

Aalto University
School of Science
Double Degree Programme NordSecMob

Eliot Estep

Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events

Master's Thesis
Espoo, June 28, 2013

Supervisors: Professor Jukka Nurminen, Aalto University
 Professor Markus Hidell, KTH Royal Institute of Technology

Instructor: Matti Siekkinen, M.Sc. Ph.D.

Aalto University
 School of Science
 Double Degree Programme NordSecMob

ABSTRACT OF
 MASTER'S THESIS

Author:	Eliot Estep		
Title:	Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events		
Date:	June 28, 2013	Pages:	109
Professorship:	NordSecMob	Code:	T-110
Supervisors:	Professor Jukka Nurminen, Aalto University Professor Markus Hidell, KTH Royal Institute of Technology		
Instructor:	Matti Siekkinen, M.Sc. Ph.D.		
<p>The advent of HTML5 (Hyper Text Markup Language revision 5) technologies are re-standardizing the web and paving the way for a new breed of real-time web applications. This has great potential for mobile browsers that are now supporting protocols such as WebSocket and Server-Sent Events (SSE). These protocols can provide efficient real-time communication in a scalable manner, especially for “always-on” applications requiring persistent connections that are now widely used.</p> <p>Mobile devices are inherently restricted due to their limited battery power and require frequent charging. Therefore, experimentation that potentially promotes breakthroughs in mobile energy efficiency is useful at this time. Extensive measurements were conducted over 3G, 4G, and WiFi networks to analyze the performance of WebSocket and SSE across a variety of popular mobile devices, browsers, and platforms. Devices were connected to a power monitor for a precise understanding of the energetic effects of keep-alive mechanisms and their overall effects on long-lasting connectivity.</p> <p>The results reveal that application level keep-alive mechanisms are not necessary to sustain the connections indefinitely, given proper server and network configurations. However, to avoid timeouts and to detect prematurely disconnected clients, keep-alive exchanges are necessary and useful in practice. The effects of short and long keep-alive interval values are examined in detail for all devices. Browser performance varies widely as no browser was completely successful for both WebSocket and SSE tests. Further improvements in mobile browser support for these technologies will be necessary to reach the full potential of real-time applications in the future.</p>			
Keywords:	mobile, HTML5, WebSocket, Server-Sent Events, browser support, performance, energy efficiency		
Language:	English		

Aalto-universitetet
Högskolan för teknikvetenskaper
Examensprogram för datateknik

SAMMANDRAG AV
DIPLOMARBETET

Utfört av:	Eliot Estep		
Arbetets namn:	Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events		
Datum:	Den 28 Juni 2013	Sidantal:	109
Professur:	NordSecMob	Kod:	T-110
Övervakare:	Professor Jukka Nurminen, Aalto University Professor Markus Hidell, KTH Royal Institute of Technology		
Handledare:	Matti Siekkinen, M.Sc. Ph.D.		
<p>Tillkomsten av HTML5 (Hyper Text Markup Language revision 5) teknik är återstandardisera webben och bana väg för en ny typ av realtid webbapplikationer. Detta har stor potential för mobila webbläsare som nu stödjer protokoll såsom WebSocket och Server-Sent Events (SSE). Dessa protokoll kan ge effektiv kommunikation i realtid på ett skalbart sätt, särskilt för “alltid-på” applikationer som kräver beständiga anslutningar som nu används i stor utsträckning.</p> <p>Mobila enheter är i sig begränsade på grund av deras begränsade batteri och kräver frekvent laddning. Därför är experiment som potentiellt främjar genombrott i mobil energieffektivitet användbar vid denna tid. Omfattande mätningar utfördes över 3G, 4G och WiFi-nätverk för att analysera resultatet för WebSocket och SSE över en variation av populära mobila enheter, webbläsare och plattformar. Enheter var ansluten till en monitor för en exakt förståelse av de energiska effekter keep-alive-mekanismer och deras samlade effekter på långvarig anslutning.</p> <p>Resultaten visar att applikationsnivå keep-alive-mekanismerna är inte nödvändigt att upprätthålla anslutningarna på obestämd tid, ges rätt server och konfigurationer nätverk. Men för att undvika timeout och att upptäcka tidigt fränkopplade klienter, keep-alive-börser är nödvändiga och användbara i praktiken. Effekterna av korta och långa keep-alive intervallvärdena granskas i detalj för alla enheter. Browser prestanda varierar kraftigt eftersom ingen webbläsare var helt lyckat för både WebSocket och SSE tester. Ytterligare förbättringar av mobila webbläsare stöd för denna teknik kommer att vara nödvändigt för att uppnå den fulla potentialen av realtidsapplikationer i framtiden.</p>			
Nyckelord:	mobil, HTML5, WebSocket, Server-Sent Events, webbläsare stöd, prestanda, energieffektivitet		
Språk:	Engelska		

Acknowledgements

I wish to thank my supervisor Jukka Nurminen for his thoughtful assistance, positive suggestions, and willingness to repeatedly meet and discuss the progress of this thesis. Additionally, I would like to thank Matti Siekkinen for his contributions to all the meetings and his unique insights and feedback. The two of them have been invaluable in steering the direction and success of this work.

I would also like to extend my gratitude to Pranas Butkus for taking the time to solve two crucial issues with the server code, essentially allowing for successful and relevant results to emerge from all the measurements. Thank you very much. Additionally, I would like to thank Simo Veikkolainen from Nokia for meeting with us to discuss the results and for offering his unique feedback.

I am extremely grateful to have had the continuous support of my family, my roommates, and my wonderful girlfriend during this process, keeping me motivated, focused, and positive. Thank you for all your help! Lastly, I would like to thank all my Unseen Teachers, along with the Creator of all life for the continued support and infinite blessings I have received.

Espoo, June 28, 2013

Eliot Estep

Abbreviations and Acronyms

3G	Third Generation Networks
3GPP	3rd Generation Partnership Project
4G	Fourth Generation Networks
API	Application Programming Interface
CSS	Cascading Style Sheets
DCH	Dedicated Channel, as in CELL_DCH state
DRX	Discontinuous Reception
DOM	Document Object Model
EUI	Event Update Interval
FACH	Forward Access Channel, as in CELL_FACH state
FD	Fast Dormancy
HB	Heartbeat
HBI	Heartbeat Interval
HTML5	Hyper Text Markup Language 5th revision
IE10	Internet Explorer 10
IP	Internet Protocol
JS	JavaScript
JSON	Java Script Object Notation
LTE	Long Term Evolution
MIME	Multipurpose Internet Mail Extension
OMA	Open Mobile Alliance Mobile Phone Standards
PCH	Paging Channel, as in CELL_PCH state
RRC	Radio Resource Control
SCRI	Signaling Connection Release Indicator
SMS	Short Message Service
SSE	Server-Sent Events
SSL	Secure Sockets Layer
UMTS	Universal Mobile Telecommunications System
VoIP	Voice over Internet Protocol
WS	WebSocket

Contents

Abbreviations and Acronyms	5
1 Introduction	10
1.1 HTML5	10
1.2 Communication Technologies	12
1.3 Problem statement	12
1.4 Structure of the Thesis	13
2 Mobile Energy Efficiency	15
2.1 Addressing The Need	16
2.1.1 Common Approaches	16
2.1.2 Target Areas	17
2.2 Key Principles	18
2.2.1 Measuring Energy Consumption	18
2.2.2 Energy Spending in Mobile Devices	19
2.2.3 Efficient Practices	19
2.2.3.1 Reducing State Changes	20
2.2.3.2 Parallel Transfers Save Energy	20
2.3 Radio States for 3G UMTS	21
2.4 Radio States for 4G LTE	23
2.4.1 Discontinuous Reception	23
2.4.2 Fast Dormancy	23
2.5 Always-On Applications	24
2.5.1 RRC Transitions	24
2.5.2 Stateful Middleboxes	25
2.5.2.1 Timeout Values	25
3 Persistent Communication	27
3.1 Persistent HTTP Connections	28
3.1.1 Advantages	28
3.1.2 Disadvantages	28

3.2	Keep-Alive Mechanisms	29
3.2.1	PING/PONG	29
3.2.2	TCP Keep-Alive	30
3.3	Web Techniques	31
3.3.1	Push vs. Pull	31
3.3.2	AJAX	31
3.3.2.1	Polling	33
3.3.2.2	Comet/Long-polling	33
3.3.2.3	Comet/HTTP Streaming	34
3.3.3	Major Issues	34
4	HTML5 Protocols	36
4.1	WebSocket	37
4.1.1	Key Characteristics	37
4.1.2	Design Principles	38
4.1.3	Connection Setup	39
4.1.3.1	Client Handshake	39
4.1.3.2	Server Handshake	39
4.1.4	Data Transfer	40
4.1.4.1	Control Frames	40
4.1.4.2	Keep-Alive Methods	41
4.1.5	Using the WebSocket API	41
4.1.5.1	Creating a Connection	41
4.1.5.2	Handling Events	42
4.1.5.3	Sending Data	42
4.1.6	WebSocket Applications	42
4.1.7	Application Potential	43
4.1.8	Mobile Usage	43
4.2	Server-Sent Events	43
4.2.1	Key Characteristics	44
4.2.2	Design Principles	45
4.2.3	Using the EventSource API	45
4.2.3.1	Client-side	46
4.2.3.2	Server-side	46
4.2.4	Automatic Reconnection	47
4.2.5	Keep-Alive Methods	48
4.2.6	Application Potential	48
4.2.7	Mobile Usage	49
4.2.7.1	Connectionless Push	49

5	Measurements	51
5.1	WebSocket Experiment	52
5.1.1	Objectives	52
5.1.2	Libraries Used	52
5.1.3	Tools Used	54
5.1.4	Application Code	54
5.1.4.1	Heartbeat Messages	55
5.1.5	Procedure	55
5.2	Server-Sent Event Experiment	57
5.2.1	Objectives	57
5.2.2	Libraries Used	58
5.2.3	Tools Used	58
5.2.4	Application Code	58
5.2.4.1	Event Update Intervals	59
5.2.5	Procedure	59
6	WebSocket Results	61
6.1	Initial Results	61
6.1.1	Improper Server Code	62
6.2	Key Findings	62
6.2.1	Device-specific Findings	63
6.3	Long Heartbeat Interval Results	64
6.3.1	Browser Support	65
6.3.2	5 Minutes	66
6.3.3	15 Minutes	66
6.3.4	70 Minutes	66
6.3.5	10 Hours	66
6.3.5.1	Device Behavior	67
6.3.6	Heartbeat Exchanges	67
6.4	Short Heartbeat Interval Results	67
6.4.1	Results for Nexus One	69
6.4.1.1	3G UMTS Elisa	69
6.4.1.2	WiFi	70
6.4.2	Results for Lumia 820	71
6.4.2.1	3G UMTS Elisa	72
6.4.2.2	3G UMTS Sonera	73
6.4.2.3	4G LTE Sonera	73
6.4.2.4	WiFi	75
6.4.3	Results for iPhone 4S	76
6.4.3.1	3G UMTS Elisa	76
6.4.4	Results for Galaxy S3	77

6.4.4.1	3G UMTS Elisa	77
6.4.4.2	4G LTE Sonera	79
6.5	Packet Behavior Analysis	79
6.5.1	Heartbeat Exchanges	80
6.5.2	WebSocket Disconnection	81
7	Server-Sent Events Results	83
7.1	Initial Results	83
7.2	Key Findings	84
7.2.1	Methodology	85
7.3	Long Event Interval Results	85
7.3.1	5 Minutes	86
7.3.2	15 Minutes	87
7.3.3	30 Minutes	87
7.3.4	60 Minutes	87
7.3.5	10 Hours	87
7.3.5.1	Device Behavior	88
7.3.6	Event Update Exchanges	88
7.4	Short Event Interval Results	88
7.4.1	Results for Nexus One	89
7.4.2	Results for iPhone 4S	90
7.4.3	Results for Galaxy S3	90
7.4.3.1	Opera Mobile Browser	91
7.4.3.2	Chrome Browser	92
7.4.3.3	Android Browser	93
7.5	Heartbeat Analysis	95
7.6	Packet Behavior	96
8	Discussion	98
8.1	Performance Implications	98
8.1.1	Application Types	100
8.1.2	Server Configuration	100
8.1.3	Heartbeat Interval Performance Trade-off	101
9	Conclusions	102
9.1	Future Work	102
9.1.1	Mobility	102
9.1.2	Additional Testing	103
9.1.3	Connectionless Push	103
9.2	Future Potential	103

Chapter 1

Introduction

The evolution of digital content and media consumption continues to grow in terms of richness, user interactivity, and scope. This is largely due to unprecedented increases in web activity and the advent of powerful mobile applications that are now commonplace. Web technologies have been undergoing significant upgrades in order to keep up with this rising Internet usage. Since 2007, the World Wide Web Consortium (W3C) has been working on standardizing a major overhaul of the core language of the web that renders and displays all web content. This is known as the 5th revision of Hyper Text Markup Language, or simply HTML5 [47].

