding. IP forwarding is network layer forwarding and provides a base to which we can compare application layer forwarding. Application layer forwarding in an Internet browser was used to exploit the WebRTC JavaScript API. The application takes data from a DataChannel and forwards this data to another DataChannel.

Metrics were chosen to evaluate the performance of WebRTC and limits of the mobile phone. These metrics are: one-way delay, throughput, CPU usage, and battery power consumption of the mobile phone.

To collect data about these metrics two sets of tests were done. The first set of tests focused on forwarding data over the network layer and over the application layer. These tests captured network packets to give an analysis of one-way delay and throughput. One-way delay is important as it indicates the overall delay along the message delivery path. Throughput is another important metric as it characterizes whether the mobile phone could forward all the received data. In these tests, one of the laptop computers was connected to the mobile phone via the Wi-Fi interface (by using android-wifi-tether) and produced data to send the other computer. The mobile phone forwarded this data over its wide area network (WAN) interface with IP forwarding or via the application in the Internet browser, specifically Firefox.

The second set of tests focused on the mobile phone's processing ability to do multiplexing and demultiplexing of DataChannels. These tests used a script running in Firefox to collect data of processing delay, CPU usage, and available

battery power of the mobile phone. Since, the mobile phone was supposed to serve a number of IoT devices, it should be able to multiplex or demultiplex coming packets. In the multiplexing test, to reduce the number of messages going out the other interface, the mobile phone combines multiple messages that it has received. In the demultiplexing test, a multiplexed message is split into separate messages to go out the other interface. These tests characterize the mobile phone's limitations in doing the relevant processing that it would need to do when acting as an IoT gateway. In this set of tests, one of the laptop computers sent messages to the mobile phone over the Internet via the mobile phone's WAN interface and the cellular network operator's connection to the Internet. The mobile phone forwarded the messages to the other laptop computer after doing multiplexing or demultiplexing. All of the computers were connected to the same local network (in this case eduroam at KTH).

The number of DataChannels were increased by 5 every 5 minutes. It was observed that the limit of concurrent DataChannel connections is 50 when using Firefox. When using this browser no further peers could not successfully obtain a connection. In these tests, each DataChannel transfers a string of 100 characters with a message frequency of 4 Hz.

During these tests, the mobile phone was not plugged into a power source since connecting a power source to the mobile phone leads to inaccurate data about battery power consumption and performance. Moreover, the screen of the mobile phone was in the dimmed mode.

3.7 Data Analysis

Both sets of tests are analyzed in separate subsections below and an overall conclusion is given in Section 3.8.

3.7.1 Forwarding DataChannels over a Mobile Phone

As explained in the previous section, these tests compare network layer forwarding and application layer forwarding in terms of one-way delay and throughput. These metrics are given in separate subsections.

3.7.1.1 One-way delay

Figure 3.8 presents the one-way delay measured in the network layer forwarding tests. Every 5 minutes (i.e., 300 seconds) the number of peers were increased by 5. It can be seen that the one-way delay increases linearly as the number of peers are increased in each 300 seconds interval of the graph, and the delay varies between

100 ms and 400 ms on average. In Figure 3.9 the one-way delay of application layer forwarding is presented. Note that the x-axis of this figure ends at 1500 seconds (i.e. 25 minutes) as half of the 50 DataChannels were used between the mobile phone and the receiver computer in order to forward incoming data from the sender computer. While the vertical axis of Figure 3.8 ends at 500 ms, Figure 3.8 ends at 4 seconds since there are some large outliers. Each peer created by the sender computer leads to 2 DataChannels in the mobile phone. While the delay was always under 500 ms for network layer forwarding in Figure 3.8, Figure 3.9 has some large outliers at multiples of 300 seconds. At these points, new DataChannels are opened in order to forward the increased number of peers. While doing network layer forwarding, opening additional connections did not affect the one-way delay as occurred in the case of application layer forwarding.

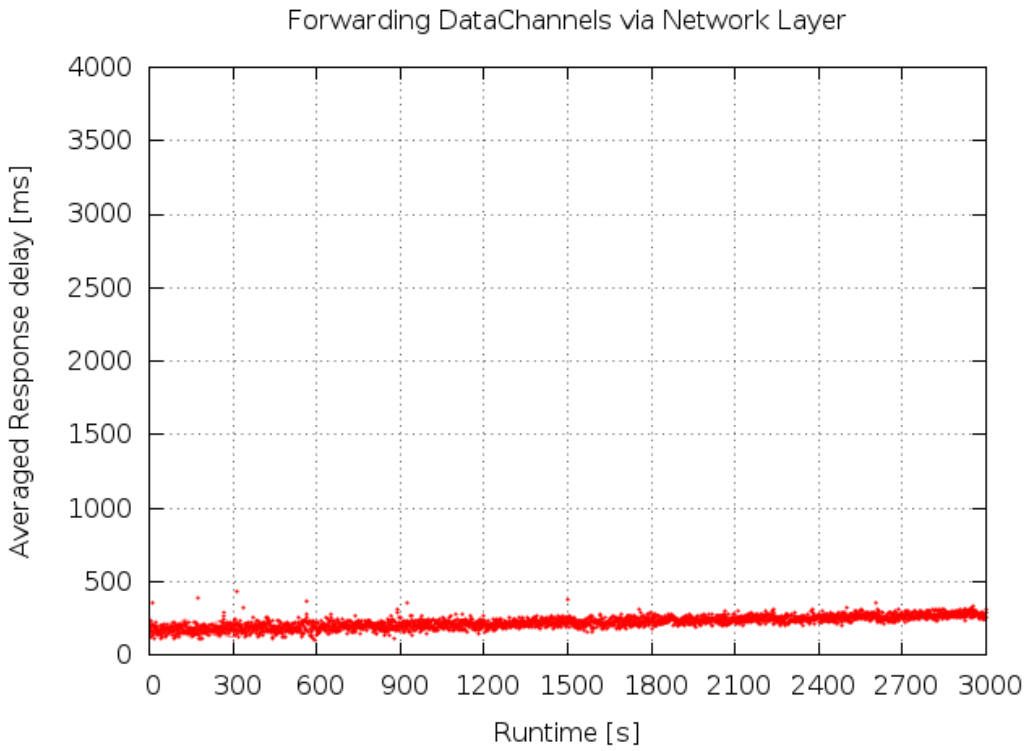


Figure 3.8: One-way delay of network layer

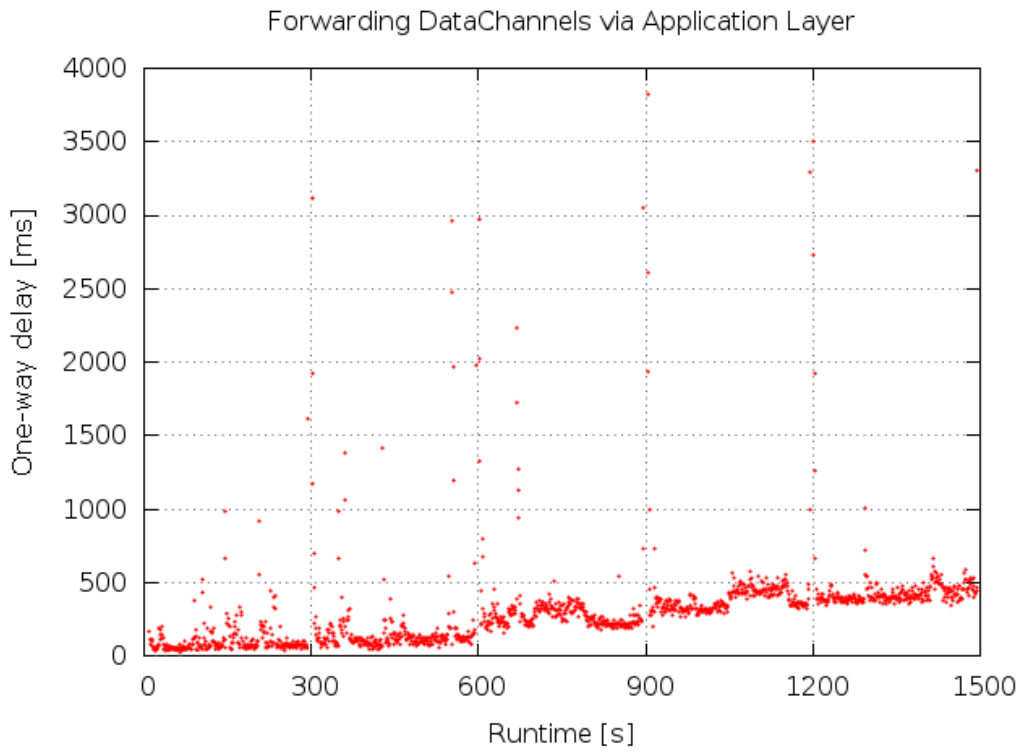


Figure 3.9: One-way delay of application layer

Table 3.3 represents the statistics of the network layer results. The mean delay increases linearly at each step of increased numbers of peers. While there are 5 peers the average one-way delay is 186 ms. With each step the average delay is increasing by 10-15 ms and it reaches to 277 ms for 50 peers. Since there are not outliers, the median is close to the mean delay. It could be expected that the standard deviation would also increase; however, it is observed that the deviation decreased with an increased number of peers.

The statistics of application layer forwarding are given in Table 3.4. The outliers at multiples of 300 seconds were omitted from these statistics to give a more clear comparison of the delay in sending messages rather than the delay in opening connections. For 5 peers, the average one-way delay is around 187 ms, and it reaches 431 ms with 25 peers. The increase at each step after 10 peers is around 100 ms, except for the last step where the increase is 30 ms as ve go from 20 peers to 25 peers. This will be explained in the throughput section by showing that the mobile phone was not able to forward all of the incoming data with 25 peers.

Table 3.3: One-way delay statistics of network layer

Peers	Average delay (ms)	Median (ms)
5	185.77 ± 34.71	173.25
10	198.90 ± 35.51	187.80
15	201.84 ± 32.35	200.13
20	209.60 ± 26.20	209.54
25	218.95 ± 22.90	216.76
30	226.21 ± 24.03	225.42
35	238.80 ± 19.66	238.46
40	249.35 ± 19.14	249.06
45	261.56 ± 20.86	259.99
50	276.69 ± 18.35	275.96

Table 3.4: One-way delay statistics of application layer

Peers	Average delay (ms)	Median (ms)
5	187.02 ± 174.78	56.25
10	196.06 ± 314.79	121.25
15	296.90 ± 179.43	266.44
20	400.25 ± 188.11	373.00
25	431.03 ± 168.81	409.11

As shown in these statistics, as the number of peers increases, the network layer forwarding performs better than application layer forwarding. The results are similar for 5 and 10 peers. However, the average one-way delay of application layer forwarding increases faster than the network layer forwarding. For example, the average one-way delay with 20 peers is 400 ms with application layer forwarding while network layer forwarding has a delay of only 277 ms of one-way delay for 50 peers. The standard deviation also increases faster in the case of application layer forwarding. Moreover, the number of concurrent peers that can be supported with application layer forwarding was half the number as for network layer forwarding since one peer occupies two DataChannels in application layer forwarding.

3.7.1.2 Throughput

In this section the throughput of the mobile phone was analyzed in order to check whether the mobile phone was able to forward all of the incoming data or not. It is expected that the data received via one interface (which is connected to the

sending computer) should be similar to the data going out of the other interface (which is connected to the receiving computer).