1.1 HTML5

The goal of HTML5 is to greatly improve web support for the latest multimedia while still maintaining readability by humans and machines alike. In essence, HTML5 is re-standardizing the web to meet modern demands and provide a unified experience across all device types. It is a cross-platform language that brings forth much new functionality, removes unnecessary features, and officially standardizes many already widely supported web technologies and techniques [44].

New semantic tags such as `<article>` and `<header>` improve the markup for modern websites. The introduction of new `<audio>` and `<video>` elements allow for the direct embedding of audio and video content into web pages without the use of additional plug-ins [47]. Additionally, the `<canvas>` element and WebGL support allows for 2D/3D graphics to be incorporated directly into the browser, which can be used for gaming. There is a wide variety of new features being standardized under the umbrella term of HTML5, including those that address the further enhancement of web applications.

The browser is the vehicle through which all web content is expressed and is becoming more powerful and capable as HTML5 support grows. Web browsing is beginning to shift away from pre-rendered content and into an era of dynamically updated, real-time content through web applications. Extensive support for JavaScript (JS) and application programming interfaces (API) allow web browsers to become full-fledged application platforms. The promise and vision is that web applications will be able to serve as complex and engaging alternatives to applications written in native code. Since browsers are ubiquitous across a wide variety of devices and platforms, the goal of “Write-Once-Run-Many” (WORM) becomes highly appealing for application developers.

From the mobile perspective, this is a truly exciting prospect. Native applications have long been dominant and currently serve as the “de facto” standard for mobile users [11]. However, with the advent of HTML5 features such as offline data storage [25] and access to device hardware like the camera [5] and GPS [4], that has begun to change. Mobile analysts are predicting [42] that HTML5-based web applications will eventually replace native ones after HTML5 matures and becomes widely adopted. The idea is that web applications will be eventually be able to match native applications in terms of performance, user experience, richness, and interactivity – all while running within the browser. This shifts the focus away from supporting many platforms and operating systems, which has become a costly and difficult issue for many developers [44].

A September 2012 Kendo UI survey [50] of over 4,000 developers revealed that 63% are actively developing with HTML5, while another 31% have plans to start using HTML5. This suggests an overwhelming interest in HTML5 development and serves as a powerful harbinger. Since the web is an open platform accessible to all, this is a disruptive and radical shift from current business models where mobile content distribution is largely controlled by a few companies. To ease this transition, hybrid applications have been developed that serve as an intermediary between web and native apps. A hybrid application is essentially an HTML5-based application that is wrapped in native code in order to be sold and installed on devices via application stores. For now, it appears that they will serve as examples of what future web applications will be able to provide in terms of capabilities and user experience.

1.2 Communication Technologies

All web, hybrid, and native applications that use Internet connectivity require a means to communicate and transfer data between hosts. Two powerful communication technologies that are part of the HTML5 Web Apps Working Group are WebSockets [28] and Server-Sent Events [27]. Both of these protocols allow for the delivery of data in real-time, either bi- or uni-directionally, with minimal framing or overhead. They have both been designed for maximum efficiency, high scalability, and to be compatible with current network infrastructures. The latest versions of Chrome, Firefox, Opera, and Internet Explorer browsers mostly support WebSockets and Server-Sent Events, including their mobile variants [34].

These protocols will be especially useful for “always-on” applications, which require a constant Internet connection in order to function properly. Push email, instant messaging, social media notifications, and Voice-over IP (VoIP) telephony are examples of these applications and they have become increasingly popular in mobile devices. Maintaining a persistent connection usually requires a mechanism to keep the connection alive to avoid timeouts, such as the sending of occasional heartbeat messages between the server and mobile device. In WebSockets and Server-Sent Events, these keep-alive mechanisms have not been formally specified or exposed for use in their APIs [26, 27], even though they are necessary to employ in most situations for long-term connectivity. Therefore, application-level techniques must be employed to test these keep-alive mechanisms for effectiveness and performance.

Mobile phones and tablets have become increasingly powerful computing devices, featuring significant processing power, storage, and high-speed transfer capabilities. However, the main downfall for all mobile devices is that they are battery constrained and often do not last longer than 24 hours without needing to be charged again. Therefore, energy consumption and energy efficiency are critical factors when it comes to all mobile communications. Since it is highly likely that WebSockets and Server-Sent Events will serve as the transport mechanisms for future real-time applications, their mobile usage must be considered and examined thoroughly to determine their overall level of implementability.

1.3 Problem statement

The fundamental goal of this thesis is to understand and measure the current level of performance and energy consumption of WebSocket and Server-Sent Events connections on modern mobile devices. To achieve relevant results,

a comprehensive set of tests was conducted on the three most popular mobile platforms: iOS, Android, and Windows Phone. Furthermore, the use of these connections was tested using several of the latest mobile browsers including Firefox, Chrome, Safari, Internet Explorer 10, Android browser, and Opera Mobile. Testing was performed using 3G UMTS, 4G LTE, and WiFi networks in Espoo, Finland to attain real-world results and identify any potentially significant network differences.

Mobile phones were connected to an external power monitor for precise and accurate energy measurements to gain insights into how keep-alive mechanisms affected energy consumption. Altering and optimizing these heartbeat mechanisms became important to develop ways to maintain the connection for long periods while consuming the least amount of excess energy. Thus, the main emphasis of these experiments was to determine how long a connection could be maintained indefinitely in an optimally energy-efficient manner.

After searching through scholarly and peer-reviewed journals, it appears that no comprehensive mobile WebSocket or Server-Sent Event measurements have been performed. Currently, these protocols are in the Candidate Recommendation phase which seeks to determine their implementability within the developer community. Therefore, it is desired that the results of this thesis will reveal accurately the current performance level of these connections on top mobile devices and browsers. WebSockets and Server-Sent Events are both relatively new and need extensive mobile testing to ensure their effectiveness and to determine areas that need improvement. These protocols are likely to be influential transport mechanisms and their usage on mobile devices is only going to continue to increase in the future.

Therefore, it is believed that the measurements performed in this thesis can be of value to application developers, for the enhancement of browser support, and for manufacturers to gain a better understanding of how their devices perform using new transport protocols.

1.4 Structure of the Thesis

This section describes the outline of the thesis. Chapter 2 examines the principles, practices, and the need for mobile energy efficiency. Chapter 3 explores the current web methods and mechanisms to maintain persistent, real-time communications on the Internet. Chapter 4 introduces WebSockets and Server-Sent Events, highlighting their characteristics, uses, and means of operation. Chapter 5 explains all necessary aspects of the measurements and experiments conducted. Chapter 6 reveals the results for all WebSocket

measurements. Chapter 7 includes the results for all Server-Sent Events measurements. Chapter 8 discusses the overall implications and significant conclusions derived from the results. Chapter 9 concludes the thesis by proposing future work opportunities and potentials for these technologies.

Chapter 2

Mobile Energy Efficiency

Today's world has drastically evolved with the advancement and widespread proliferation of mobile devices. We have to come to rely on mobile devices for many uses in our everyday lives, particularly for communication, business, education, entertainment, and a variety of other important areas. Mobility has added a new dimension to the human experience, particularly with the advent of instant global communications and vast social networking now being commonplace.

It is now expected that traffic from wireless and mobile devices will exceed all wired traffic by 2016 [13]. Cisco estimates a 13-fold increase in mobile traffic over the next 4 years, growing at a rate 3 times faster than fixed IP traffic [13]. The emphasis on mobile video, streaming, real-time applications, and other high-bandwidth media content puts great pressure on the mobile networks and devices to meet these rising usage demands. In this digital age, each new innovation in the mobile and web related fields is sure to produce significant consequences for large numbers of people. Therefore, it is paramount that the field of mobility continue to evolve for the betterment of all, bringing forth ingenuity and a dedication to serving the people responsibly using these powerful technologies.

Mobile technology has seen great improvements in recent years to display technology, powerful parallel processing, high-speed data networks, increased storage and memory, along with slimmer and more efficient chipsets. Form factors continue to evolve, with the focus now on touchscreen devices. The anticipated future standard are nanoparticle screens that can bend, twist, and fold while remaining ultra-crisp and responsive [38]. As these technologies continue to improve, the crucial factor that ensures the success of widespread mobility lies in the energy sources used to power these devices. Energy consumption is one of the most critical factors of mobile evolution, given that it is the foundation of using the technology itself.

The battery of the mobile device is one of the most important factors to consider. Unfortunately, battery technology has not improved dramatically in comparison to other components. High-powered devices connected to high-speed networks tend to consume energy rapidly and usually require charging within 1-2 days. In cases of heavier use, charging can be required multiple times per day. This excessive battery drain is one of the most common complaints when it comes to mobile device usage in general. In fact, this power drain is a bottleneck for the success of many mobile applications. Battery capacity has not been meeting the increased demand for power. Therefore, it is imperative to develop wise and novel solutions when it comes to utilizing battery power and mobile energy sources. This is an active and vital area of research that must continue to deliver new breakthroughs in order to meet the demand of consumers and industries worldwide.

2.1 Addressing The Need

There are multiple ways of addressing this energy problem. First, the need for efficiency and feasibility should be established as the foundation when it comes to all of these approaches. These techniques should emphasize practicality and benefits to the end-users. Let us review the common ways to maximize battery and energy performance.

2.1.1 Common Approaches

One approach is to have users recharge their devices more often, perhaps using novel solutions such as wireless charging. This may temporarily alleviate the problem for some users, but it still does not solve the core issues. Mainly, that the source of energy for these devices is inherently limited and does not provide energy efficiently enough for modern mobile users.

Another approach is to increase the battery power by building larger batteries. This strategy is indeed being employed, with batteries of over 2000 mAh becoming standard for many smartphones. However, larger batteries usually require an increase in overall device size, which some users are not willing to compromise on. Larger batteries may be applicable to the tablet market, but the consumer mobile market generally appreciates thin, lightweight, and portable devices.

The third approach involves developing new battery technologies that can potentially revolutionize the energy consumption within mobile devices. There are two splits in this area: building a more powerful battery from a novel source other than lithium-ion or optimizing current battery technolo-

gies to use energy in a more efficient manner. Research in the latter usually involves nano-technology and has seen some impressive claims from companies such as Eta Devices.

The company, founded by two researchers from MIT, has developed power amplification technology that can potentially double the battery life for mobile devices [36, 48]. They claim this dramatic improvement is due to a 90% reduction in heat waste within the device [16] by dynamically adjusting how much power the phone needs to pull from the battery. Their systems target both operator base station equipment as well as mobile chipsets. This is an exciting development to keep an eye on, but until these technologies reach the consumer level, it is likely that current battery trends will continue.

Finally, the last approach has been to improve the component technologies and chipsets within the mobile devices themselves. Through much optimization and refinement, newer components are now able to spend less energy and achieve the same performance as older models. This has brought increased battery savings for consumers but still has not solved the problem itself. Therefore, until new solutions have been employed that bring significant, tangible results to the mass market, further research and optimization must be conducted to find smarter ways of using battery power.

2.1.2 Target Areas

The main areas of focus include the application level, the network level, and the component level. Various strategies and mechanisms can be implemented at each of these levels in an attempt to optimize energy consumption. All of these areas are inter-related, so bringing an improvement to one area can affect the rest of the system positively. The application level techniques can be tested and implemented fairly easily, with flexibility on the configuration of parameters and the way the application interacts with the network. These can be implemented as middleware solutions on mobile devices, which optimizes the software that connects two otherwise separate applications. Further refinements can also be made at the proxy-level on the network side, or by modifying the server behavior within data centers.

Improvements in these areas have been known to bring high energy savings of several tens of percentages in certain cases [40]. The main disadvantage for application level techniques is that only certain types of applications may be positively influenced, so these improvements remain rather case-specific. Network-level improvements must be handled with care because the networks accommodate a larger variety of communication and computing protocols, not just for mobile. At the component and lower levels, changes and improvements are likely to affect a greater set of users and applications.