Figure 3.10 presents the received throughput in the case of network layer forwarding. The received data from the sending computer was the same as the data forwarded to the receiving computer for each step in the number of peers. This implies that the mobile phone was able to do network layer forwarding for 50 concurrent peers. The statistics are given in Table 3.5. When the captured packet logs were investigated, it was seen that the packet frames on WAN interface of the mobile phone were 2 bytes larger than the packet frames of the Wi-Fi interface. It turns out that packet capturing uses a pseudo-link-layer to capture packets on some devices where the native link layer header is not available or can not be used. This is called Linux cooked-mode and it adds 2 bytes to the captured frames [118]. These 2 bytes were omitted when the results were compared for the receiving and sending interface. Note that a packet of message data is around 245 bytes in total IP packet length after the protocol overheads.

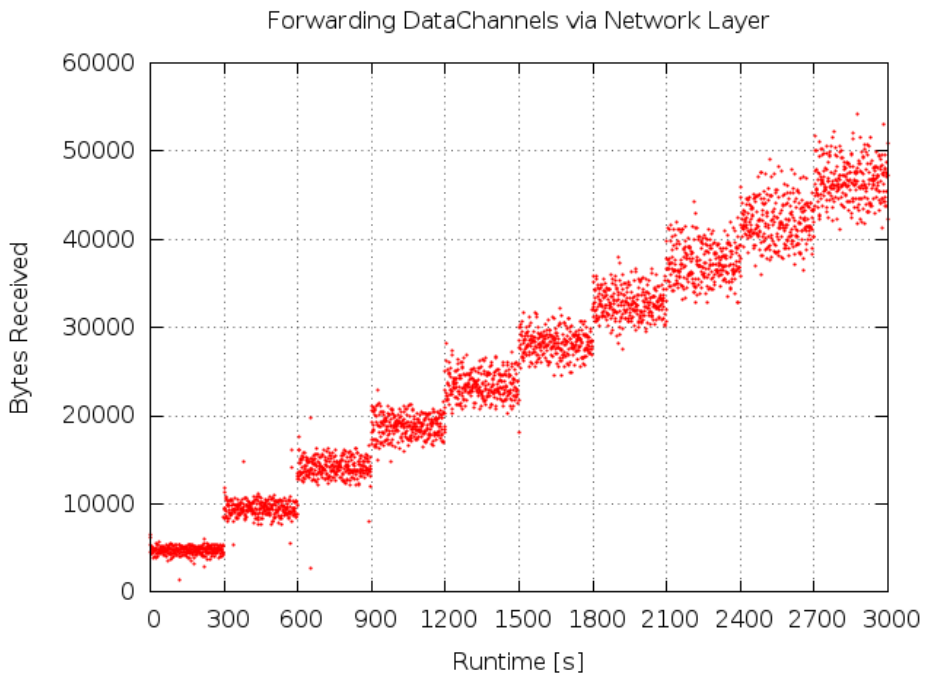


Figure 3.10: Received throughput of network layer. Sent throughput was the same.

Table 3.5: Througput statistics of network layer

Peers	Average received (bytes)
5	4954.21 ± 540.76
10	9864.98 ± 1116.25
15	14198.87 ± 1400.96
20	18848.31 ± 1367.86
25	23513.29 ± 1575.48
30	28253.16 ± 1518.30
35	32815.08 ± 1752.26
40	37373.82 ± 2147.85
45	42046.02 ± 2460.80
50	46550.84 ± 2432.92

Figure 3.11 and 3.11 represent throughput of application layer forwarding. In the last test case with 25 peers, it is seen that the the average throughput is much less than expected and the amount of data forwarded by the mobile phone is not similar to the received data. This shows that the mobile phone could not handle the test case of 25 peers. The statistics of these results are given in Table 3.6. The standard deviation on the sending interface is greater than the receiving interface. This was not observed in the network layer tests, which means that the application layer forwarding increases the throughput deviation of the forwarded throughput.

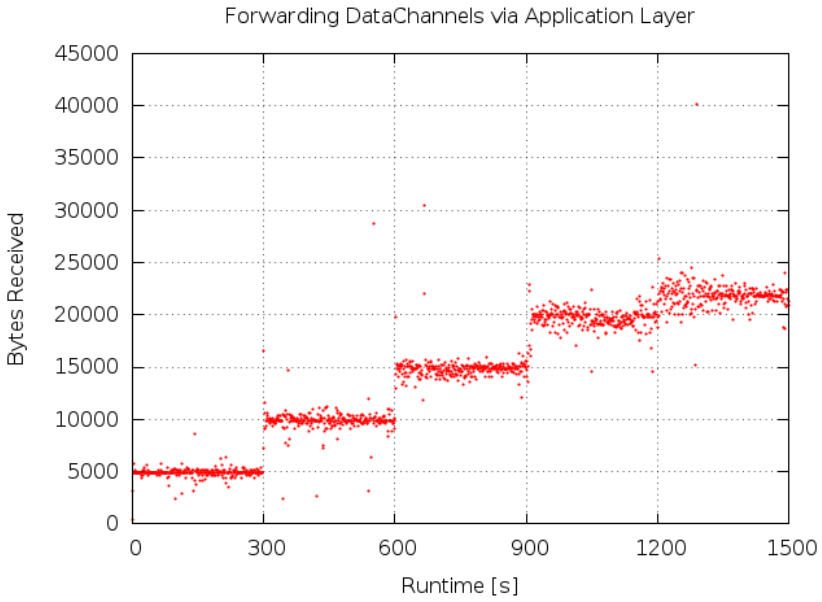


Figure 3.11: Received throughput by the mobile phone

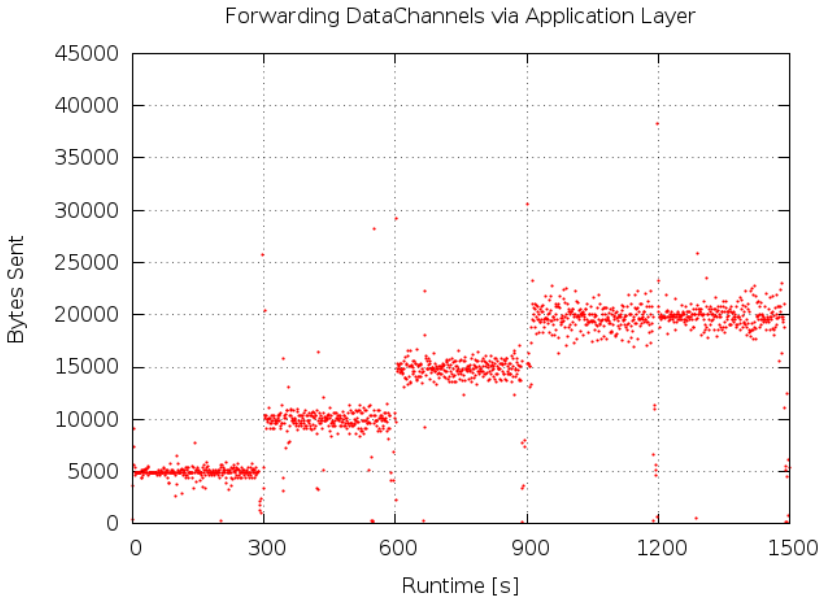


Figure 3.12: Sent throughput by the mobile phone

Table 3.6: Througput statistics of application layer

Peers	Average received (bytes)	Average sent (bytes)
5	4992.86 ± 930.77	4982.65 ± 1763.36
10	9992.74 ± 1632.14	9982.98 ± 3390.04
15	14908.06 ± 1305.86	14864.23 ± 4048.26
20	19693.64 ± 1392.69	19559.88 ± 3919.07
25	21827.21 ± 1415.41	19586.51 ± 5108.58

3.7.2 Multiplexing and Demultiplexing DataChannels

Since the mobile phone can connect to a number of IoT devices, it will do either multiplexing to reduce number of messages going out the other interface or demultiplexing to split a multiplexed message into pieces to each go to a different IoT device. This section characterizes the mobile phone's ability (in terms of CPU and battery power consumption) to do multiplexing and demultiplexing.

3.7.2.1 Multiplexing

In the multiplexing test, the incoming messages to the mobile phone were buffered for a message period (250 ms) and aggregated into one message. Then, this

aggregated message was forwarded into another DataChannel. The processing delay of aggregation, CPU usage, and battery power consumption were measured and are presented in separate subsections. The number of peers were increased by 5 every 5 minutes. The last case has only 49 peers because one of the peer connections was used for forwarding the multiplexed data to the other laptop computer.

3.7.2.1.1 Processing delay

Figure 3.13 presents the processing delay measured in the multiplexing test. Note that the x-axis of this figure shows the time since the start of the test. As stated before, each test case with a number of peers ran for 5 minutes (i.e., 300 seconds). As seen in the figure there is a general increase in processing delay deviation as the number of peers is increased.

The statistics of the results are presented in Table 3.7. The average delay of multiplexing usually increases as the number of peers increases. The average delay is around 2 ms for 5-10 peers, around 3 ms for 15-25 peers, and around 4 ms for 30-40 peers. At 45 peers it is 4.66 ms and at 49 peers it reaches 7.76 ms showing a larger increase than would be expected given the other measurements. The median value and deviation of multiplexing delay also increase as the number of peers increases over these test cases. Figure 3.14 shows the curve fitted to median of this data by using linear regression. The R^2 value indicates the goodness of fit for the line.

Table 3.7: Processing delay statistics

Peers	Average delay (ms)	Median (ms)
5	1.55 ± 1.41	1.28
10	2.27 ± 2.96	2.20
15	2.91 ± 3.57	2.32
20	3.09 ± 3.18	2.38
25	2.96 ± 3.01	2.38
30	4.25 ± 4.02	2.51
35	4.19 ± 4.47	2.56
40	4.21 ± 4.11	3.39
45	4.66 ± 4.65	4.09
49	7.76 ± 5.71	5.92