However, the improvements in these areas are typically more modest than at the application-level [40].

In the case of new communication protocols, it is important that these be developed with efficiency as a foundation, removing all unnecessary overhead, framing, handshakes, etc. This can greatly reduce bandwidth consumption, improve latency, and eliminate unneeded network activity. Since Web technologies are no longer being used by just PCs, the mobile factor is being considered in standardization efforts by organizations such as W3C.

2.2 Key Principles

Research on mobile energy consumption has been widespread and is an active field today. While theoretical studies can provide useful insights, there are so many complex interactions and influencing factors within mobile operations that experimentation is often needed to gain a concrete understanding of what is occurring and how it can be improved. This is done by making measurements using various devices across several network types for specific purposes. At this time, the main focus of our research is on 3G and 4G LTE networks. The possibilities for gaining a deeper understanding of mobile performance are virtually limitless and should be pursued with great enthusiasm.

2.2.1 Measuring Energy Consumption

To determine how much energy consumption is taking place, several methods are used during experimentation. These include:

1. **Using the battery indicator.** This method is crude and often unreliable in terms of accuracy, failing to provide specific values in terms of power drawn, current, etc.
2. **Using a power metering application.** These applications are installed to the mobile device themselves and report feedback on how much energy is consumed. This method can be highly accurate if it is able to access the low-level hardware of the device. The only notable application in this class is Nokia Energy Profiler, which is supported only by older Symbian phones. Manufacturers and API limitations often restrict low-level access, so most energy metering applications must resort to alternative techniques that are less reliable.
3. **Using model-based power metering applications.** These applications measure several performance metrics such as CPU utilization

or network use to determine how much energy is being consumed. However, these derivations are based on models, which vary in quality and ultimately limit the overall accuracy of these applications in many cases.

4. **Using an external power meter.** This approach involves a physical device that acts as a power source while being able to simultaneously record the amount of spent power. One such device is the Monsoon Power Monitor [37], which was used in all measurements for this thesis. This approach works well for devices with replaceable batteries where access to the power terminals is provided. However, the growing trend towards non-replaceable batteries makes this difficult in certain cases. Some devices, such as the Apple iPhone, must be physically opened, which can be risky and usually voids the warranty.

2.2.2 Energy Spending in Mobile Devices

A mobile device has several key internal components and it is important to identify which are the most costly in terms of energy consumption. It has been identified that network communication accounts for almost 50% of daily energy consumption in GSM networks [10], with the share of CPU energy being less than 20%. This is very revealing and suggests that the radio is the top culprit for excessive energy drain. This area becomes a great focus as most popular mobile applications are heavily communication-based and thus require a significant amount of energy. Therefore, influencing the communication in a way that produces energy savings becomes highly desirable. There are several ways to accomplish this, usually by shaping the network traffic. This is often invisible to the end user, although certain delays and trade-offs in performance can result. Ultimately, the goal is satisfactory performance with a high level of energy efficiency.

2.2.3 Efficient Practices

A key principle in mobile power consumption is that a higher bitrate increases the overall energy efficiency of a data transfer. Measurements have shown [39] that the power consumption of TCP data transfers are almost constant and possess a weak dependency on the bitrate in 3G networks. A mobile device is only able to enter a low-power sleep state when the bitrate has dropped to zero and no communication is taking place. Therefore, in order to maximize battery performance, the bitrate should be as high as possible

during all data transfers. In all other instances, the device should remain in a low-power idle state for as long as possible.

2.2.3.1 Reducing State Changes

Additionally, energy savings can be achieved by reducing the amount of radio state changes the mobile device makes. Each time a device moves from idle to an active state, signaling messages must be exchanged between the device and the network, which consume time and energy and result in energy overhead. The goal is to limit this overhead by reducing and minimizing the amount of unnecessary state changes. While spending time in the idle state is extremely efficient, no data transfer can occur during this time. Therefore, most useful work involving communication will take place in the active state.

Communication will occur by alternating between the two states. While active, data should be transferred using the highest data rate possible in bursts. In between bursts, the device should remain idle. These communication bursts should last long enough to avoid excessive energy overhead, yet not too long as to consume unnecessary amounts of energy. In many cases, by returning to an idle state too quickly, excessive overhead is caused due to the device having to return to an active state too quickly again. Therefore, finding the right balance between these two extremes is a difficult practice.

2.2.3.2 Parallel Transfers Save Energy

Mobile devices are now running multiple services which each require a data connection. These services usually schedule their activities independently of one another and transfer data whenever necessary. Some services, such as email or RSS updates, are periodic in nature, while others require a constant stream of data to function. Regardless, for any active service, including voice calls, the radio must be active. It has been discovered that if the services were to collaborate and schedule their data transfers in a parallel manner, energy savings are achieved. Experiments conducted by Nurminen [39] reveal that waiting to transfer data until a voice call takes place is a particularly efficient practice. Since transferring data for periodic services and voice calls both require radio access, it is beneficial to coordinate these activities in parallel if possible. This is particularly true because voice calls only consume a small amount of bandwidth and the transmission must be continuous. Therefore, using as much of this available bandwidth during each active state is a priority and can be accomplished with parallel data transfers. While this may not be feasible for some users who require instant delivery of emails, etc, this is still an important practice that should be considered and implemented in

applicable situations.

2.3 Radio States for 3G UMTS

It is important to understand the energetic effect that switching between radio states has on the mobile device. For this, a brief review of the Radio Resource Control (RRC) states for 3G Universal Mobile Telecommunications System (UMTS) and 4G Long Term Evolution (LTE) networks is necessary. The RRC protocol is responsible for assigning radio resources between the mobile and the network [22]. Each time the wireless interface is activated and deactivated, time and energy is required. These are called the head and tail energies and they vary depending on the communication medium used. In mobile networks, the tail energy is known to be rather costly. The network operators control these values using timers which vary per operator. In 3G networks, it typically takes between 10-12 seconds following a data transfer to return to the idle state [19], which is much slower than 2nd generation GSM at 6 seconds and WiFi at less than 1 second [7]. Again, this suggests that data should be sent in fewer but longer bursts to maximize energy savings.

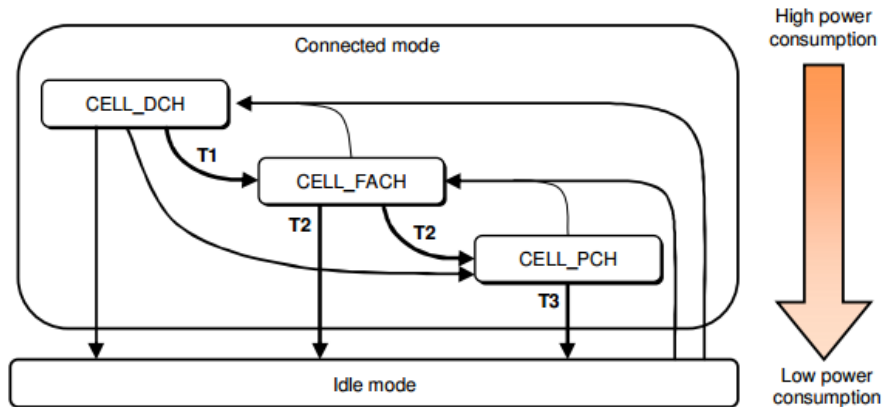


Figure 2.1: Overview of 3G RRC state machine [22]

Figure 2.1 displays the state transition behavior for 3G UMTS networks. These states will each be briefly examined individually.

- **CELL_DCH (Dedicated Channel):** This is the active state where data transfers occur. The device is allocated a dedicated channel to use with maximum throughput and minimum delay. This state is the most costly in terms of power consumption.

- **CELL_FACH (Forward Access Channel):** In this state, the device shares a common channel with other phones, where only small amounts of traffic can be transmitted. The power consumption is approximately 40-50% that of CELL_DCH [19].
- **CELL_PCH (Paging Channel):** This optional state consumes around 1-2% of the energy of CELL_DCH. The control connection is maintained between the device and the network, but packets cannot be sent or received by the mobile. Instead, it can be paged if there are available packets, triggering a switch over to the CELL_FACH or CELL_DCH states temporarily. Since the control connection is maintained, new data transfers can be made faster and with less signaling as only the actual data is sent.
- **IDLE:** In this state, the device does not have an RRC connection and only rarely transmits location area updates. The mobile can still possess an IP address, however, and it can be reached by paging. The battery consumption is similar to the CELL_PCH state and is optimal for periods of no data activity. Any new data transfers will require re-establishing a control connection with the network first before data can be sent again.

Additionally, the URA_PCH state exists in the specification but is not currently deployed. This state is similar to CELL_PCH and gives additional performance when mobility is present [19]. The transition between all states shown above is dependent upon the T1, T2, and T3 timer values as defined by the network operator. For example, following the last packet of a data transfer in DCH state, the T1 timer is activated. This is done in case the device needs to send or receive more data, which is often the case. Once this timer expires, the device will transition into the FACH, PCH (if available), or IDLE state. The T1 and T2 timers are usually range between 2-12 seconds, while the T3 timer usually lasts several minutes [22].

Since switching states requires signaling, involves delay, and produces energy overhead, using very short timer values is not feasible in practice. This can affect the end-user experience negatively, such as slower page response during browsing. If the device were to switch into FACH or IDLE modes too quickly, this would require extra network resources to re-establish the active connection again in DCH mode. Therefore, the trade-off is that the shorter the timers are, the more delay the user experiences.

2.4 Radio States for 4G LTE

The evolution of UMTS technologies into 4G LTE networks has made much progress since 2009 with the standardization of the 3rd Generation Partnership Project (3GPP) Release 8 [1]. The standard offers high bandwidth, enhanced data rates, new modulation techniques, and a simplified all-IP architecture. However, the complexity of the components circuitry has increased and this results in increased energy drain [8]. Therefore, there is great emphasis on power savings and optimal energy performance for all LTE-enabled devices. In LTE networks, only two radio states exist: `RRC_IDLE` and `RRC_CONNECTED` [1]. In the `IDLE` state, no radio resources are allocated. To improve battery life in the mobile devices, a mechanism called discontinuous reception (DRX) was introduced to work with the network scheduling algorithms.

2.4.1 Discontinuous Reception

The goal of the DRX mechanism is to offer a balance between power savings and quality of service [31]. DRX allows the device to sleep, even while in the `CONNECTED` state, for periods of time in order to conserve energy. During a DRX cycle, the device will monitor the downlink control channel for incoming data for a short period of time known as the “On duration” period [8]. Then it will sleep for an “Off duration” cycle of time. These cycles alternate in an effort to conserve energy. DRX periods have been split into Short DRX and Long DRX intervals. There is a trade-off between energy usage and latency for both modes. Short DRX allows for less sleep and a faster response time. Long DRX sleeps longer and responds slower, but consumes less energy. These parameters greatly affect the overall energy consumption of the mobile device using LTE and vary per network. Ultimately, the DRX mechanism is not actively employed on all LTE networks but this can be individually determined by analyzing power traces for mobile devices.

2.4.2 Fast Dormancy

Fast Dormancy (FD) is a mechanism implemented in certain 3G UMTS networks that support Release 8 [1] and also in specific LTE networks. Rather than waiting for the T1/T2 timers following data transfer, the mobile device is now able to send a Signaling Connection Release Indication (SCRI) packet to the network indicating that it is immediately ready to be transferred to a low-power state [19]. This can be very useful in situations where only a small amount of data is transferred, such as a heartbeat exchange, resulting

in less tail energy and increased battery life. Upon receiving the SCRI, the network will send a response packet and immediately transfer the device into the FACH, PCH, or IDLE state, depending on the configuration. However, it is important to note that both the mobile device and the network must support FD or there will be problems such as excessive battery drain. For example, if the phone supports FD but the network does not, the SCRI will not be properly understood. Thus, the mobile will wait for a response from the network before suspending its data connection, wasting resources unnecessarily in the meantime.

2.5 Always-On Applications

An increasing amount of applications today can be deemed as “always-on” and require the mobile device to constantly maintain a radio connection to receive TCP/IP traffic. These can include push email, social media notifications, instant messaging, and IP voice/video. These applications do not send and receive traffic constantly, but instead require a persistent connection that usually remains idle most of the time. Maintaining these connections requires the use of keep-alive mechanisms, which often involve sending tiny packets to alert the service or device that the connection is still active.