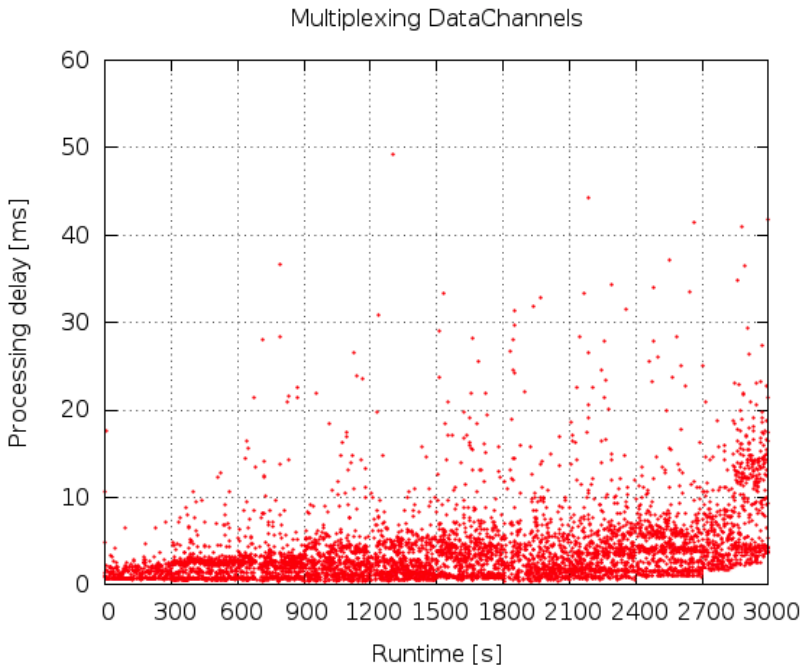


Figure 3.13: Processing delay of multiplexing DataChannels

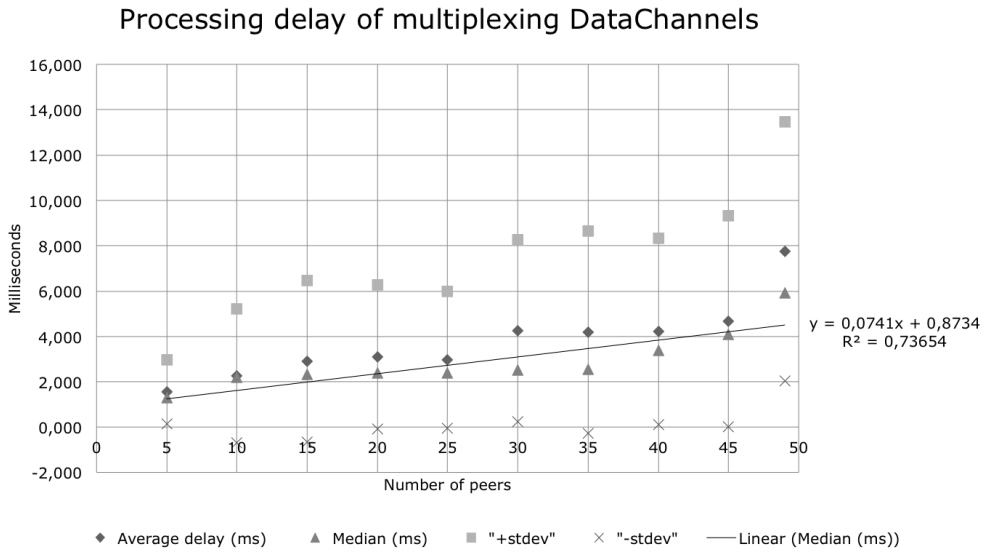


Figure 3.14: Processing delay of multiplexing DataChannels

3.7.2.1.2 CPU usage

The Trepn application monitored the CPU usage during the tests. The collected data is depicted in Figure 3.8. Similar to processing delay of multiplexing, CPU usage increases as the number of peers increases. In this figure, it can be seen that there are some outliers at multiples of 300 seconds. These peaks correspond to the time when the new peers are connected. When new peers are connected, the CPU usage reaches 40%. While multiplexing incoming messages, the CPU usage is between 10% and 17%. As in the WebSocket tests, it is seen that opening connections consumes more CPU processing time than sending messages.

Statistics of the CPU usage are given in Table 3.8. For 5-35 peers, the CPU usage varies between 9% and 10%. The CPU usage is approximately 12% for 40 peers, 13% for 45 peers, and around 17% for 49 peers. It can be seen that there is an increase of 1% from 40 to 45 peers, while there is a larger increase of nearly 4% for 49 peers. The median value of CPU usage shows a similar pattern of increases.

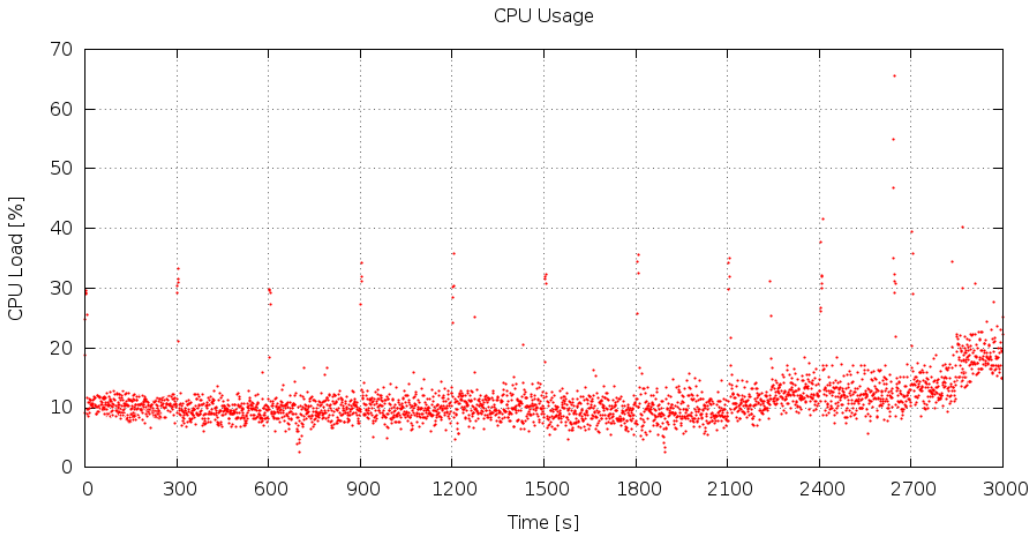


Figure 3.15: CPU usage of multiplexing DataChannels

Table 3.8: CPU load statistics of multiplexing

Peers	Average (%)	Median (%)
5	9.54 ± 1.30	9.11
10	9.51 ± 1.61	9.44
15	9.56 ± 1.78	9.50
20	9.94 ± 1.71	9.49
25	10.20 ± 2.07	9.51
30	10.12 ± 1.81	9.82
35	10.43 ± 2.14	9.73
40	11.95 ± 2.49	11.67
45	13.02 ± 5.85	12.11
49	16.68 ± 4.12	16.56

3.7.2.1.3 Battery Power Consumption

Trepn measures battery power consumption in mW. Before the start of the tests, Trepn collected baseline data as an estimate of the power drain of the Android OS and the Trepn application. Trepn measured the baseline as 1195 mW before the start of the multiplexing test.

Figure 3.16 shows the sampled data of battery power consumption during the multiplexing test. As expected, the graph of power consumption is similar to the graph of CPU usage shown in the previous section, since CPU usage is related to power consumption. As seen in the CPU graph before, this graph also has some outliers at multiples of 300 seconds when the new peers are connected. While opening new connections, the power consumption can reach 2500 mW. While multiplexing messages, the average power consumption is between 1325 mW and 1695mW. Note that the battery power consumption is related to the CPU usage, the usage of network interfaces, and also the brightness of the screen.

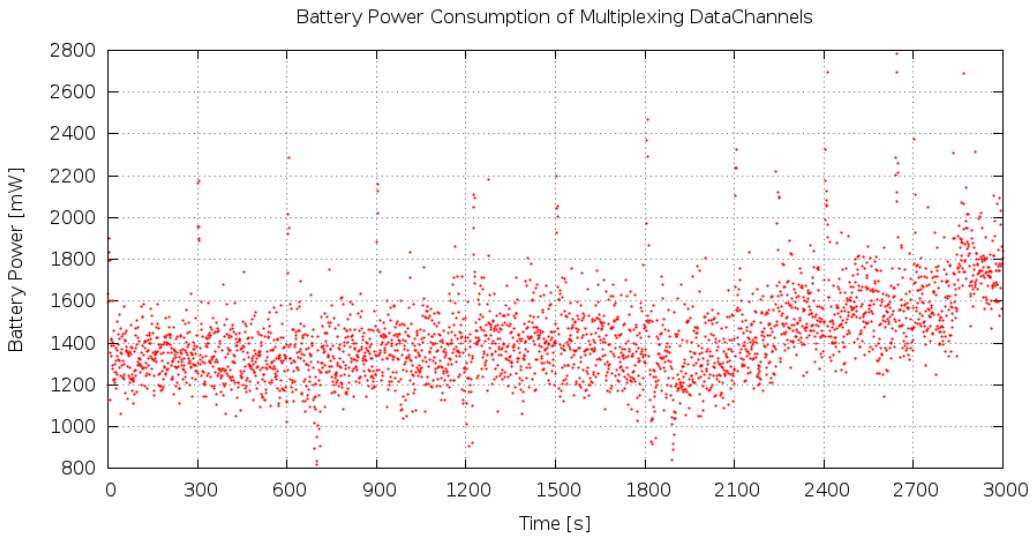


Figure 3.16: Battery power consumption of multiplexing DataChannels

Statistics of the power consumption data are given in Table 3.9. The baseline was measured as 1195 mW for the idle state of phone before the tests start. For 5 every peers, there is a difference of 130 mW between the measured power consumption and the baseline. As the number of peers increases, this difference increases and reaches 500 mW for 50 peers consuming a total of 1695 mW. The mobile phone's battery has 8 Wh of energy when it is fully charged. For example, 10 peers consume 1.3 W on average, hence with ten peers the battery can offer 6 hours of service. For 50 peers, the battery's life time is 5 hours.

Table 3.9: Battery power consumption statistics of multiplexing

Peers	**Average (mW)**	**Median (mW)**
5	1325.73 ± 117.34	1310.71
10	1320.72 ± 117.37	1313.70
15	1332.41 ± 139.60	1324.88
20	1368.26 ± 138.62	1364.19
25	1398.09 ± 147.85	1406.71
30	1385.87 ± 145.86	1391.41
35	1393.08 ± 165.78	1396.83
40	1490.85 ± 164.62	1477.90
45	1569.78 ± 209.51	1547.89
49	1695.09 ± 180.96	1694.87

3.7.2.2 Demultiplexing

In the demultiplexing test, each incoming message to the mobile phone was an aggregation of 10 individual messages. These aggregated messages were demultiplexed and each demultiplexed message was forwarded into a DataChannel. The number of peers were increased by 5 every 5 minutes as done in the multiplexing test. The test ends with 40 peers because 10 of 50 DataChannels were used to forward demultiplexed data. The processing delay of demultiplexing, CPU usage, and battery power consumption were measured and are presented below in separate subsections.

3.7.2.2.1 Processing delay

The processing delay of demultiplexing messages is presented in Figure 3.17. Note that the x-axis of this figure ends at 2400 seconds because each test case increases the number of peers by 5 and the tests ends with 40 peers (as the other 10 DataChannels are in use for the demultiplexed data). The figure shows that there are greater deviation as the number of peers increases.

The statistics of these results are presented in Table 3.10. The average delay of demultiplexing usually increases while the number of peers increases. The average delay is between 2 ms and 3 ms. The median value is always under 2 ms. In the multiplexing test, the delay reaches 4 ms at 40 peers while in this demultiplexing test the delay is 2.98 ms with 40 peers. Figure 3.18 shows the statistics in a graph with a curve fitted to median values.

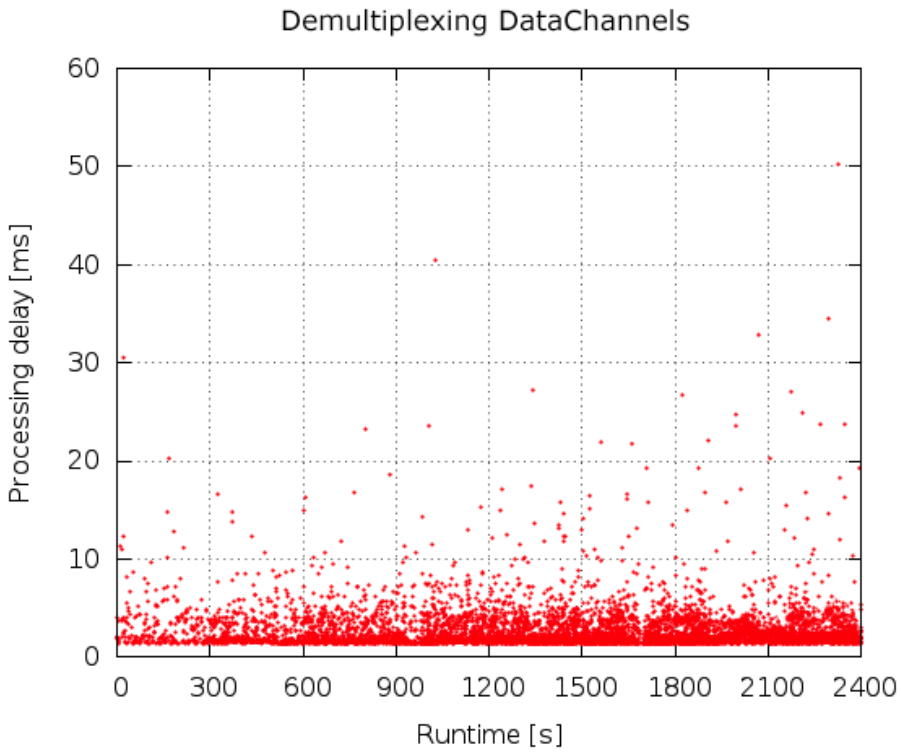


Figure 3.17: Processing delay of demultiplexing DataChannels

Table 3.10: Processing delay statistics

Peers	**Average delay (ms)**	**Median (ms)**
5	2.42 ± 1.07	1.72
10	2.63 ± 1.89	1.82
15	2.68 ± 1.99	1.89
20	2.71 ± 2.22	1.89
25	2.69 ± 2.21	1.91
30	2.72 ± 2.30	1.95
35	2.83 ± 2.37	1.93
40	2.98 ± 2.58	1.97