Keep-alive mechanisms can have adverse affects on mobile energy consumption when interacting with the 3G RRC configuration of modern networks. Measurements performed by Haverinen et al. from Nokia [22] reveal that the energy consumed by keep-alive mechanisms can lead to extremely poor battery lifetimes, especially when using IPSec or Mobile IP. For this reason, special configuration of RRC parameters is necessary to meet the rising usage of always-on applications.

2.5.1 RRC Transitions

When a keep-alive message is sent by an always-on application, the transitions between RRC states is dependent upon the network configuration. The device wakes up to transfer and/or receive a heartbeat message before returning to a low idle state. There are three possible scenarios [22] for 3G UMTS networks:

1. If CELL_PCH is supported: the sequence is $CELL_FACH \rightarrow CELL_PCH \rightarrow CELL_FACH \rightarrow CELL_PCH \rightarrow \text{etc.}$
2. If CELL_PCH is not supported, but the connection can be created directly to the CELL_FACH state: the sequence is $CELL_FACH \rightarrow$

$IDLE \rightarrow CELL_FACH \rightarrow IDLE \rightarrow etc.$

3. If the previous cases are not supported, the sequence is $CELL_DCH \rightarrow CELL_FACH \rightarrow IDLE \rightarrow CELL_DCH \rightarrow CELL_FACH \rightarrow IDLE \rightarrow etc.$

The most efficient transitions take place in the PCH \rightarrow FACH exchange, followed by IDLE \rightarrow FACH [22].

2.5.2 Stateful Middleboxes

The reason that keep-alive mechanisms must be employed is usually to avoid timeouts for long-held connections with little to no activity. In today's mobile networks, there are firewalls, proxies, and Network Address Translators (NAT) that serve as stateful "middleboxes" between the mobile device and the Internet. While their presence is not formally acknowledged in 3GPP architectures, they are present and have a great effect on the energy performance for mobiles using always-on applications. When packets are sent through a middlebox, a state is created for that connection. After a certain period of inactivity, the state is removed and the connection is released in an effort to save network resources. This requires the mobile device to send keep-alive messages before reaching timeout values as defined by these middleboxes. The difficulty is that these timeout values vary widely, depending on the manufacturer and the operator configuration.

2.5.2.1 Timeout Values

For NATs and firewalls, the timeout values usually vary depending on TCP or UDP connections. Typically, TCP timeout values for several popular firewalls and NATs (Cisco, Juniper, Nokia, etc.) range from 30-150 minutes [22]. For UDP, this usually ranges from 40-300 seconds [22], which is significantly shorter than TCP connections. However, in some cases it has been shown that these TCP timeouts can be much shorter. Rob Mueller, administrator of the FastMail.FM webmail service, discovered that his users were unable to maintain a persistent HTTP connection for 5 minutes. By testing using 4 desktop browsers, he determined that each could not exceed a persistent TCP-based HTTP connection for more than 2 minutes [46]. His results strongly suggest that there are TCP timeout values of 120 seconds or less within some stateful NATs, firewalls, and/or proxies.

To compensate for this, Mueller discovered that browsers such as Opera, Firefox, and Internet Explorer 9 (IE9) will close idle connections before reaching 120 seconds. For Chrome, TCP keep-alive packets are sent every 45 sec-

onds to ensure that the connection stays active. These low timeout values have rather disturbing implications for connections that require a long-lived connection, such as Server-Sent Events, as will be shown in Chapter 7. Since it is not feasible to bypass the middleboxes, strategies to compensate for their presence must be made. This means configuring RRC parameters with properly balanced timer values, as well as supporting the PCH state or IDLE \rightarrow FACH transitions.

Regardless of these optimal configurations, it appears that keep-alive messages may have to be sent within the threshold of these 2 minute timeout intervals in order to avoid premature connection terminations. However, this is contrary to the results of the Nokia measurements, which state that keep-alive intervals of less than 2 minutes will not yield acceptable battery performance [22]. Clearly there is a discrepancy here and with the rise of always-on applications, it will be important to study this issue carefully to accurately gauge if this level of mobile performance is energetically feasible.

Let us now examine the nature of persistent communications and the modern techniques used for real-time applications.

Chapter 3

Persistent Communication

Communication requiring persistent connections have become a popular standard across a wide variety of web applications and platforms. The goal is to update content dynamically without having to refresh the page or application, which simulates the appearance of providing real-time communication. Some of these services include chat applications, instant messaging, social media updates (Twitter, Facebook, etc), live stock exchanges, gambling services, sports score updates, weather information, and many other uses. The applications for delivering real-time content are expanding as web browsers become more versatile with increased HTML5 support. As the use of these services grows, the demand for instant delivery of digital content becomes more widespread. Thus, the main function of this real-time paradigm is to deliver content to a client or end-user as soon as new information is available, in the speediest manner possible.

There are three significant real-time protocols that are currently being standardized under the umbrella of HTML5: WebSocket, Server-Sent Events, and WebRTC. It will be interesting to see how the usage of these protocols evolves within mobile devices. With the advent of WebRTC [2], it is now possible to make voice calls, video calls, and share files from browser to browser without the need for additional plug-ins. However, this protocol has not reached maturity or extensive use yet. The other two protocols are very useful for uni- and bi-directional communication that are both highly scalable and simple to implement within the browser. In this chapter, the traditional web methods used to maintain persistent connections and achieve real-time communication are examined.

3.1 Persistent HTTP Connections

The vast majority of all web applications and services are still using HTTP. Since HTTP is a half-duplex protocol, communication can only occur in one direction at a time: starting from either the client or the server. It is based on the *request-response* paradigm, which has never been ideal for full-duplex or streaming communications. Specifically in HTTP 1.0, every time a client wishes to retrieve a webpage or specific content, it makes a new request to the server. After receiving the complete request, the server then sends the content in a formal response and closes the connection. This works well for static pages where content does not need to be updated, but not for dynamically updated content, such as stocks or sports scores, where values change frequently. For this reason, the use of persistent connections was standardized in section 8 of HTTP 1.1 [45], where the connection is kept open to allow multiple requests/responses to quickly load all necessary objects. It is now the default behavior that all HTTP connections remain open, unless otherwise indicated. The decision to keep a connection persistently open is found in all HTTP headers under:

`Connection: keep-alive`

3.1.1 Advantages

There are several advantages to maintaining persistent HTTP connections [45]. First, because fewer TCP connections are opened/closed, CPU time and memory usage is reduced in both routers and hosts. This includes the computation required within clients, servers, proxies, gateways, caches, and tunnels, which can be quite substantial. Second, HTTP requests can be pipelined on a single TCP connection. This allows a client to make multiple requests without waiting for each response, resulting in much more efficient use of the connection. Third, network congestion and latency is reduced by avoiding opening unnecessary TCP connections and eliminating extra handshakes. Finally, keeping a connection alive and pipelining can reduce a user's waiting time for loading pages and objects.

3.1.2 Disadvantages

The HTTP 1.1 specification states that clients should not open more than two simultaneous connections with a web server [45]. This is likely because servers support only a finite number of concurrent connections. Therefore, servers can become overburdened by clients who are keeping their connections open

for long periods without actually needing any additional data. This results in poor server performance and potentially denies new incoming connections when full capacity is reached. Wasteful performance is especially true for services that only provide single objects, such as image providers. The result is that persistent connections must be managed and configured effectively in order to maintain optimal server performance.

3.2 Keep-Alive Mechanisms

Web servers now set a specific *keep-alive timeout* value in order to manage persistent connections. This value usually ranges between 5-120 seconds by default for most HTTP servers. If a client does not send any additional requests within the timeout period, the server will close the connection. Both the client and server have the ability to initiate a graceful closing of the connection at any time, and each are constantly listening for this. If the graceful closing is not detected by the other party, this can cause an unnecessary drain of resources on the network [45]. Additionally, a server can specify how many total requests can be made over a single persistent connection. This can be useful to ensure that clients do not stay infinitely connected. For services that wish to keep a connection persistently open for long periods of time, packets must be repeatedly sent within the timeout interval. These keep-alive packets are known as *heartbeats*, because they let the server or client know that the connection is still active and should be kept open.

It is important to realize that client browsers and web servers are not the only hosts that possess keep-alive timeout values. Stateful middleboxes also play significant role in whether the connection is maintained or not. The default timeout value for many NATs and firewalls is 120 seconds, meaning all packets received after 2 minutes of inactivity will be dropped. When this occurs, a RST packet is *not* usually sent to the originating host. This causes the TCP retransmission timer to expire, eventually informing the application that the connection has timed out. Both host end-points consume unnecessary network resources while they believe the connection is still active, which is wasteful and unnecessary.

3.2.1 PING/PONG

Traditionally, a keep-alive protocol involves the exchange of PING/PONG messages between a client and server. This allows the server to maintain an awareness of which clients are active or inactive. Similarly, a client may want to know if a server is still active or not. Therefore, both the client or server

can initiate a heartbeat exchange, depending on the scheme used. In general, a peer sends a PING message, which is replied to with a PONG message. If the PONG message reply is successfully received, the peer knows the other party is still active. Otherwise, the connection can be assumed inactive and closed. Neither of these messages contain a payload and they are small in size. Once a heartbeat exchange is completed, the timeout timer begins again for another cycle. Typically, configuring the interval and frequency of keep-alive messages is done at the application level. However, this has also been implemented closer to the transport layer for use with TCP.

3.2.2 TCP Keep-Alive

During an idle TCP connection, no data is flowing between the two hosts. This means that once a TCP connection has been established, it can be left inactive for prolonged periods exceeding many hours or even days. The connection will remain as long as each end-host does not disconnect or restart, and that there are no application-level heartbeat schemes being employed. However, there are times when a server wishes to detect if a client is still connected or has crashed. This is where the TCP keep-alive functionality comes into effect. The keep-alive feature is intended to be used by server applications that consume resources on behalf of a client, to determine if a client really needs those resources allocated [52]. It can also be used to prevent disconnection due to network inactivity. The default keep-alive timer for TCP is 2 hours and can be modified [17]. Once the timer expires, the server sends a keep-alive probe to the client with no data and the ACK flag activated [17]. Should the client not respond with an ACK, it can be assumed that the connection is no longer in use and can be terminated.

However, there are several reasons why using TCP keep-alives is discouraged. Technically, they are not part of the TCP specification. There are three reasons listed in the Host Requirements RFC for this [52]:

1. They can cause perfectly good connections to be dropped during transient failures.
2. They consume unnecessary bandwidth.
3. They are costly on an internet that charges for packet use.

3.3 Web Techniques

Because HTTP was not designed for real-time communication (RTC), several workarounds and novel techniques were developed to overcome the “page-by-page” model of the web. Simulating full-duplex communication over a half-duplex protocol usually requires maintaining two connections for both the upstream and downstream. These approaches are still in use today by the majority of web services and will not be discontinued anytime soon. These techniques are known as polling, long-polling, and streaming, and they are based on the use of Asynchronous Javascript and XML (AJAX) and Comet [3] technologies.

3.3.1 Push vs. Pull

In all web operations, there are essentially two modes of operation: pulling data and pushing data. Pull is based on the *request/response* paradigm, where the client browser “pulls” content from the server with each request, and is synchronous in nature. During the client request, the active thread of execution is usually frozen while waiting for the server response. On the other hand, pushing content is based on the *publish/subscribe* paradigm, where a client subscribes to a certain channel of information, and content is “pushed” to the client as updates become available. This type of communication is asynchronous, meaning it can happen at any time, and usually in the background. The client’s execution is never blocked and the server can send data and events at its own discretion.

3.3.2 AJAX

AJAX is an umbrella term of technologies coined by Jesse James Garrett in 2005 [3]. Using AJAX technologies allows for web applications to pull data from a server asynchronously without altering or refreshing the page contents. This is used to simulate the appearance of receiving real-time data, where, in fact, repeated requests for updates are being exchanged with the server silently in the background. AJAX techniques are browser and platform independent, and are used in popular services such as Google Maps, Google Suggest, GMail, YouTube, and Facebook [53]. These applications are designed to have high user interactivity with low user-perceived latencies [9].

AJAX is a series of techniques based on widely deployed internet standards [53]. Its main components include:

- **XMLHttpRequest (XHR) Objects:** used for asynchronous data exchange with the server.
- **Javascript/Document Object Model (DOM):** used to dynamically display and interact with the information.
- **HTML/CSS:** used to style and format the data.
- **XML/JSON:** the format used for transferring the data itself (the payload).

Let us briefly describe the AJAX process used to dynamically update a webpage in a browser without the user having to refresh.