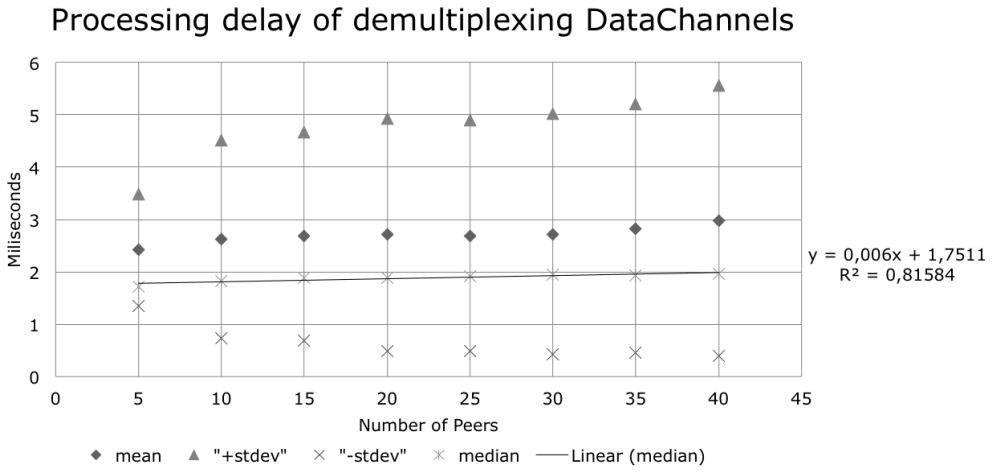


Figure 3.18: Processing delay of demultiplexing DataChannels

3.7.2.2.2 CPU usage

The collected data of the CPU usage is depicted in Figure 3.11. In this figure, it can be seen that there are some outliers at multiples of 300 seconds as seen in the multiplexing test. These peaks correspond to the time when the new peers are connected. Demultiplexing incoming messages uses between 10% and 35% of the CPU.

Statistics of the CPU usage are given in Table 3.11. The mean CPU usage increases as the number of peers increases. Until 30 peers, the increase of the CPU usage in each step of 5 additional peers varies between 2% and 3%. From 30 to 35 peers there is a difference of around 6% and from 35 to 40 peers this increase is around 5%. The CPU usage reaches approximately 35% for 40 peers while in the multiplexing test it reached 12%. Note that demultiplexing tests used 10 DataChannels to forward the demultiplexed messages while multiplexing tests used only one DataChannel.

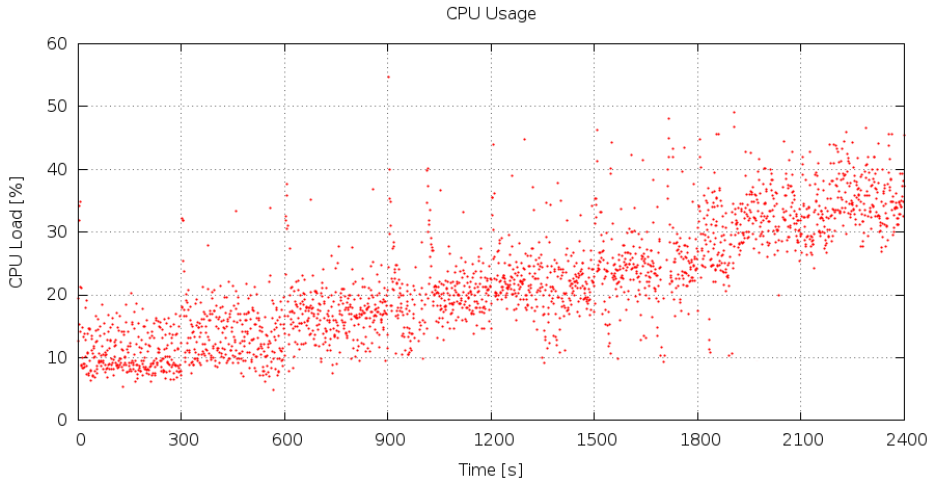


Figure 3.19: CPU usage of demultiplexing DataChannels

Table 3.11: CPU load statistics of demultiplexing

Peers	Average (%)	Median (%)
5	11.16 ± 3.90	9.95
10	13.66 ± 4.20	13.22
15	16.98 ± 4.30	17.00
20	19.02 ± 5.17	19.17
25	21.86 ± 4.50	21.56
30	23.87 ± 7.12	24.05
35	29.95 ± 7.58	31.11
40	34.72 ± 7.37	34.05

3.7.2.2.3 Battery Power

Figure 3.20 shows sampled data of battery power consumption during the demultiplexing test. As expected, the graph of power consumption is similar to the graph of CPU usage shown in the previous section. During the demultiplexing tests the average power consumption varies between 1388 mW and 2323 mW.

Table 3.12 shows the statistics of the power consumption data. The baseline was measured as 1058 mW for the idle state of phone before the tests started. For 5 peers, there is a difference of 330 mW between the measured power consumption and the baseline while this number was 130 mW for the multiplexing test. For 40 peers this difference increases to 1264 mW and the power consumption is 2322.57 mW.

For 10 peers it is seen that the average power consumption is around 1.5 W. Thus, the battery life time would be around 5 hours because of the mobile phone's battery 8 Wh. For 40 peers, the battery can offer around 3.5 hours of demultiplexing service.

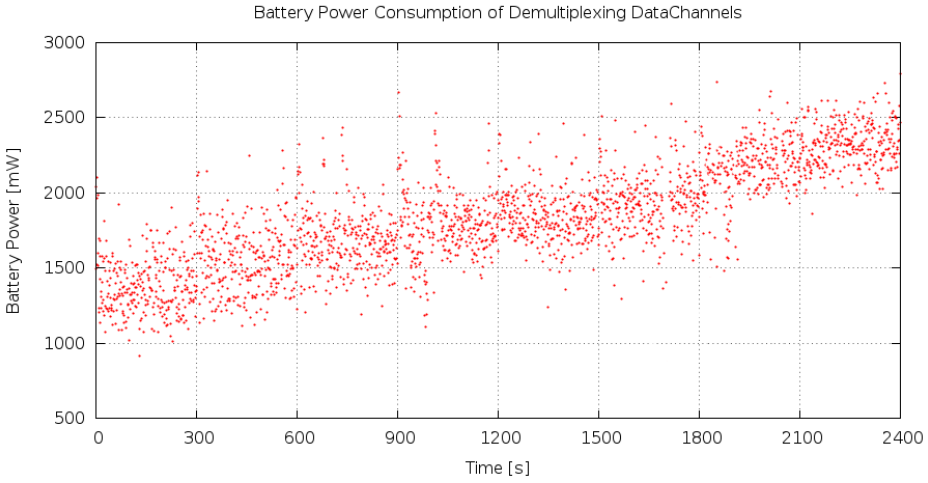


Figure 3.20: Battery power consumption of demultiplexing DataChannels

Table 3.12: Battery power consumption statistics of demultiplexing

Peers	**Average (mW)**	**Median (mW)**
5	1388.17 ± 192.06	1358.78
10	1555.48 ± 217.92	1549.83
15	1684.90 ± 213.40	1676.93
20	1789.94 ± 261.15	1779.88
25	1850.88 ± 251.50	1833.69
30	1888.66 ± 255.46	1925.67
35	1945.96 ± 271.92	2115.80
40	2322.57 ± 260.12	2319.42

3.8 Conclusion

This chapter provided performance tests of WebRTC protocol over a mobile phone. The first set of tests provided a comparison of network layer forwarding and application layer forwarding. In network layer forwarding tests the packets are forwarded directly based upon on IP addresses. In application layer forwarding tests the messages are forwarded from one DataChannel to another one by a script running in the Firefox browser. As expected network layer forwarding performs better because the packets are forwarded by the operating system rather than an application. However, these tests were needed to provide a comparison of application layer forwarding. The performance of application layer forwarding was important to measure since Earl does not implement an IP stack. The results showed that using application layer forwarding the mobile phone is able to forward data from 20 peers simultaneously with a one-way delay of 400 ms on average. As stated in Chapter 1, the requirements of the arm-probe were that data should be transferred with less than half second of delay and at least 10 peers should be supported. This in terms of these requirements, this solution is applicable.

The second set of tests measured the performance of the mobile phone while multiplexing and demultiplexing DataChannel messages. Since the mobile phone can connect to a number of IoT devices, it needs to do either multiplexing or demultiplexing. For example, for the requirement of the arm-probe of 10 peers, the mobile phones would consume 1.3 W in average while doing multiplexing and it consumes 1.5 W on average while doing demultiplexing. Thus, the battery can offer 6 hours of multiplexing and 5 hours of demultiplexing where the peers send data with a frequency of 4 Hz simultaneously. These tests were needed to know how long a mobile phone can last under the expected load. Unfortunately, these results show that even if one of these mobile phones were acting as a uplink gateway (i.e., only doing multiplexing) of data from the IoT devices - the operating time for this phone would be less than a working shift in a typical plant (i.e., less than 8 hours).

The results also showed that opening new DataChannels requires more CPU usage than sending messages over DataChannels. For this reason, we can state that DataChannels should be kept open by IoT devices rather than opening and terminating them frequently.

Future work and required reflections concerning this chapter are given in the final Chapter.

Chapter 4

LED Module for Earl

This chapter presents the work conducted to design and build an LED module for Earl. The chapter starts with background information and explains the goal of this part of the thesis project. The chapter continues with a description of the components, the final schematic, and the PCB designed for the LED module. This is followed by the description of the Inter-Integrated Circuit (I^2C) interface which is used by Texas Instruments MSP430* to drive the LED module. Following this, the source code of LED module is included. This chapter concludes with an overview of the work done.

4.1 Background

As described in Chapter 1, technology probes are simple and flexible devices used in a design process aiming to understand the needs and desires of users. This technology probe was to be a small ball with LEDs controlled by an Earl. This probe (called the ball-probe) was designed to change color and brightness based upon interactions of the user. An example use case was to locate this ball in a rest area used by the control room workers and LEDs would be varied depending upon on the number of people who had been in this rest area. It should be possible to vary the LED, brightness and color with commands via the Earl device. As Earl did not provide an LED module a new Earl module was designed. Rather than focusing on the design of the technology probe itself, this chapter focuses on building the LED module that is merged with Earl.

* Model of MSP430 used in Earl is MSP430G2553.

4.2 Goal

Earl was designed to have components such as light sensors, motors, etc. Designing and prototyping interactive solutions was thought to be easier by using Earl with various components (which could be implemented as modules). In this project, an LED module was needed for the ball-probe. The following activities were defined as the goals of this part of the thesis project:

- Design and build an LED module such that Earl can control the brightness and color of the LEDs.
- Implement a software library that can be used to control the LED module and provide an I^2C library for additional future components.

4.3 Components

While Earl is designed to have peripherals, it is not able to drive LEDs, hence a separate LED module was needed. The LED module consists of an LED driver (Texas Instruments TCA6507 [119]) and two multicolor LEDs (OSRAM LRTBG6TG) [120]. One of the 7 outputs of TCA6507 was not used because two output was used for each color of each LED.

The LED module should provide different colors and different outputs, such as blinking and fading. The color of each LED should be controlled separately. The TCA6507 LED driver was chosen to control the LEDs. The reasons to use the TCA6507 LED driver TCA6507 are:

- The TCA6507 provides 7 LED outputs and they can be set as: On, off, blinking and fading at programmable rates. It has two independent blinking modules: PWM0 and PWM1. It provides 16 steps of brightness from fully off to fully on states. This was sufficient to create interaction patterns for use by the users of the ball-probe.
- The TCA6507 works with low power as does Earl. Earl's voltage regulator outputs 3.3 V and the TCA6507 is optimized to work between 1.65 V and 3.6 V.
- The TCA6507 can be programmed through a two-line bidirectional bus, called I^2C, which is also supported by MSP430. The I^2C interface is simple and flexible to use. It will be explained in Section 4.5
- Each output port of the TCA6507 can support up to 40 mA of sink current to drive an LED. However, this output drive current is higher than what a

processor such as MSP430 can deliver directly. Because we need to drive the LED with large currents (to have sufficient brightness) we used the TCA6507 rather than driving the LEDs directly from the processor. The maximum output of the digital outputs of the processor is 6 mA and the total output power of the processor is limited to a sum of 48mA. Further note that the version of the MSP430 processor that was used does not have a digital to analog converter.

For the LEDs, two OSRAM LRTBG6TG LEDs were used. The reason for choosing these LEDs were:

- LRTBG6TG has three colors (red, green, and blue) and each color can be controlled separately to display various colors.
- The viewing angle of LRTBG6TG is 120 °. This was convenient for the design of the chosen ball. A narrow angle such as 30 ° would light only some part of the ball and a more wider angle was not needed.
- LRTBG6TG is an SMD (Surface-mount device) LED that can be soldered directly onto the surface of a PCB. SMD LEDs are tiny and provide many colors while using only a small amount of board space. This leads to a more compact board.

4.4 Schematic

The module was designed using CadSoft's Eagle PCB Software version 6 [121]. The size of the module's board was designed to be 3.0 x 2.5 cm so that it fits in the ball-probe together with an Earl and a battery *. The PCB footprints and schematic symbols of the TCA6507 are available for download from Texas Instruments' website [122]. The LRTBG6TG's schematic and PCB footprint can be found in the Github repository of a project called pentawall [123]. Figure 4.1 shows the resulting schematic. All of the components are listed in Table 4.1. The lower LED is attached to ports P1:P3 of the TCA6507; while the upper LED is attached to ports P4:P7. P0 is unused in this design. The 5 pin connector JP1 is used to connect this module via an I2C interface to the Earl. A prototype of PCB was milled out with the help of Jordi Solsona, a Ph.D. student working at Mobile Life. An LPKF ProtoMat S42 [124] was used to mill out the PCB, and a double sided FR4 sheet with 35 micrometers copper plating was used. Figure 4.2 shows the PCB design.

* The size of the Earl board is 5 x 35 x 25 mm. The size of the battery is 5.7 x 29.5 x 48.27 mm, and the diameter of the ball is 50 mm

Table 4.1: Component list of the LED module

Name	Value	Distributer:PartID	Details
TCA6507	-	Farnell:1647813	Texas Instruments, TCA6507PW, LED Driver, 7CH, I^2C, SMB, TSSOP14
LRTBG6TG	-	Farnell:1244114RL	Osram, LRTBG6TG MULTILED, SMD, RGB
R1, R2, R5, R6	10 Ohm	Farnell:2078941	Package:0402
R3, R4	68 Ohm	Farnell:2078947	Package:0402
JP1	-	Sparkfun:PRT-00116	Pin Spacing: 2.54 mm. Pin length: 11.5 mm

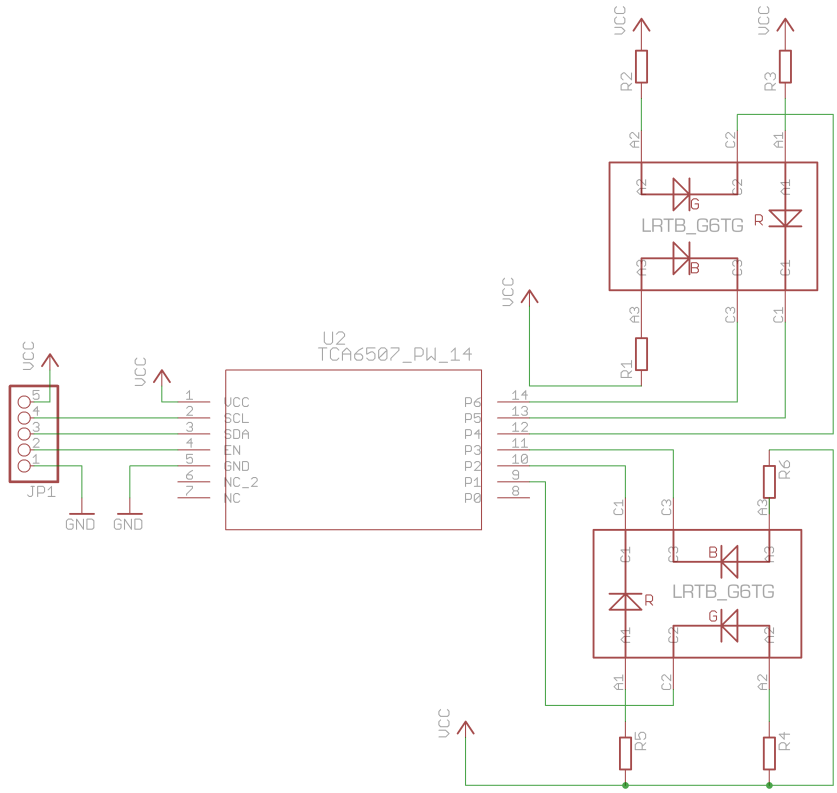


Figure 4.1: Schematic of the LED module

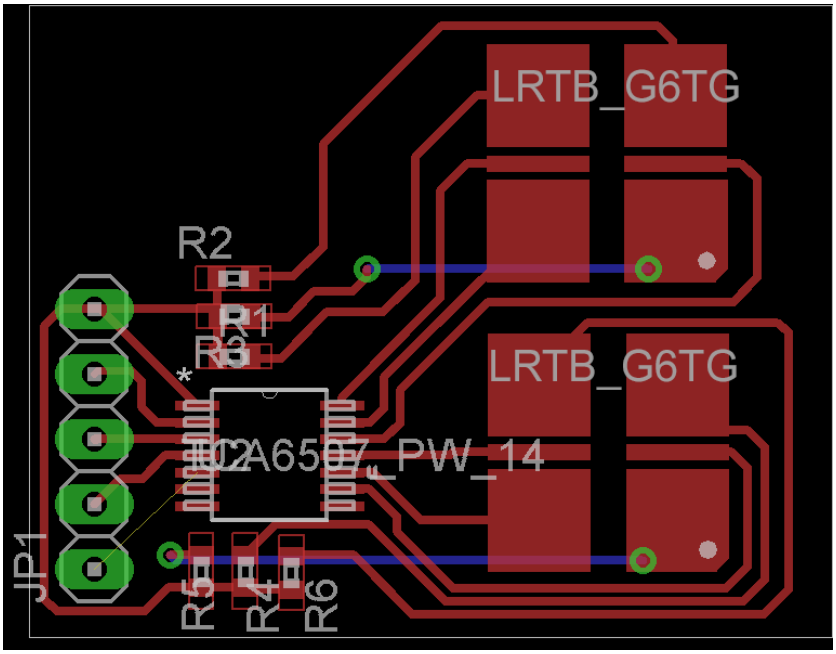


Figure 4.2: PCB of the LED module

After printing the PCB and soldering the components under a microscope, the module was tested with another microprocessor, mbed NXP LPC1768 Microcontroller [3] since mbed provides a very simple I²C library. Earlier testing with a working I²C library on mbed provided a separation of hardware and software problems which could occur. After ensuring that the LED module was working properly, the software to control this module was implemented for Earl.

4.5 I²C Interface

I²C bus was invented by Philips Semiconductors in the early 1980's. Its main purpose is to provide a communication link between integrated circuits [125].

I²C uses two bidirectional lines: Serial Data Line (SDA) and Serial Clock Line (SCL). Figure 4.3 shows an overview of the I²C bus. Each device on the I²C bus has a unique address (either 7 bits or 10 bits) and can be a slave or a master. The master device (MSP430 in this case) starts the communication with the slave(s) (the TCA6507 in this case). The master device generates one clock pulse for each data bit transferred. The data is operated as a byte and each byte is transferred most significant bit first. First a START condition is created by the master. A START condition is a high to low transition of SDA while SCL is high. Then 7 bit address of the slave (TCA6507 has 7-bit address) follows the start bit, and then a bit representing a read (1) or write (0) to the slave is sent. If a slave's

address matches the address sent by the master, the slave responses with an ACK bit. Once the master receives the ACK response, transfer of data can start. Any number of data bytes can be transferred between a START and STOP condition. After the transfer of each byte, one ACK bit should be transferred by the receiver. When the transmission is over, the master creates a STOP condition. A STOP condition is a low to high transition of SDA while SCL is high. Figure 4.4 depicts a data transfer of 2 bytes.

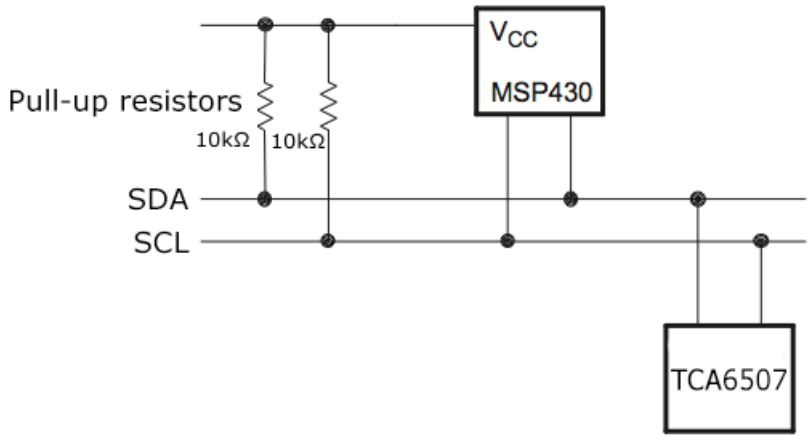


Figure 4.3: General overview of I^2C bus. SCL and SDA are pulled up with resistors. The LED module does not have these resistors as Earl already has these resistors.

Number of bits	1	7	1	1	8	1	8	1	1
Type	Start	Slave Adress	R/$\overline{W}$	ACK	Data	ACK	Data	ACK	Stop

Figure 4.4: Data flow on I^2C bus

MSP430 supports multiple serial communication modes with its universal serial communication interface (USCI) modules. Different USCI modules (each named with a letter and a number such as USCIA0) support different modes. USCIB modules support I^2C.

TCA6507 includes 11 registers to set the brightness and function of the LEDs. The MSP430 controls the LEDs via the TCA6507. A common control flow can would be:

1. Send a START condition.
2. Send the slave address of the TCA6507 plus a write bit (1000101 0).

3. Send a byte to indicate which register in the TCA6507 should be written.
4. Send data to be written.
5. Send a STOP condition.