1. **Browser-side:** an event occurs such as a click, triggering the creation of an XMLHttpRequest (XHR) object. This object is sent to the server.
2. **Server-side:** after receiving and processing the XHR object, the requested data is sent in a formal response back to the browser.
3. **Browser-side:** the data is processed using JavaScript through a callback function and the page content is updated accordingly with the new data.

Due to the asynchronous nature of AJAX, the client-side JavaScript does not have to wait for pending server responses and can execute other scripts while waiting for the response. This is a very welcome improvement for web developers, especially considering that many tasks on the server can be very time consuming. However, there are a few significant disadvantages to using AJAX programming.

The main disadvantage is the complexity required to maintain working, scalable asynchronous code. This is mainly due to the inherent limitations of the request/response paradigm that AJAX must depend on [9]. For each request, a callback function must be designated to handle the response. When there are multiple conditional statements within callback functions, extra careful attention must be paid to ensure proper control flow is maintained. If a single improper *return* statement within an entirely asynchronous callback function is found, the entire dynamic functionality of the page will stop working without warning [51]. The more complex these functions are, the harder they are to maintain, debug, and scale outward [51]. Another disadvantage is found in pre-HTML5 browsers, where dynamic page updates are not registered correctly within the browser history. Here, all dynamically updated AJAX calls are ignored when the user presses the Back button, navigating one full page away rather than an update away.

3.3.2.1 Polling

Polling is one of the most common techniques used by web browsers to deliver real-time information [41]. Essentially, the browser repeatedly sends HTTP GET requests to the server at regular intervals (i.e. 30 seconds) and receives a response immediately. If the exact interval of updated information availability is known, then this technique works perfectly. However, real-time data is often unpredictable and repeated polling frequently results in unnecessary requests and needless connections being opened and closed. Even when new data is not available, the server will send an empty response. The result of polling is usually a large amount of extra HTTP overhead, which, over time and with increasing clients, leads to decreased overall network throughput [41]. The total overhead from the HTTP request and response header is at least 871 bytes without containing any data [41]. For small data payloads (i.e. 20 bytes), this is enormously wasteful. Therefore, more effective alternatives were developed.

3.3.2.2 Comet/Long-polling

Comet-based technologies have been used since 2000 and were made popular in 2006 by Alex Russell [3]. Comet is a family of web techniques that allows the server to hold an HTTP request open for prolonged periods of time to push data to the browser. This is accomplished without the browser having to make new requests, so data can be received in real-time. Comet has also been called Reverse AJAX or HTTP server push and have been noted to possess advantages over exclusively pull-based approaches [9]. Comet uses long-polling with AJAX as one of its main techniques. All browsers that support XHR objects support this feature.

Long-polling is similar to polling, except that the server keeps the HTTP request open if data is not immediately available. The server determines how long to keep the request open, also known as a “hanging GET”. If new data is received within the time interval, a response containing the data is sent to the client and the connection is closed. If new data is not received within the time period, the server will respond with a notification to terminate the open request and close the connection. After the client browser receives the response, it will create another XHR object request to handle the next event, therefore always keeping a new long-polling request open for new events. This results in the server constantly responding with new data as soon as it is made available. However, in situations with high-message volume, long-polling does not provide increased performance benefits over regular polling [41]. Performance could actually be decreased if long-polling requests turn

into continuous, unthrottled loops of regular polling requests [41].

3.3.2.3 Comet/HTTP Streaming

Streaming involves setting up a persistent HTTP connection where an open response is continuously updated and kept open indefinitely by the server. The server will continuously send new data as it receives it, but never complete the HTTP response fully, which allows the browser to keep waiting for more information. The two main techniques on how this is accomplished include using hidden iframe elements and XHR objects as part of multipart-responses.

An iframe (inline frame) allows HTML documents to be embedded inside one another and to load content asynchronously. To facilitate Comet-based streaming, an invisible, infinitely-long iframe is used which accumulates returned JavaScript code from the server as events occur. However, iframes were not originally designed for streaming, so this technique is commonly regarded as a “hack” and becomes increasingly complex and difficult to scale out for larger applications.

Additionally, because streaming is done over HTTP, firewalls and proxy servers may choose to buffer server responses, which increases the latency of message delivery [41]. It is for this reason that many Comet-based streaming solutions fall back to long-polling whenever encountering a buffering proxy server [41]. However, this can be circumvented by using secure TLS/SSL for the streaming connection, which will incur its own additional costs during setup and teardown.

3.3.3 Major Issues

Ultimately, HTTP was not designed for real-time communication. Simulating full-duplex communication over a half-duplex protocol requires ingenuity and greater complexity than a protocol designed specifically for RTC. While AJAX/Comet-based techniques achieve real-time functionality and are widely deployed, there are some major drawbacks that must be considered, namely cost and complexity.

The most noticeable downside is the huge amount of unnecessary HTTP overhead that is found in each request/response message during these polling and long-polling exchanges. This overhead increases message latency and, if eliminated on a wide scale, could reduce bandwidth consumption significantly [24, 41]. Additionally, many web applications use two connections to maintain a RTC experience: one for the downstream and one for the upstream. Maintaining and coordinating these connections is costly, prone to errors,

and complex for both the server and client, especially for large scale applications that use multiple proxies and back-end data sources.

Therefore, it was deemed necessary to create new protocols that were innately designed for scalable, widely-supported, and efficient real-time communication. These protocols include the WebSocket and Server-Sent Events and are covered in the next chapter.

Chapter 4

HTML5 Protocols

The advent of new communication protocols under HTML5 provides exciting implications for web users and developers. Web browsers are continuing to evolve in their technological capabilities and this is affecting how users interact with web applications. This is particularly true with the ever-increasing rise of mobile and tablet users. The increased demand for a rich, satisfying, and content-driven web experience is strong and this is pushing developers and standards committees to ensure that the next-generation of communication protocols are ready to handle this demand. As web applications continue to become more interactive and offer feature performance similar to that of native applications, we will see these next-generation protocols becoming used more frequently in both desktop and mobile devices.

While there are several new protocols within the family of HTML5, we will focus on two of them. These protocols are the WebSocket (WS) and Server-Sent Events (SSE) and they are both in the Candidate Recommendation phase by the W3C [27, 28]. This means that the developer community is now being called to verify how implementable these protocols are in their current state. It is one of the main goals of this thesis to determine how mobile-ready these protocols are. These exciting technologies have the potential to become popular standards for future web communication, primarily due to the innate efficiency of these protocols. Simply put, both WS and SSE are designed for communication that is event-driven, scalable, simple to implement, and compatible with existing hardware and infrastructure. Both of these protocols have their own uses and applications, which will be expanded upon in this chapter. First, the WS and SSE specifications and functionality will be examined in greater detail.

4.1 WebSocket

The WebSocket protocol was originally created by the Web Hypertext Application Technology Working Group (WHATWG) and included within the original HTML5 specifications [41]. However, to keep standardization efforts focused, the WebSocket specification [24] was submitted to the Internet Engineering Task Force (IETF) in December 2011 for further development.

WebSockets are designed to facilitate real-time web communication in a genuinely efficient manner. This web technology allows for two-way, full-duplex communications to occur over a single TCP connection between remote hosts. This provides a standard that will allow for scalable, real-time web applications to become seamless and widespread. WebSockets are native to the browser and can be used in conjunction with HTTP, resulting in more simplicity compared to many of the complicated polling-based workarounds.

Simply put, WebSockets were designed for the purpose of two-way real-time interaction over a single socket, bringing forth a powerful new standard that will serve Web users for the years to come. The WebSocket API is supported by the latest major browsers for both desktop and mobile devices, including Chrome, Firefox, Opera, and Internet Explorer [34]. The API [26] is currently in the “Candidate Recommendation” stage of standardization [28] by the W3C, and will eventually become an official W3C Recommendation standard as the technology matures.

4.1.1 Key Characteristics

The WS specification has been created with the intention to replace traditional bidirectional communication techniques over HTTP such as polling. The protocol was designed to work well with the existing web infrastructure types such as proxies, firewalls, filtering, and authentication [24]. This means that using polling methods for real-time applications will be reduced and phased out over time as the WS grows in popularity.

In order to support full backwards compatibility, the WS supports HTTP as a transport protocol over ports 80 and 443, but is not limited strictly to HTTP [24]. The creators of the specification understood that not all applications and services will use HTTP or standard ports, so future implementations allow for flexibility in this regard.

Once the WS connection has been established, both end hosts can communicate in either direction in an unsolicited manner. Therefore, servers now need only to maintain a single TCP connection for exchanging real-time data and at a fraction of the overhead cost. WebSocket data transfers send only

the payload encapsulated with 2 additional bytes [24], which is a massive reduction compared to always sending HTTP message headers of at least 871 bytes [41]. This reduction in the number of connections and overhead will greatly reduce the amount of server load and overall complexity of handling real-time web applications.

WebSocket communication can reach all the way to the back-end due to the ability to detect proxy servers and firewalls easily, making streaming possible over any connection [24]. This has been known to be a problem area for many applications employing bidirectional communication means that maintain two connections for the upstream and downstream.

A WS can detect the presence of a proxy server and create a traversable tunnel through the proxy [41]. This is done by sending an HTTP CONNECT request to the proxy instructing it to open a TCP connection to the desired final destination. After the tunnel is set up, communication to the back-end flows seamlessly. Because HTTPS transmissions also work in a similar manner, WebSocket fully supports secure transmissions over SSL [24].

4.1.2 Design Principles

The WS is an independent TCP-based protocol designed on the principle that there should be minimal framing during data transfer [24]. The only framing that should be present is to distinguish between Unicode text and binary frames, which contains only 2 bytes of data [24]. This implies that all metadata is layered on top of WebSocket via application layer protocols. Conceptually speaking, WebSocket is a layer on top of TCP is built for modern web usage. The protocol does the following [24]:

- Adds a web origin-based security model for browsers.
- Adds an addressing/naming protocol mechanism allowing multiple services on one port and multiple hostnames on one IP address.
- Creates a framing mechanism layered on top of TCP to utilize IP packet functionality without length limits.
- Creates an additional closing handshake designed for use with proxies and other intermediaries.

The WS is essentially designed to be as close to raw TCP as possible, except with the restraints of Web functionality taken into account. This allows for WS servers to share ports with HTTP servers, making the WS highly deployable and scalable.

4.1.3 Connection Setup

There are two main parts of the protocol: the handshake and the data transfer. First, the handshake used to setup the connection will be examined. WebSocket connections are usually first negotiated using HTTP. This was done to ensure backwards compatibility with older clients. Initially, the browser will send a GET request with an offer to Upgrade the connection. This HTTP Connection Upgrade request is found in the HTTP header and specifies the usage of WebSocket, usually over port 80 or 443. The actual handshake takes place when the protocol switch from HTTP to WebSocket is made.