TCA6507 also provides a mode called "auto-increment" in which data is written in to the registers in a sequence by incrementing the register address which is set in step 3. In this mode, step 4 is repeated to send data in a sequence. This mode can be set in the byte sent in step 3. Note that the maximum operating frequency of both the TCA6507 and the MSP430 used on the Earl are both 400 kHz in I^2C mode, however in this project the data rate (of SCL) will be set to 100 kHz. This data rate was chosen because Earl has built-in 10 kohm pull-up resistors which are typical values for 100 kHz (standard mode).

4.6 Software

This section gives information about the development of the software to control TCA6507 over I^2C. Earl's software is separated into different modules which are each responsible for different functionalities, such as scheduling and ANT communication. The LED driver module is one of these modules.

For the code development, TI's Code Composer Studio (CCS) integrated development environment (IDE) v4.2.4 [126] was used. The CCS IDE includes a compiler for TI's MSP430 microprocessors. CCS IDE was installed on a computer running Microsoft Windows 7 Professional as its operating system. The software for the LED module was written in ANSI C language. During the development both a Texas Instruments' MS430 Launch Pad [127] and Earl were used to test the LED Driver software module *.

The LED driver module is separated into two files: `i2c.c` and `tca6507.c`. `i2c.c` contains general functions to control I^2C interface. `tca6507.c` controls the TCA6507 by using the functions of `i2c.c`. The pseudo code of the program is explained below and the actual code can be found in the Github repository [41]. Documentation of MSP430 and TCA6507 must be studied to understand the whole code.

Listing 4.1: Pseudo code of functions in i2c.c

```
1 void function initI2C (slave)
2       Assign pins to USCI_B0
3       Enable SW reset to configure USCI
4       Set as Master
```

* The MSP430 LaunchPad also used the same version of the MSP430 as the Earl.

```
5       Set SCL frequency to 100 kbit/s standard mode
6       Set slave adress
7       Clear SW reset
8       Enable interrupts
9
10 void function writeI2C (bytesArray)
11      Initialize global bytesToWrite array with bytesArray
12      Send a START condition
13
14 interrupt TX_ISR
15      if bytesToWrite completed
16            Send a STOP condition
17      else
18            Load TX buffer from bytesToWrite
```

Listing 4.1 shows functions to use I^2C bus from MSP430. The data is sent byte by byte in the transmission interrupt. After a transmission of a byte, the array of bytes to be sent is checked for whether it is empty or not. If this array is empty, it means the transmission is over, and a STOP condition will be sent. If the array is not empty, the transmission continues with the next byte in the array.

Listing 4.2: Pseudo code of functions in tca6507.c

```
1  void function initTCA6507 (portValues)
2       Initialize bytesArray with portValues
3       Set auto increment of TCA6507
4       writeI2C (bytesArray)
5
6  void function transmitCommand (registerAddress, value)
7       Initialize bytesArray with registerAddress and value
8       writeI2C (bytesArray)
9
10 char function incrementPWMx
11      Increment the count for PWMx
12      Initialize portValues
13      initTCA6507 (portValues)
14      Return count
15
16 char function decrementPWMx
17      Decrement the count for PWMx
18      Initialize portValues
19      initTCA6507 (portValues)
20      Return count
21
```

```
22 void function testPorts
23       initI2C (slave)
24       Foreach port of tca6507
25             Set the port to ON state in portValues
26             initTCA6507 (portValues)
27             Delay
28             Set the port to OFF
29
30 void function testCount
31       initI2C (slave)
32       while(1)
33             incrementPWM()
34             Delay
```

Listing 4.2 shows functions which exploit the general I^2C functions to control TCA6507. By using `incrementPWMx` or `decrementPWMx` function, the brightness of initialized LEDs can be set. The color can be chosen by setting port values by `initTCA6507`. Test functions were used to examine whether the LED module could drive the LEDs properly. The test functions showed that the LED module worked as expected.

4.7 Conclusion

This chapter described the design and implementation of an LED module in terms of both its software and a PCB prototype. I^2C interface was described and a sample of its usage was shown with the LED module. An I^2C library was implemented for Earl and this can be used as a reference for future components of Earl.

The goal of Earl is to have many components, such as LED modules, temperature sensors, etc. With such components, designers can quickly discover product ideas by developing proof of concept systems or technology probes as was described in this chapter.

Chapter 5

Conclusion

This chapter presents the overall conclusion, future work, and required reflections.

5.1 Conslusion

This thesis project investigated both the WebSocket and WebRTC protocols. These two protocols could exploit a mobile phone as a gateway to connect IoT devices to the Internet. As discussed in Chapter 1, there are two main approaches to IoT today: using gateways and embedding web servers into devices. This thesis project examines the approach of using a mobile phone as a gateway. This approach takes advantage of an Internet browser in the mobile phone in order to exploit this browser's implementation of these two real-time web protocols.

This thesis describes a series of tests of WebSocket and WebRTC in order to evaluate their limits in terms of scalability. The WebSocket tests showed that it is possible to scale the servers up by introducing a load balancer. These tests used a Node.JS server and a publisher-subscriber mechanism to simulate IoT and clients connected to them. For a message frequency of 4 Hz, one subscriber per publisher, and two load balanced server instances, the two servers could support a total of 1600 publishers. This was sufficient for this project's requirements. Note that the actual performance depends on the chosen server side technology, the power of the server instances, the regions, where the virtual machines are allocated and the number of servers. The number of clients that could be served by a given number of servers can be improved with more powerful physical servers. However, using more powerful servers comes at a higher price.

WebRTC is potentially a game changer due to its JavaScript API and P2P architecture. Today, JavaScript is the main language for web development. WebRTC takes advantage of this by providing a WebRTC API to web browsers without requiring any plugins. The test results showed that using this API a

mobile phone is able to forward data coming from 20 peers with a one-way delay of 400 ms on average. Although application layer forwarding performs worse than network layer forwarding, the application layer forwarding could meet the requirements of the project. The result is that any mobile phone running a browser that supports WebRTC could be used as a platform for application layer forwarding (note that this is independent of the OS of the phone).

The WebRTC test results showed that network layer forwarding has higher performance (lower delay and higher throughput) than application layer forwarding. As stated in the first Chapter, 6LoWPAN enables the latest Internet protocols to be used in low power embedded devices. It is expected that most future embedded devices will be seamlessly integrated into the Internet. Thus, forwarding IP packets from a 6LoWPAN device could provide higher performance than application layer forwarding with the Earl device.

The LED module was implemented for Earl, however it could also be used with MSP430 Launchpad for further projects if the version of MSP430 is the same as the Earl. Moreover, the I^2C library was implemented for Earl and this could be extended for future components of Earl or MSP430 Launchpad.

5.2 Future Work

Possible future work includes:

- WebSocket tests can be extended to use a secure WebSocket. If a secure WebSocket is used then the communication will take place over Transport Layer Security (TLS). This will require increased resources and cause increased latency, thus testing should be performed to quantify these resource requirements.

- In the WebSocket tests the clients were configured as publishers and subscribers to simulate devices which send data to clients connected to them. In these tests only the publishers sent messages. These tests could be extended to the case where subscribers sending data back to the publishers. In that case, each publisher would also be a subscriber and the number of simultaneous clients that could be supported with the configurations described in these thesis would be decreased because of the increased load on the application servers. It would be useful to quantify how large this decrease in performance would be.

- In the WebRTC tests the data flow was from a laptop to another laptop through a mobile phone. The data flow was one-way. These tests could

be extended to the case where a receiving laptop also sent data back to the sending laptop.

- WebRTC multiplexing and demultiplexing tests could be extended to the case where a mobile phone did both multiplexing and demultiplexing simultaneously.

- The Earl toolkit should be introduced into the tests in order to measure the end-to-end latency from an Earl device to a client endpoint. As the WebSocket and WebRTC tests described in this thesis start from the mobile phone, one must add the latency of the connection to the Earl device and the processing latency within the phone. These end-to-end measurements of Earl based IoT devices communicating with Internet attached client would be more representative of what the performance of IoT devices would be (if they used Earl with a mobile phone gateway). A simple test of this would be to use the LED module together with a light level sensor to change the LED's brightness level according to some pattern compare this with the brightness as detected by the light level sensor (thus measuring the servo loop from emitter to detector via the Earl device via a mobile phone acting as a gateway to a server running in a virtual machine in the cloud).

5.3 Required Reflections

In this thesis project I had chance to do both web programming and embedded software programming. This combination of programming is expected to be increasingly common as more and more IoT devices connect to the Internet. During the development of this thesis, I came to understand that it is very important to agree on responsibilities and requirements of a project before starting to work on it.

This thesis project facilitated the usage of the Earl toolkit in the design process of technology probes. The data that has been collected should help designers and users formulate new product ideas to help the control room workers avoid boredom. New product ideas, generated based upon the collected data, would be a social benefit of this thesis, if these products actually proved to be beneficial to people's lives. Moreover, in the ABB project the prevention of boredom in the control rooms is expected to increase the performance of workers, which would be an economic contribution to the company. In the experiments EC2 servers were used to take advantage of the reasonable costs and ease of scaling up services running in a cloud environment. However, dedicated hardware could be more economical as part of a long term solution for a company operating a control room (assuming that the desired functions were integrated into the design and

implementation of the control room). The experiments conducted in this thesis project used very simple data solely for testing. Thus, there was no violation of ethical values such as privacy of control room workers. Technology probes are supposed to be temporary solutions to formulate new product ideas which could become permanent. This thesis project was not concerned about the sustainability of the solutions since technology probes are temporary devices. If the ideas formulated by technology probes turn into a product, the software and hardware should be implemented with the concern of sustainability.

Bibliography

[1] D. Evans, "The Internet of Things How the Next Evolution of the Internet Is Changing Everything," Cisco Internet Business Solutions Group, Tech. Rep., Jun. 2011. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[2] "Arduino," accessed: 2013-10-08. [Online]. Available: http://arduino.cc

[3] "mBed," accessed: 2013-10-08. [Online]. Available: http://mbed.org

[4] "littleBits," accessed: 2013-10-08. [Online]. Available: http://littlebits.cc

[5] "Mobile Life Research Institute," accessed: 2013-10-08. [Online]. Available: http://www.mobilelifecentre.org/

[6] D. Akkaya, "Wireless inspirational bits for facilitating early design," Master's Thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, System on Chip Design, May 2013, TRITA-ICT-EX-2013:202.

[7] "ANT - Wireless Sensor Network Solution," accessed: 2013-10-08. [Online]. Available: http://www.thisisant.com

[8] Bluetooth Special Interest group, "BLE - Low Energy Bluetooth," accessed: 2013-10-08. [Online]. Available: http://www.bluetooth.com/Pages/Low-Energy.aspx

[9] Texas Instruments, "MSP430 Microcontroller," accessed: 2013-10-08. [Online]. Available: http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page?DCMP=MCU_other&HQS=msp430

[10] "ABB," accessed: 2013-11-27. [Online]. Available: http://www.abb.se/

[11] H. Hutchinson, W. Mackay, B. Westerlund, B. B. Bederson, A. Druin, C. Plaisant, M. Beaudouin-Lafon, S. Conversy, H. Evans, and H. Hansen, "Technology probes: inspiring design for and with

families," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2003, p. 17–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=642616

[12] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008. doi: 10.1145/1460412.1460415. ISBN 978-1-59593-990-6 p. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460415

[13] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt

[14] D. Guinard, "A Web of Things Application Architecture – Integrating the Real-World into the Web," Ph.D. dissertation, ETH Zurich, Information Management, Auto-ID Labs, 2011. [Online]. Available: http://webofthings.org/dom/thesis.pdf

[15] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "The web of things: Interconnecting devices with high usability and performance," in *International Conference on Embedded Software and Systems, 2009. ICESS '09*, 2009. doi: 10.1109/ICESS.2009.13 pp. 323–330.

[16] D. Guinard, "Towards the web of things: Web mashups for embedded devices," in *In MEM 2009 in Proceedings of WWW 2009. ACM*, 2009.

[17] "OpenPicus FlyPort," accessed: 2013-10-08. [Online]. Available: http://wiki.openpicus.com/index.php?title=FLYPORT

[18] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-Based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE Trans. Serv. Comput.*, vol. 3, no. 3, pp. 223 – 235, Jul. 2010. doi: 10.1109/TSC.2010.3. [Online]. Available: http://dx.doi.org/10.1109/TSC.2010.3

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2068 (Proposed Standard), Internet Engineering Task Force, Jan. 1997, obsoleted by RFC 2616. [Online]. Available: http://www.ietf.org/rfc/rfc2068.txt

[20] J. Garrett, "Ajax: A New Approach to Web Applications," Tech. Rep., 2005. [Online]. Available: http://www.adaptivepath.com/publications/essays/archives/000385.php

[21] E. Bozdag, A. Mesbah, and A. van Deursen, "A comparison of push and pull techniques for AJAX," in *9th IEEE International Workshop on Web Site Evolution, 2007. WSE 2007*, 2007. doi: 10.1109/WSE.2007.4380239 pp. 15–22.

[22] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[23] I. Hickson, "The websocket api," W3C Editor's Draft, Apr. 2013. [Online]. Available: http://dev.w3.org/html5/websockets/

[24] "Parse," accessed: 2013-08-15. [Online]. Available: https://www.parse.com/

[25] "Xively Public Cloud for the Internet of Things," accessed: 2013-08-15. [Online]. Available: https://xively.com/

[26] "Pusher HTML5 WebSocket Powered Realtime Messaging Service," accessed: 2013-08-15. [Online]. Available: http://pusher.com/

[27] C. A. Gutwin, M. Lippold, and T. C. N. Graham, "Real-time groupware in the browser: testing the performance of web-based networking," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, ser. CSCW '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1958824.1958850. ISBN 978-1-4503-0556-3 p. 167–176. [Online]. Available: http://doi.acm.org/10.1145/1958824.1958850

[28] P. Lubbers and F. Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web," accessed: 2013-08-15. [Online]. Available: http://www.websocket.org/quantum.html

[29] D. G. Puranik, D. C. Feiock, and J. H. Hill, "Real-time monitoring using AJAX and WebSockets." [Online]. Available: http://cs.iupui.edu/~hillj/PDF/ecbs2013-websockets.pdf

[30] C. Marion and J. Jomier, "Real-time collaborative scientific WebGL visualization with WebSocket," in *Proceedings of the 17th International Conference on 3D Web Technology*, ser. Web3D '12. New York, NY, USA: ACM, 2012. doi: 10.1145/2338714.2338721. ISBN 978-1-4503-1432-9 p. 47–50. [Online]. Available: http://doi.acm.org/10.1145/2338714.2338721

[31] "Autobahn Testsuite," accessed: 2013-09-07. [Online]. Available: http://autobahn.ws/testsuite

[32] G. Barish, *Building scalable and high-performance Java Web applications using J2EE technology*. Boston: Addison-Wesley, 2002. ISBN 0201729563 9780201729566

[33] "Cubeia WebSocket Performance," accessed: 2013-09-07. [Online]. Available: http://www.cubeia.com/2012/11/web-socket-performance/

[34] "Realtime Node.js App: Stress Testing Procedure," accessed: 2013-08-15. [Online]. Available: http://weblog.bocoup.com/node-stress-test-procedure/

[35] "Thor - WebSocket Benchmarking/Load Generator," accessed: 2013-08-15. [Online]. Available: https://github.com/observing/thor

[36] "A Benchmarking Suite for Node PubSub Servers," accessed: 2013-08-15. [Online]. Available: https://github.com/mixu/siobench

[37] "Simple Socket.io Benchmark," accessed: 2013-08-15. [Online]. Available: https://github.com/michetti/socket.io-benchmark

[38] "Web Framework Benchmarks," accessed: 2013-08-15. [Online]. Available: http://www.techempower.com/benchmarks/

[39] "Ksar, a sar grapher," accessed: 2013-09-08. [Online]. Available: http://sourceforge.net/projects/ksar/

[40] "GitHub." [Online]. Available: https://github.com/

[41] "My GitHub Repository," accessed: 2013-10-11. [Online]. Available: https://github.com/gmertk

[42] "NTP Home Page," accessed: 2013-10-15. [Online]. Available: http://www.ntp.org/

[43] Slicehost, LLC, "Using NTP to Sync Time on Ubuntu," 8 November 2010, Accessed: 2013-10-25. [Online]. Available: http://articles.slicehost.com/2010/11/8/using-ntp-to-sync-time-on-ubuntu

[44] "Gnuplot Homepage," accessed: 2013-10-25. [Online]. Available: http://gnuplot.sourceforge.net/

[45] "Node.JS in the Industry," accessed: 2013-08-15. [Online]. Available: http://nodejs.org/industry/]

[46] P. Teixeira, *Professional Node.js building Javascript based scalable software*. Hoboken, N.J.; Chichester: Wiley ; John Wiley [distributor], 2012. ISBN 9781118227541 1118227549. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=490422

[47] "Node Optimist," accessed: 2013-08-15. [Online]. Available: https://github.com/substack/node-optimist

[48] "Node Gauss," accessed: 2013-08-15. [Online]. Available: https://github.com/wayoutmind/gauss

[49] "Socket.io Issue-438," accessed: 2013-08-15. [Online]. Available: https://github.com/learnboost/socket.io/issues/438

[50] "Node WebSocket," accessed: 2013-08-15. [Online]. Available: https://github.com/Worlize/WebSocket-Node

[51] "Redis," accessed: 2013-08-15. [Online]. Available: http://redis.io

[52] "node-toobusy," accessed: 2013-10-20. [Online]. Available: https://github.com/lloyd/node-toobusy

[53] "libev Homepage," accessed: 2013-10-25. [Online]. Available: http://software.schmorp.de/pkg/libev.html

[54] "libuv Github Page," accessed: 2013-10-25. [Online]. Available: https://github.com/joyent/libuv

[55] "libev removed," accessed: 2013-10-30. [Online]. Available: https://github.com/joyent/libuv/issues/485

[56] D. Barrett, R. E. Silverman, and B. Robert, *SSH, The Secure Shell: The Definitive Guide*. O'Reilly, 2005. [Online]. Available: http://shop.oreilly.com/product/9780596000110.do

[57] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon EC2 data center," in *Proceedings of the 29th conference on Information communications*. ACM; Piscataway, NJ, USA: IEEE Press, 2010. ISBN 978-1-4244-5836-3 pp. 1163–1171. [Online]. Available: http://dl.acm.org/citation.cfm?id=1833515.1833691

[58] "Linux Increase The Maximum Number Of Open Files," accessed: 2013-09-08. [Online]. Available: http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files/

[59] B. Veal and A. Foong, "Performance scalability of a multi-core web server," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007. doi: 10.1145/1323548.1323562. ISBN 978-1-59593-945-6 pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323562

[60] "Bees With Machine Guns," accessed: 2013-08-15. [Online]. Available: https://github.com/newsapps/beeswithmachineguns

[61] "HAProxy Website," accessed: 2013-08-15. [Online]. Available: http://haproxy.1wt.eu/

[62] "They Use HAProxy," accessed: 2013-08-15. [Online]. Available: http://haproxy.1wt.eu/they-use-it.html

[63] "They Use HAProxy," accessed: 2013-08-15. [Online]. Available: https://code.google.com/p/haproxy-docs/wiki/balance

[64] "HAProxy Documentation," accessed: 2013-08-15. [Online]. Available: http://haproxy.1wt.eu/#doc1.5

[65] "HAProxy Wiki," accessed: 2013-08-15. [Online]. Available: https://code.google.com/p/haproxy-docs/wiki/Configuring

[66] Silver Bucket, "3 Ways to Configure HAProxy for WebSocket," blog entry, 20 September 2012 Accessed: 2013-08-15. [Online]. Available: http://blog.silverbucket.net/post/31927044856/3-ways-to-configure-haproxy-for-websockets

[67] "Balancer Battle," accessed: 2013-08-15. [Online]. Available: https://github.com/observing/balancerbattle

[68] "Nginx WebSocket," accessed: 2013-08-15. [Online]. Available: http://nginx.com/news/nginx-websockets/

[69] "Redis Documentation," accessed: 2013-08-15. [Online]. Available: http://redis.io/documentation

[70] "Redis Persistence," accessed: 2013-10-25. [Online]. Available: http://redis.io/topics/persistence

[71] "redis-stat," accessed: 2013-10-25. [Online]. Available: https://github.com/junegunn/redis-stat

[72] "Redis INFO command," accessed: 2013-10-25. [Online]. Available: http://redis.io/commands/info

[73] "JSON Activity Streams 1.0," accessed: 2013-10-08. [Online]. Available: http://activitystrea.ms/specs/json/1.0/

[74] "Amazon EC2 Instances," accessed: 2013-08-15. [Online]. Available: http://aws.amazon.com/ec2/instance-types/#instance-features

[75] O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, A. Bassi, I. S. Jubert, M. Mazura, M. Harrison, and M. Eisenhauer, *Internet of things strategic research roadmap*. River Publishers, 2011. [Online]. Available: http://books.google.com/books?hl=sv&lr=&id=Eug-RvslW30C&oi=fnd&pg=PA9&dq=Internet+of+Things+Strategic+Research+Roadmap&ots=3SybyDiCCu&sig=mieegMvgmSQouLfBgYSSX7aIAm8

[76] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: design and implementation of interoperable and evolvable sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008. doi: 10.1145/1460412.1460438. ISBN 978-1-59593-990-6 p. 253–266. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460438

[77] V. Pimentel and B. Nickerson, "Communicating and displaying real-time data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012. doi: 10.1109/MIC.2012.64

[78] "On HTTP Load Testing," accessed: 2013-10-15. [Online]. Available: http://www.mnot.net/blog/2011/05/18/http_benchmark_rules

[79] J. Little and S. Graves, "Little's Law," accessed: 2013-09-02. [Online]. Available: http://web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf

[80] "Skype," accessed: 2013-08-15. [Online]. Available: https://www.skype.com/

[81] "Google Hangout," accessed: 2013-08-15. [Online]. Available: http://www.google.com/+/learnmore/hangouts/

[82] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core," RFC 6120 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6120.txt

[83] ——, "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence," RFC 6121 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6121.txt

[84] ——, "Extensible Messaging and Presence Protocol (XMPP): Address Format," RFC 6122 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6122.txt

[85] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261 (Proposed Standard), Internet Engineering Task Force, Jun. 2002, updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[86] Harald Alvestrand, "Overview: Real time protocols for brower-based applications," IETF draft, September 3, 2013, expires on March 7, 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-rtcweb-overview-08

[87] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6096, 6335. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[88] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347 (Proposed Standard), Internet Engineering Task Force, Apr. 2006, obsoleted by RFC 6347, updated by RFC 5746. [Online]. Available: http://www.ietf.org/rfc/rfc4347.txt

[89] J. Postel, "User Datagram Protocol," RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[90] R. Jesup, S. Loreto, and M. Tuexen, "RTCWeb Data Channels," Internet-Draft, Internet Engineering Task Force, Feb. 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-04

[91] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. [Online]. Available: http://www.ietf.org/rfc/rfc1034.txt

[92] ——, "Domain names - implementation and specification," RFC 1035 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604. [Online]. Available: http://www.ietf.org/rfc/rfc1035.txt