4.1.3.1 Client Handshake

The client handshake from the perspective of a Nokia Lumia 820 is shown below. Note the **Origin** header field is used to protect against unauthorized cross-origin use of a WebSocket server by malicious scripts using the WS API within a browser [24]. The server is able to selectively decide which origins to accept for WS connections.

```
Origin: http://estepel-web.cs.hut.fi
Sec-WebSocket-Key: 4nI2Q8cWcxVNN248rF7ufA==
Connection: Upgrade
Upgrade: WebSocket
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0;
Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 820)
Host: estepel-web.cs.hut.fi
DNT: 1
Cache-Control: no-cache
```

4.1.3.2 Server Handshake

If the server supports WS and validates the origin, it agrees to the protocol switch and the upgrade is made. Any HTTP status code other than 101 indicates that only HTTP will continue to be used. The server takes in the **Sec-WebSocket-Key** value and concatenates it with the Globally Unique Identifier (GUID) in string form [24]. This value is then hashed with SHA-1, further encoded into Base 64 format, and returned in the server handshake response under the field **Sec-WebSocket-Accept**. Following the server response, the HTTP connection then breaks down and is replaced with a WS

connection over the same TCP/IP connection. The server handshake is as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s8bl700wN8dhEaRjJ4/2wFwEuLs=
```

4.1.4 Data Transfer

Once both the client and server have successfully exchanged handshakes, data transfer is ready to begin. Each end host can communicate at their own discretion at any time, without needing to alert the other party. This is the essence of real bidirectional communication. Text and binary frames can be exchanged as data, with these being minimally framed with just 2 bytes [24]. The type of payload data is determined by the *op-code* that accompanies each frame. If an unknown op-code is received, the receiving endpoint must drop the connection [24]. For text frames, each frame begins with a 0x00 byte and ends with a 0xFF byte and contains UTF-8 data in between [41]. Text frames use a terminator to indicate the end of the payload, while binary frames use a length prefix [41]. The entire payload consists of the concatenation of the extension data and application data, shown below in Figure 4.1.

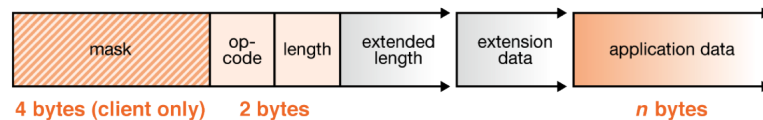


Figure 4.1: A WebSocket frame. [41]

4.1.4.1 Control Frames

In addition to the data frames mentioned above, control frames play an important role in maintaining the connection by communicating state information about the WS. The three control frames that are currently defined include: closing the connection, ping, and pong. Each of these are assigned a unique op-code. Closing the connection is rather simple and can be initiated by either endpoint. A closing handshake process begins which leads to the closing of the underlying TCP connection. If the TCP connection was closed after the WS closing handshake completed successfully, this signifies a clean closing.

4.1.4.2 Keep-Alive Methods

To maintain the WS connection for longer periods of time, it may be necessary to use keep-alive mechanisms such as ping and pong packets. The specification states that these packets can be used as a keep-alive, to verify if a remote endpoint is still active [24], or for network status probing. However, access to ping and pong frames has not been exposed in the current API [26], therefore these techniques must be employed at the application level in order to utilize their functionality.

Either endpoint can send a ping frame. These ping frames can be sent anytime a connection has been established and remains active. After receiving a ping frame, a pong message must be returned by the receiving endpoint as soon as possible. A pong frame will contain identical application data to the ping frame it is responding to. Additionally, if an endpoint has received multiple ping requests, it can choose to only send a pong to the most recently processed ping frame [24]. A pong frame can be sent by an endpoint unsolicited, which serves as a unidirectional heartbeat mechanism.

These ping/pong messages are exchanged directly between the browser and a server, not the application. They are not currently exposed for use in the WS API [28]. Since they are very small in size, approximately 2 bytes long, there is little overhead when exchanging these messages. The purpose of these heartbeat exchanges is to avoid closing the channel due to TCP timeout and to detect when the channel has closed without warning. However, in situations where middle-box timeout values may prematurely close active connections, it may be necessary to implement application-level heartbeat mechanisms to maintain the WS connection.

4.1.5 Using the WebSocket API

The WebSocket API has been designed to be simple to implement and use. A WS connection is established using the two following Uniform Resource Identifier (URI) syntax:

- ws:// for normal communication.
- wss:// for secure communication.

4.1.5.1 Creating a Connection

The creation of a new WS connection requires only specifying a URL resource that represents the destination endpoint:

```
var myWebSocket = new WebSocket("ws://examplehost.org");
```

4.1.5.2 Handling Events

The WS connection is said to exist in three states: `CONNECTING`, `OPEN`, `CLOSED` [28]. These states reflect the current status of the connection. Since JavaScript is event-driven, the WS API has defined four main event handlers that must be defined:

- **`onopen`** - occurs when a socket connection is established.
- **`onmessage`** - occurs when the client receives data from the server.
- **`onerror`** - occurs following an error in communication.
- **`onclose`** - occurs when the connection is closed.

Customizing these event handlers gives developers great flexibility in using WebSocket connections for a variety of application purposes.

4.1.5.3 Sending Data

Once the connection has been established, data can be exchanged using the **`send(data)`** method. Text frames are used for **`string`** objects, while binary frames are used for **`Blob`**, **`ArrayBuffer`**, and **`ArrayBufferView`** objects [28]. A connection can be closed by calling the **`close()`** method. The WS API is essentially simple to implement within the browser. However, servers do require WS-specific support which goes beyond just HTTP support only.

4.1.6 WebSocket Applications

A thorough review [21] of peer-documented research shows that several new applications have been demonstrated that make use of WebSockets. For example, Chen and Zu [12] have shown that online multiplayer games are possible by using WS in conjunction with other HTML5 technologies. Additionally, WebSockets have been featured in several prototype systems, including for retrieving real-time recommendations [49] and for passing messages over a real-time coordination platform. Lin et al. have shown [32] that accessing different IP-based multimedia services is possible using WS. In another scenario, Heinrich and Gaedke used WS to bind data objects to user interface elements with real-time updates [23]. Finally, progress in remote data visualization applications have been made [15, 54] due to lower latencies during transfer. It is clear that the benefits of WS can be applied to a wide-range of systems and applications, bringing improvements to many areas that require efficient communication.

4.1.7 Application Potential

Any application that needs to achieve real-time functionality with bidirectional communication is a prime candidate for WebSockets. These types of applications can include: online gaming, stock tickers, chat/messaging, social media, multi-user applications with simultaneous editing, and many more possibilities. At the time of this writing, WS browser support is widespread and increasing. However, back-end support is still limited. This has resulted in many applications still using traditional HTTP for bidirectional communication. Developers are now starting to see the potential of the browser as a real application interface, rather than just for displaying content. WebSockets are largely responsible for that and they will be increasingly implemented in new application types in the future.

4.1.8 Mobile Usage

Currently several mobile browsers support WebSockets, including Mozilla Firefox, Google Chrome, Microsoft Internet Explorer, and Opera Mobile [34]. Due to the lack of popular mobile native and web applications that utilize WebSockets, this usage is not widespread yet. It is important that mobile browsers are equipped to handle WS connections efficiently and properly. This is particularly true for mobile devices, as they seek to spend as much time as possible in low-power states to maximize battery life. Ensuring correct performance of the mobile-based WS is a priority for browser developers and phone manufacturers so that there is not a large performance gap between desktop and mobile devices.

4.2 Server-Sent Events

Another powerful HTML5 communication technology is Server-Sent Events (SSE). Originally developed by Opera in 2006 [6], this protocol is suited for unidirectional communication from a server to a client. The W3C defines SSE as “an API for opening an HTTP connection for receiving push notifications from a server in the form of DOM events [27].” The W3C further states that SSEs are designed to be extended to work with other push notification schemes such as Push Short Message Service (SMS), Open Mobile Alliance (OMA), etc [27]. The specification is in the Candidate Recommendation stage and is currently awaiting feedback from developers and the web community at large to deem how implementable they are.

Contrary to WebSocket’s bidirectional capabilities, SSEs are focused on

delivering one-way real-time communication to the browser. They are not competing technologies, but rather should be used for the purposes and applications that suit each protocol. SSEs are essentially standardizing the Comet-based methods of delivering push notifications so that this functionality becomes native to the browser. SSEs are similar to HTTP streaming, in which a single connection is kept open for an indefinite period of time to deliver notifications in real-time. SSEs are much simpler to implement and do not require any complicated workarounds to achieve persistent unidirectional communication.

4.2.1 Key Characteristics

Server-Sent Events utilize a persistent connection between a client and a server, which is initiated using HTTP. By operating over standard HTTP connections, web application server support becomes widespread and issues such as port conflicts are not a worry for developers. SSEs are widely supported by the latest desktop and mobile browsers, except for IE10 mobile [34]. It is likely that future versions of mobile Internet Explorer will support SSEs.

SSEs include a new HTML element called **EventSource**, which is the client-side object used to receive events from the server [27]. Additionally, a new Multipurpose Internet Mail Extension (MIME) type called **text/event-stream** is used to define an event framing format [18].

A *subscribe-publish* model is the mode of operation used, where clients subscribe to a channel and receive streaming updates from the server. Using this model, a client subscribes to an event-stream offered by the server. Updates for this event-stream are pushed to the client in real-time as they occur, without the client having to initiate any requests. Since an event-stream may have many clients subscribed to it, the server is easily able to push messages to a large amount of clients easily and in a scalable fashion.

In the event of an error occurring or the connection being dropped, the browser will automatically try reconnect to the server. By default, this occurs every 3 seconds after the disruption begins [27]. This does not need to be configured on the client side, although it is possible to change the reconnection interval. Should a disconnection occur, the client is able to receive all missed events after reconnecting by leveraging the **Last-Event-ID** header. When the event-stream is re-established, the server will look for this header and re-send all missed events for that client if necessary.

4.2.2 Design Principles

Server-Sent Events is not a protocol specifically. It is a JavaScript API that is being standardized for widespread support within all HTML5-compliant browsers. The `EventSource` object and `text/event-stream` MIME types are the core of Server-Sent Events technology. The goal is native, cross-browser streaming that is both real-time and highly scalable. SSEs support HTTP and is compatible with other dedicated server-push protocols as well.

They were designed to simplify and standardize the Comet-based strategy of HTTP streaming for future web applications [18]. They are not a replacement for polling and long-polling techniques, but rather serve as an optimized complement to these technologies. They have been designed to be efficient at the foundational level, avoiding all unnecessary HTTP request overhead during data transfer.

SSEs are a single persistent connection that remain open for prolonged periods of time. They support Document Object Model (DOM) events which can be received in a variety of formats, such as text or JavaScript Object Notation (JSON). Custom events can be created and defined, giving great flexibility to developers seeking to target specific client needs.

One downside to SSEs arises from the browser limiting the number of connections per server. In certain circumstances, multiple pages can be loaded that include an `EventSource` from the same domain, resulting in each `EventSource` object creating a dedicated connection [18]. This is wasteful and the maximum number of connections is often quickly reached. To overcome this per-server connection limitation, a shared `WebWorker` object [29] can be used to distribute a single `EventSource` object for all connections [18]. Additionally, browser-specific implementations of `EventSource` can manage the re-use of connections provided that the absolute URLs are equal to the previous one used.

4.2.3 Using the EventSource API

The creation of an `EventSource` object and registering an event listener is the first step to using SSEs. This is done on the client-side. The server is configured to send data/messages as it pleases, as long as it follows the specifications and proper formatting conventions outlined in the specification [27]. No handshaking protocols are required other than the initial HTTP GET request and the support for event-stream.

4.2.3.1 Client-side

To begin, let us examine the creation of an `EventSource` object and its association with an event-stream. In the constructor, a URL is passed that defines the location of the desired stream. An event handler function must be created to specify how the client reacts to receiving events. The default event type is “message”, with its behavior defined by the `onmessage` handler. Additional event handlers that can be optionally configured include `onopen` and `onerror`. A simple example within an HTML webpage could be configured as follows:

```
source = new EventSource("stream");
source.addEventListener("message", function(event) {
    output.textContent = event.data;
}, false);
};
```

The `EventSource` object opens a URL called “stream” and will update the text content on the page upon each received event. When this object is created, an HTTP GET request is sent to the server, indicating the `text/event-stream` MIME type and to keep the connection alive. The format of this request on an Apple iPhone 4S is as follows:

```
Host: estepe1-web.cs.hut.fi
Accept: text/event-stream
Connection: keep-alive
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 6_1_2 like Mac OS X)
AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10B146
Safari/8536.25
Full request URI: http://estepe1-web.cs.hut.fi/stream
```

4.2.3.2 Server-side

After receiving the request, the server constructs plaintext event responses for that event-stream in a specific format. These events will be interpreted by the client and can be parsed in several ways, including as JSON data. Events consist of one or more comment lines and/or field lines. All field lines will take on the form of `field:value`, where `field` indicates what type of data will be interpreted. The `value` is the actual data for that corresponding field. A space following the colon is acceptable and will be ignored. The following field values are recognized [27]:

- **event:** Describes the name of the event. Example: `event:update`
- **data:** The actual data/message to be parsed and interpreted. Example: `data:This is an event update`
- **id:** This sets the Last-Event-ID for the event being currently processed. Example: `id:461`
- **retry:** Sets the reconnection timer for the event-stream in milliseconds. Example: `retry:5000`

Only comments and field value lines are accepted as valid. To send a comment, the line always begins with a colon (:) value. Events are delimited by blank lines. Two blank lines signals the end of an event and indicates that it is ready for processing by the client. In the following example, the server sets the client reconnection time to 7 seconds. This is followed by a custom event called update, featuring 3 lines of data and setting an event ID. The form could look like this:

```
retry: 7000
event: update
data: 63
data: This event is broken up
data: into three lines.
id: 999
```

Events can be sent without specifying the event field and using the data field only. These messages would be interpreted by the client's default `onmessage` handler. For the event above, the client must have configured an event handler for "update" events to function, such as the following:

```
source.addEventListener("update", function(event) {
    update.textContent = "Update Number: " + event.data;
}, false);
```

4.2.4 Automatic Reconnection

When a connection is disrupted or closed improperly, the browser will attempt to reconnect automatically. An `EventSource` object possesses a **reconnection time** and **Last-Event-ID** attribute, which is used to guarantee that the browser can pick up where it left off prior to disconnection. By default, the

reconnection time value is 3 seconds. This can be reconfigured at the server's discretion.

When a connection fails, the browser will wait for the reconnection time period and then initiate a new request using the exact same URL parameters as before. The Last-Event-ID attribute is also included, which will allow the server to determine which events were missed and require re-sending, without repeats. This capability is highly appealing due to its simplicity and the lack of additional configuration required by clients. Handling error-prone or lossy connections is easily solved with automatic reconnection inherently built-in.

4.2.5 Keep-Alive Methods

Since an EventSource connection is a persistent, open connection, there must be some methods to keep the connection alive. This is mostly due to the presence of proxy servers and other stateful middleboxes which contain low timeout values for stale HTTP connections, often less than 120 seconds. This requires that the server send at least a comment or event to keep the connection alive before the timeout interval is reached. The specification claims this should be done every 15 seconds or so [27], which is very frequent and impractical for mobile devices looking to maintain highly efficient energy consumption patterns.

If a connection is torn down by anything other than the client or server explicitly, the SSE will attempt to reconnect automatically. With Last-Event-ID support configured correctly, a client could receive all missed events as soon as the connection is re-established, which slightly diminishes the importance of keeping the connection alive. However, to reduce the overhead cost of unnecessary reconnections, a heartbeat event optimally timed within the proxy timeout interval may be the best approach for maintaining the connection efficiently.

4.2.6 Application Potential

Server-Sent Events have great promise in becoming a widely used standard for one-way server push applications. This is due to their simplicity, widespread compatibility, automatic reconnection, and high scalability. There are a tremendous amount of applications using unidirectional push [9, 43] such as email, social networking/browser notifications, stock tickers, currency updates, RSS feeds, and many more. Server push functionality is an essential modern feature in today's web applications and SSE standardization will bring this feature into the browsers as a native component.

4.2.7 Mobile Usage

Currently, the latest versions of Firefox, Safari, Android Browser, and Opera all support SSEs in their mobile browsers [34]. The only notable exception is Internet Explorer at this time. However, there exists various poly-fill techniques using JavaScript that are used to provide SSE support for non-compliant browsers [18]. At this time, there do not appear to be many mobile web applications that are making use of SSEs. Since the technology is still quite new and in its early phases of standardization, this will change over time. It is important to ensure that SSE connections are properly supported by mobile browsers and take into account mobile-unique aspects such as screen timeouts and the need for maintaining a low-power state. The mobile performance of SSEs will be examined in greater detail in Chapter 7.

4.2.7.1 Connectionless Push

One highly interesting feature of SSEs is the ability to support connectionless push [27]. This is a feature particularly suited for mobile devices that possess a great need to conserve battery power. The premise is that the device will offload the management of the persistent connection to a proxy server, in order to remain in a sleep/idle state for as long as possible. The *push proxy* server then delivers events using a technology such as OMA push which will wake the device enough to process the event and go back to sleep mode. This process on a mobile device can be described as follows:

1. Mobile browser connects to HTTP server using EventSource constructor.
2. After the server sends some messages, the browser determines that the only network activity is being used to keep the TCP connection alive and decides to switch to sleep mode.
3. The browser disconnects from the server and requests that a service on the network (push proxy) maintain the connection instead.
4. The push proxy service contacts the HTTP server and requests the same EventSource resources, including the Last-Event-ID header.
5. The browser allows the mobile to enter sleep mode.
6. The server sends another event, which is sent by the push proxy to the mobile device using a technology such as OMA push.

7. The mobile device wakes up just enough to receive the event, process it, and go back to sleep.

This connectionless push scheme is optimal for mobile efficiency when maintaining persistent connections is required. However, this type of service must be supported by the carrier network and has not been discovered to be operational yet. For truly efficient mobile push connections using SSE, this feature must be leveraged in future network implementations and proxy servers.

Let us now proceed to the measurements setup used to test the mobile readiness of both WebSockets and Server-Sent Events.

Chapter 5

Measurements

One of the main goals of this thesis was to test the mobile performance and readiness of the WebSocket and Server-Sent Event protocols. These technologies are important because they will serve the needs of a rapidly growing number of web users. For this reason, these technologies must be efficient and function correctly, taking into account all the unique characteristics of mobile devices. Namely the large variety of manufacturers, form factors, screen sizes, processing power, memory, various browsers, operating systems, and most of all, battery life.

The goal in these experiments was to use relevant mobile devices, browsers, and operating systems to determine how implementable these new HTML5 communication technologies are. Since they are not widely used in web applications yet, there is still some time before major adoption takes place. These experiments were focused on two main metrics: *performance* and *energy efficiency*. Using relatively simple implementations of both WebSocket and Server-Sent Event-based applications, we were able to test these across a variety of mobile devices and connection mediums.

Since both of these technologies require persistent connections, keep-alive mechanisms were a major focus in these experiments. The goal was to determine if these heartbeat mechanisms play a significant role in maintaining the connection and their effect on the overall power consumption within the mobile device. This chapter will outline the scope of the experiments for both WebSockets and Server-Sent Events. This will cover the objectives, procedures, application setup, and the tools used.

5.1 WebSocket Experiment

The WebSocket (WS) experiments were primarily focused on maintaining a persistent connection successfully and efficiently across a spectrum of mobile devices and connections. Testing various browsers across 4 mobile devices, a WebSocket-based chat application was used to determine the usability and efficiency of various keep-alive strategies for a long-lasting idle connection. This was tested using two 3G UMTS networks, Elisa and Sonera in Espoo, Finland, one 4G LTE network, and WiFi as shown in Table 5.1.

Device	Platform	Browser	Connections	Number of Tests
HTC Nexus One	Android 2.3.5	Mozilla Firefox 19	3G UMTS - Elisa WiFi - aalto open	47
Nokia Lumia 820	Windows Phone 8	Internet Explorer 10	3G UMTS - Elisa 3G UMTS - Sonera 4G LTE - Sonera WiFi - aalto open	45
Apple iPhone 4S	iOS 6.0	Safari	3G UMTS - Elisa WiFi - aalto open	19
Samsung Galaxy S3	Android 4.1.2	Mozilla Firefox 20 Chrome 25.0.1364.169 Opera Mobile 12.1.4	3G UMTS - Elisa 4G LTE - Sonera	35

Table 5.1: Mobile devices, browsers, and connections tested using WebSocket.

5.1.1 Objectives

The main objectives for these measurements were to determine:

- The impact of heartbeat keep-alive messages on the power consumption of mobile devices using WebSocket connections.
- An effective means of maintaining maximum power efficiency while keeping a WebSocket connection persistently active.
- The strategies used for optimal power consumption by network operators over various connection types.

5.1.2 Libraries Used

For these experiments, a multi-user chat application was used, based on the client and server code created by Michael Mukhin [35] and adapted by Thi Van Anh Pham. The web server chosen was Node.js [30] version 0.8.18 in conjunction with Socket.io [20] version 0.9.11. Node.js was selected as the

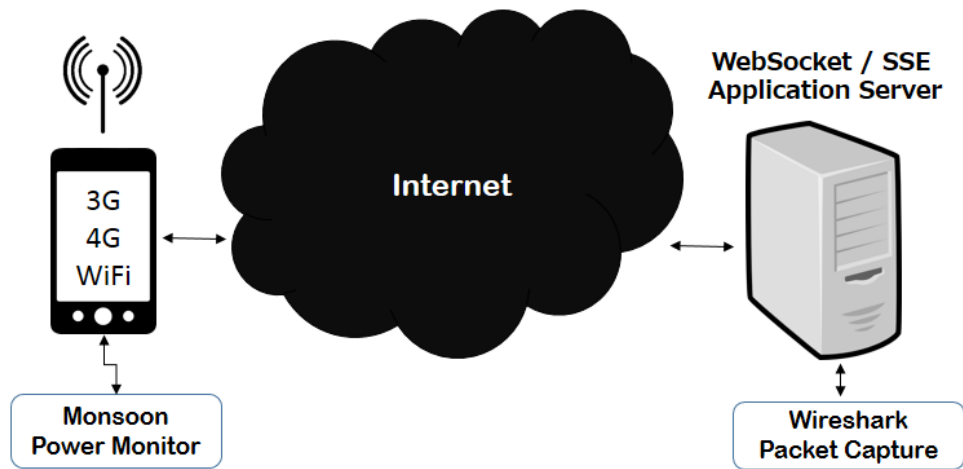


Figure 5.1: Experiment topology used in all measurements.

server framework due to its event-driven, non-blocking, and asynchronous nature. This allows for the creation of simple and scalable real-time network applications that can support many concurrent connections. Socket.io is used to provide support for WebSockets. It is written in JavaScript and its goal is to support real-time applications across all browsers and mobile devices [20].



Figure 5.2: Nokia Lumia 820 connected to Monsoon Power Monitor.

5.1.3 Tools Used

Additional tools used in these experiments include the following:

- **tcpdump** - for capturing incoming/outgoing network traffic from the server for packet behavior analysis.
- **Wireshark Network Protocol Analyzer** - for detailed packet capture analysis.
- **Monsoon Power Monitor** - for power/energy consumption measurements and analysis on all mobile devices.

All devices were powered entirely using the Monsoon Power Monitor, bypassing the battery itself and allowing the software to accurately record and measure power, current, and energy consumption levels. This required specially aligning copper wire on the positive and negative power terminals of the device. Electric tape was used to insulate the positive terminal, and the battery was replaced with these copper wires now correctly in place. Positive and negative probes were then attached to these wires, successfully allowing the Power Monitor to power the device safely using standard 3.7 voltage levels. This is shown for the Lumia 820 above in Figure 5.2.

5.1.4 Application Code

The client code for the chat application is written in HTML and Javascript. The user is prompted to enter a username upon visiting the webpage. This is followed by a simple chat interface which can be engaged at will. The chat application view from the client and server perspective on a Windows 7 PC is shown in Figure 5.3. The server code is written in JavaScript and provides functionality for multiple users. Some of the most important code is as follows:

```
var app = require('http').createServer(handler)
    , io = require('socket.io').listen(app)
    , fs = require('fs')