[93] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550 (INTERNET STANDARD), Internet Engineering Task Force, Jul. 2003, updated by RFCs 5506, 5761, 6051, 6222, 7022. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[94] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010, updated by RFC 6336. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[95] J. Postel, "Internet Protocol," RFC 791 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1349, 2474, 6864. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[96] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5389.txt

[97] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5766.txt

[98] "Does WebRTC support IPv6?" accessed: 2013-09-30. [Online]. Available: http://searchunifiedcommunications.techtarget.com/tip/Does-WebRTC-support-IPv6

[99] "WebRTC Internals - VoENetwork, EnableIPv6," accessed: 2013-09-30. [Online]. Available: http://www.webrtc.org/reference/webrtc-internals/voenetwork#TOC-EnableIPv6

[100] "Chrome WebRTC," accessed: 2013-08-15. [Online]. Available: http://www.webrtc.org/chrome

[101] "Firefox WebRTC," accessed: 2013-08-15. [Online]. Available: http://www.webrtc.org/firefox

[102] A. Johnston, J. Yoakum, and K. Singh, "Taking on WebRTC in an enterprise," *Communications Magazine, IEEE*, vol. 51, no. 4, p. 48–54, 2013. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6495760

[103] J. Hautakorpi, G. Camarillo, R. Penfield, A. Hawrylyshen, and M. Bhatia, "Requirements from Session Initiation Protocol (SIP) Session Border Control (SBC) Deployments," RFC 5853 (Informational), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5853.txt

[104] J. K. Nurminen, A. J. Meyn, E. Jalonen, Y. Raivio, and R. G. Marrero, "P2P media streaming with HTML5 and WebRTC." [Online]. Available: http://cse.aalto.fi/wp-content/uploads/2012/09/InfocommPaper_CR.pdf

[105] V. Singh, A. A. Lozano, and J. Ott, "Performance analysis of receive-side real-time congestion control for WebRTC," in *Proc. of IEEE Packet Video*, vol. 2013, 2013. [Online]. Available: http://www.netlab.tkk.fi/~varun/singh2013rrtcc.pdf

[106] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," RFC 4566 (Proposed Standard), Internet Engineering Task Force, Jul. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4566.txt

[107] J. Marcon and R. Ejzak, "SDP-based WebRTC data channel negotiation." [Online]. Available: http://tools.ietf.org/html/draft-ejzak-dispatch-webrtc-data-channel-sdpneg-00

[108] "PeerJS," accessed: 2013-08-15. [Online]. Available: http://peerjs.com

[109] "PeerJS API," accessed: 2013-08-15. [Online]. Available: http://peerjs.com/docs/#api

[110] "Speedtest by OOkla," accessed: 2013-12-10. [Online]. Available: http://www.speedtest.net/

[111] "Trepn Profiler," accessed: 2013-10-15. [Online]. Available: https://developer.qualcomm.com/mobile-development/performance-tools/trepn-profiler

[112] "Android-wifi-tether," accessed: 2013-12-10. [Online]. Available: http://code.google.com/p/android-wifi-tether/

[113] "Nexus 4 Root Guide," accessed: 2013-12-10. [Online]. Available: http://forum.xda-developers.com/showthread.php?t=2018179

[114] "Wireshark," accessed: 2013-12-12. [Online]. Available: http://www.wireshark.org/

[115] "Shark For Root," accessed: 2013-12-12. [Online]. Available: https://play.google.com/store/apps/details?id=lv.n3o.shark

[116] "Clock Sync," accessed: 2013-12-12. [Online]. Available: https://play.google.com/store/apps/details?id=ru.org.amip.ClockSync

[117] "Kernel Configuration for brctl," accessed: 2013-12-05. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge#Kernel_Configuration

[118] "Linux cook-mode capture," accessed: 2013-12-23. [Online]. Available: http://www.wireshark.org/lists/ethereal-users/200412/msg00314.html

[119] Texas Instruments, "TCA6507 Home Page," accessed: 2013-12-10. [Online]. Available: http://www.ti.com/product/tca6507

[120] OSRAM, "LRTBG6TG Home Page," accessed: 2013-12-10. [Online]. Available: http://catalog.osram-os.com/catalogue/catalogue.do?favOid=0000000000031bac00060023&act=showBookmark

[121] CadSoft, "Eagle PCB Software," accessed: 2013-12-10. [Online]. Available: http://www.cadsoftusa.com/

[122] Texas Instruments, "TCA6507," accessed: 2013-12-23. [Online]. Available: http://webench.ti.com/cad/dlbxl.cgi/TI_BXL/TCA6507_PW_14.bxl

[123] "Github repository of pentawall," accessed: 2013-12-23. [Online]. Available: https://github.com/sebseb7/pentawall

[124] "LPKF ProtoMat S42," accessed: 2013-12-23. [Online]. Available: http://www.lpkfusa.com/protomat/s42.htm

[125] "I2C Bus," accessed: 2013-12-23. [Online]. Available: http://www.i2c-bus.org/

[126] Texas Instruments, "Code Composer Studio," accessed: 2013-12-10. [Online]. Available: http://www.ti.com/tool/ccstudio

[127] ——, “MSP430 LaunchPad,” accessed: 2013-12-10. [Online]. Available: http://www.ti.com/tool/msp-exp430g2

Appendix A

Network Kernel Configuration

```
## Network kernel parameters for high performance.
## "Performance Scalability of a Multi-Core Web Server", Nov 2007
## Bryan Veal and Annie Foong, Intel Corporation, Page 4/10.

fs.file-max = 5000000

## Increase the socket listening backlog.
net.core.somaxconn = 100000

## Increase backlog for UNIX sockets.
net.unix.max_dgram_qlen = 100

## Maximum backlogs.
net.core.netdev_max_backlog = 400000
net.ipv4.ip_local_port_range=4096 65535
net.core.netdev_max_backlog = 400000
net.ipv4.tcp_max_syn_backlog = 65536
net.ipv4.tcp_max_orphans = 262144

## increase TCP max buffer size setable using setsockopt() 16 MB with
## a few parallel streams is recommended for most 10G paths 32 MB
## might be needed for some very long end-to-end 10G or 40G paths.
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
```

```
25
26 ## Increase Linux autotuning TCP buffer limits min,
      default, and max
27 ## number of bytes to use (only change the 3rd value, and
      make it 16
28 ## MB or more).
29 net.core.rmem_default = 10000000
30 net.core.wmem_default = 10000000
31 net.ipv4.tcp_mem = 30000000 30000000 30000000
32 net.ipv4.tcp_rmem = 30000000 30000000 30000000
33 net.ipv4.tcp_wmem = 30000000 30000000 30000000
34
35 ## More TCP stack stuff.
36 net.ipv4.tcp_mem = 50576 64768 98152
37 net.ipv4.tcp_fin_timeout= 10
38 net.ipv4.tcp_tw_reuse= 1
39 net.ipv4.tcp_tw_recycle= 1
40 net.ipv4.tcp_sack = 1
41 net.ipv4.tcp_syncookies = 0
42 net.ipv4.tcp_timestamps = 1
43 net.ipv4.conf.all.rp_filter = 1
44 net.ipv4.conf.default.rp_filter = 1
45 net.ipv4.tcp_congestion_control = bic
46 net.ipv4.tcp_ecn = 0
47 net.ipv4.tcp_max_tw_buckets = 2000000
```

Appendix B

HAProxy Configuration

```
global
      maxconn 999999
defaults
      mode http
      log global
      option httplog
      option http-server-close
      option dontlognull
      option redispatch
      option contstats
      retries 3
      backlog 10000
      timeout client 25s
      timeout connect 5s
      timeout server 25s
      timeout tunnel 3600s
      timeout http-keep-alive 1s
      timeout http-request 15s
      timeout queue 30s
      timeout tarpit 60s
      default-server inter 3s rise 2 fall 3
      option forwardfor

frontend ft_web
      bind *:80 name http
      maxconn 1000000
      ## routing based on Host header
      acl host_ws hdr_beg(Host) -i ws
      use_backend bk_ws if host_ws
```

```
30
31        ## routing based on websocket protocol header
32        acl hdr_connection_upgrade hdr(Connection) -i upgrade
33        acl hdr_upgrade_websocket hdr(Upgrade) -i websocket
34
35        use_backend bk_ws if hdr_connection_upgrade
             hdr_upgrade_websocket
36        default_backend bk_web
37
38  listen admin
39        bind *:1946
40        stats enable
41        maxconn 2500
42
43  backend bk_web
44        server websrv1 127.0.0.1:1234 maxconn 100 weight 10
             cookie websrv1 check
45
46  backend bk_ws
47        balance roundrobin
48
49        ## websocket protocol validation
50        acl hdr_connection_upgrade hdr(Connection) -i upgrade
51        acl hdr_upgrade_websocket hdr(Upgrade) -i websocket
52        acl hdr_websocket_key hdr_cnt(SecWebSocketKey) eq 1
53        acl hdr_websocket_version
             hdr_cnt(Sec-WebSocket-Version) eq 1
54        acl hdr_host hdr_cnt(Sec-WebSocket-Version) eq 1
55        http-request deny if ! hdr_connection_upgrade !
             hdr_upgrade_websocket ! hdr_websocket_key !
             hdr_websocket_version ! hdr_host
56        acl ws_valid_protocol hdr(Sec-WebSocket-Protocol)
             echo-protocol
57        http-request deny if ! ws_valid_protocol
58
59        ## websocket health checking
60        server websrv1
             ec2-54-216-227-41.eu-west-1.compute.amazonaws.com:8080
             maxconn 500000 weight 10 cookie websrv1
61        #server websrv2
             ec2-46-51-145-150.eu-west-1.compute.amazonaws.com:8080
             maxconn 500000 weight 10 cookie websrv2
```

Appendix C

Example of WebRTC DataChannel API

```
// Configure servers and options
var servers = {
  iceServers: [
    {url: "stun:stunserver.com:12345"},
    {url: "turn:user@turnserver.com", "credential": "pass"}
]};
var options = {
  optional: [
    { RtpDataChannels: true}
  ]
};

// Create PeerConnection
var pc = new RTCPeerConnection(servers, options);
pc.onicecandidate = function(evt) {
  if (evt.candidate) {
    signalingChannel.send(JSON.stringify({"candidate":
       evt.candidate}));
  }
};

// Create DataChannel
dataChannel =
   pc.createDataChannel("DataChannel",{reliable: false});
dataChannel.onmessage = onDataChannelMessage;
function onDataChannelMessage(ev){
  console.log('Received message: ' + event.data);
```

```
26 }
27
28 // Start by creating an offer to other peer
29 pc.createOffer(function(desc){
30   pc.setLocalDescription(desc);
31   signalingChannel.send(JSON.stringify({"offer": desc}));
32 });
33
34 // SignalingChannel constructor is an abstraction
35 // for a signaling mechanism such as WebSocket.
36 var signalingChannel = new SignalingChannel();
37 signalingChannel.onmessage = function(msg) {
38   msg = JSON.parse(msg);
39   if (msg.offer) {
40     pc.setRemoteDescription(JSON.parse(msg.offer));
41
42     pc.createAnswer(function (answer) {
43       pc.setLocalDescription(answer);
44       signalingChannel.send(JSON.stringify({"answer":
             answer}));
45     }, errorHandler, constraints);
46   }
47   else if (msg.candidate) {
48     pc.addIceCandidate(msg.candidate);
49   }
50 };
```

Appendix D

Google's STUN Servers

- stun.l.google.com:19302
- stun1.l.google.com:19302
- stun2.l.google.com:19302
- stun3.l.google.com:19302

TRITA-ICT-EX-2014:2

www.kth.se