app.listen(80);
app.timeout = 0;
io.set('transports', ['websocket']);
io.set('heartbeat timeout', 120);
io.set('heartbeat interval', 60);
io.set('close timeout', 1800);
```

This sets up the HTTP server to listen on port 80. The close timeout for TCP connections is set to 30 minutes to keep all idle connections active for at least that period. This value was originally overlooked, causing the connections to be disconnected prematurely after the default value of 60 seconds. The transport protocol used is set to WebSocket only, preventing all other forms of bidirectional communication from taking place. Finally, the code to set the heartbeat timeout and heartbeat interval is listed. These two values were crucial during these experiments as they determined the frequency of the heartbeat messages used to keep the connection alive.

5.1.4.1 Heartbeat Messages

The Socket.io library uses a default *heartbeat interval* of 25 seconds, with a client *heartbeat timeout* interval of 60 seconds. This means that every 25 seconds, the server will send the client a heartbeat message, after which the client will respond directly to the server, assuring that it is still connected. If the client does not respond, the connection is closed by the server.

Additionally, the client will wait *heartbeat timeout* seconds since the last heartbeat it received from the server, before deciding that the server is no longer active and begin to disconnect. This is the current mechanism to keep persistent WS connections alive using Socket.io.

There is also a *close timeout* value which informs the client for how long to wait before closing the connection. If packets are not exchanged within this *close timeout* interval, then the client will disconnect. This value is crucial for applications that require long-lasting connections as it must be configured for longer lengths of time accordingly. The default value is 60 seconds, which was initially causing premature disconnections for any heartbeat interval that exceeded 60 seconds.

These values have been altered and tested to determine their effect on mobile power consumption and connectivity behavior. This was done primarily by adjusting the heartbeat interval to values lower and greater than the default, such as 10, 30, 60, 300, 900, 4200 seconds, etc. In all measurements, the client *timeout interval* was always set to double the heartbeat interval. Thus, if the *heartbeat interval* was 25 seconds, the client *timeout interval* was 50 seconds.

5.1.5 Procedure

The procedure for all test runs on each mobile device is as follows:

1. Activate the Node.js WebSocket-chat server. Start tcpdump PCAP capture for all HTTP traffic.

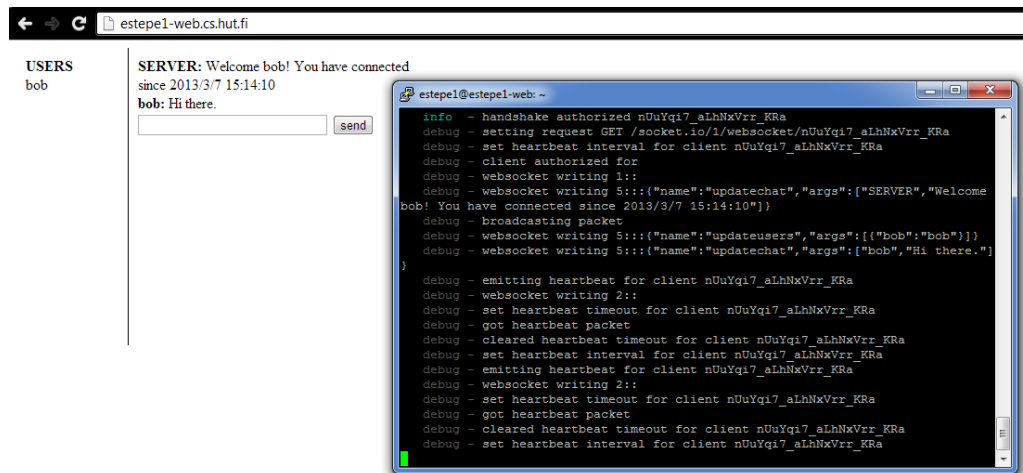


Figure 5.3: Client and server view of WebSocket chat application.

2. Start recording on Monsoon Power Monitor.
3. On HTC Nexus One device, open the Firefox browser and access the chat application via the recently visited pages menu. When prompted for username, enter “bob”. Press enter key.
4. On Nokia Lumia 820 device, open the chat application page via the Live Tile on the start screen. After webpage loads, enter message “bob” in the chat message box. Press enter key.
5. On Apple iPhone 4S device, open the chat application bookmark from the home screen, opening the Safari browser. When prompted for username, enter “Bob” and press OK.
6. On Samsung Galaxy S3 device, open the browser from the home screen, which navigates directly to the chat application. When prompted for username, enter “bob” and press OK.
7. With the WebSocket connection now established, leave the device idle as the run continues.
8. For the Nexus One, the screen times out after 1, 2 or 10 minutes, depending on the test. For the Lumia 820, the screen times out after 5 minutes. For the iPhone, the screen times out after 2 or 5 minutes. For the Galaxy S3, the screen times out after 1 or 2 minutes.

9. If the connection is maintained, even after the screen has timed out, the run is considered successful. Otherwise, note the time of disconnection. Close the mobile browser. Stop the Power Monitor measurements, end packet capture tracing, and stop the server. Save the results.

5.2 Server-Sent Event Experiment

The Server-Sent Event (SSE) measurements were also focused on maintaining a persistent connection efficiently using a variety of mobile devices and browsers. In this experiment, similar measurements were conducted in order to determine how long a mobile connection could be sustained using EventSource. A simple clock update application was used with various update interval frequencies to determine the optimal interval for event updates to keep the connection alive. The same devices were used as in the Web-Socket measurements, with the exception of the Lumia 820, which did not support SSEs. WiFi was not tested in these measurements to keep the focus solely on the performance and efficiency of mobile networks. The experiment overview is shown below in Table 5.2.

Device	Platform	Browser	Connections	Number of Tests
HTC Nexus One	Android 2.3.5	Mozilla Firefox 19.0 Opera Mobile 12.1.4	3G UMTS - Elisa	9
Samsung Galaxy S3	Android 4.1.2	Mozilla Firefox 20.0.1 Chrome 25.0.1364.169 Opera Mobile 12.1.4 Android Browser 4.1.2	3G UMTS - Elisa 4G LTE - Sonera	70
Apple iPhone 4S	iOS 6.0	Safari Chrome 26.0.1410.50	3G UMTS - Elisa	12

Table 5.2: Mobile devices, browsers, and connections tested using Server-Sent Events.

5.2.1 Objectives

The main objectives for these measurements are:

- To study the behavior and performance of EventSource connections on several mobile platforms and browser types.
- To assess the performance of SSE and the effect this has on mobile energy consumption, particularly during screen idle mode.
- To determine effective mechanisms to keep the SSE connection alive for prolonged periods in an energy efficient way.

5.2.2 Libraries Used

For these experiments, a simple updating clock webpage was created based on the client and server code by Colin Ihrig [14]. Node.js version 0.8.18 was used again as the server framework, due its efficient single-threaded, event-based mode of operation. Since SSEs require persistent HTTP connections, servers such as Apache which use operating system threads to handle individual requests become overburdened quickly [30] and are not ideal for SSEs.

5.2.3 Tools Used

The additional tools used in these measurements are identical to those in the WebSocket experiments. These are: **tcpdump**, **Wireshark**, and **Monsoon Power Monitor**. They are used in the same manner as before, to capture packets and perform power measurements.

5.2.4 Application Code

The client code was written in HTML and JavaScript, featuring support for real-time updates within a simple interface as shown in Figure 5.4 on a Windows 7 PC. The server is setup to continuously stream the current date and time at specified intervals using an EventSource event-stream. The user is able to disconnect and reconnect to the stream at any time. Should the connection become severed on either end, the client browser is set to attempt to reconnect every 1 second.

Each time a time/date update event is sent, a counter is incremented and updated in the client browser, in order to demonstrate the continuity of the event-stream. By altering the event update interval (EUI), the stability of the connection is affected, as well as the overall energy consumption of the mobile device. The aim of the experiment is to determine the optimal EUI that keeps the connection alive for prolonged periods of time, while remaining as energy efficient as possible.

The server code that defines the custom event called *update* is listed below. The event update interval frequency is specified here, in this example, at 5 seconds.

```
interval = setInterval(function() {
  counter++;
  res.write("event: update\n");
  res.write("data: " + (counter) + "\n\n");
  res.write("data: " + (new Date()) + "\n\n");
}, 5000);
```

The client must be configured to receive and process update events. A simple event handler is created for this.

```
source.addEventListener("update", function(event) {
  update.textContent = "Update Number: " + event.data;
}, false);
```

5.2.4.1 Event Update Intervals

The EUIs used were between 5-120 seconds, followed by longer measurements of 5-600 minutes. The EUI is the critical variable in these measurements. This interval specifies how often a new update event is sent to the mobile device, and essentially serves as a keep-alive mechanism or heartbeat. Without this, the connection would likely fail due to a timeout value being reached somewhere on the network path. If the device is frequently receiving events, it stays in an active state that consumes a high amount of energy. If the device receives updates less frequently, it is able to enter the more-efficient IDLE state, where the device can still receive updates without breaking the original connection. Therefore, the goal was to determine the optimal interval where the connection could still be maintained indefinitely, while remaining as energy efficient as possible.



Figure 5.4: Client and server view of SSE clock application.

5.2.5 Procedure

The procedure for all test runs on each mobile device is as follows:

1. Activate the Node.js SSE Clock Streaming server. Start tcpdump packet capture on *-eth0* interface on the server.

2. Start recording on Monsoon Power Monitor.
3. On each mobile device, open up the web browser which proceeds directly to the SSE server. The EventSource stream connection is now opened and will start processing the updates within the browser.
4. Leave the device idle until the screen times out and the run continues. By observing the Monsoon Power Monitor, it is possible to determine when event updates are sent by the server while the device is in idle mode.
5. After a sufficient amount of time, unlock the mobile phone and examine the update number. Determine whether the update number is correct and possibly wait until the next interval to determine if the stream is still actively updating correctly.
6. If the stream updates are still being displayed correctly, the run is considered successful. If the update number is the wrong value or has returned to 0, the run is considered unsuccessful.
7. Touch the “Disconnect” button within the browser. Stop the packet captures. Stop the Monsoon Power Monitor recording. Stop the server.
8. Save the results and examine the packet capture and power measurement files to better understand the run behavior and performance.

In total, 146 WebSocket tests were conducted across all devices. For SSE testing, more than 90 tests were conducted. The results of these measurements will now be examined in Chapters 6 and 7.

Chapter 6

WebSocket Results

The results for all sets of WebSocket tests will be discussed now. Each device will be examined and analyzed in terms of power consumption and performance in maintaining the connection. A better understanding of how the mobile device behaves and performs across several browser and connection types was the goal of these measurements. First, let us examine some of the key findings and factors that played a significant role in obtaining these results.

6.1 Initial Results

Over the course of these measurements, it has been revealed that the heartbeat interval (HBI) does not play a significant role in whether or not the connection is kept alive, even during a period of complete inactivity on the client side. During the first bulk of tests, it was discovered that the connection could not be maintained without a heartbeat occurring at least every 59 seconds for most devices. Thus, it was originally believed that the HBI was a significant factor in maintaining the connection for longer periods.

However, it was later discovered that the *close timeout* interval on the server was still in its default state of 60 seconds and was responsible for these premature disconnections. Once this value was extended to 30 minutes or greater, the connection was able to last without problems on several devices. Therefore, the focus of these measurements evolved and became twofold:

1. To reveal the draining energetic effect that frequent heartbeats can have on a mobile device, especially during idle mode.
2. To reveal how long a mobile WebSocket connection could be maintained without the use of heartbeat/keep-alive mechanisms.

6.1.1 Improper Server Code

The majority of the measurements were conducted on the false notion that something other than the server was responsible for severing the WebSocket connections prematurely. This was due to my own lack of care when setting the proper timeout values, which I believed to be working correctly at the time. Thus, the HBI used in most measurements are quite low, reaching a maximum of 59 seconds. Once it was discovered this threshold could be breached easily, additional measurements were conducted using HBIs of 5, 15, and 70 minutes. A test run was also conducted with a HBI of 10 hours. This was to demonstrate the energetic superiority of using infrequent heartbeat values and to reveal that the WebSocket connection can be successfully maintained without any additional packets or keep-alive mechanisms.

Unfortunately, the discovery that the close timeout was not set properly came near the end of the thesis period. Thus, the majority of these measurements have been conducted using heartbeat intervals that are too low for practical and sustainable mobile use. Still, they will be discussed below, giving a clearer idea on how keep-alive mechanism frequency within mobile devices can affect energy consumption significantly.

6.2 Key Findings

Some of the key findings from all measurements are summarized here now:

- Keep-alive mechanisms are not necessary to maintain a WebSocket connection persistently on a mobile device.
- The longest-lasting connection tested used a heartbeat interval of 10 hours for a total duration of 21 hours and 35 minutes before ending the test.
- The mobile browser correctly displayed all chat messages from other clients and was still perfectly functional.
- Heartbeat packets can play a significant role in keeping the connection alive depending on the server timeout values for all connections.
- If the server possesses a close timeout value, indicating the amount of time the client should wait before closing an idle connection, then a heartbeat exchange must occur between the mobile and server before that threshold is reached.

- A HB interval that is too short (1 minute or less) will cause the device to over-utilize network resources and consume excessive energy, resulting in unacceptable battery life.
- An optimal HB interval will keep the WebSocket connection alive for prolonged periods while avoiding all timeout values inherent in the server and network middleboxes.
- Optimal HB intervals of 15-70 minutes or more allow enough time for the device to remain idle as to be truly energy efficient and cause minimal battery drain.
- If a mobile client connection is abruptly cut off or power is lost on the mobile device, the server will not know about this disconnection until a keep-alive message is sent. In the meantime, the server will be wasting network resources and possibly sending messages to a disconnected client.
- Therefore, keep-alive mechanisms should be used at appropriate intervals to make sure that the client or server are still actively connected and require the allocated network resources to maintain that connection.

6.2.1 Device-specific Findings

- All devices and browsers did not perform in a uniform manner.
- The Lumia 820 with Internet Explorer 10 was only able to maintain the connection with a HB interval of 29 seconds or less.
- The iPhone 4S with Safari was unable to maintain the connection after the screen timed out.
- The Nexus One and Galaxy S3 using Firefox and Chrome were able to maintain the WebSocket connection successfully for long periods even with the screen off. This was successfully tested up to 21 hours and 35 minutes. Opera Mobile was unable to respond to heartbeat packets correctly and caused the connections to fail prematurely.
- Thus, browser support and operating system are significant factors in WebSocket keep-alive performance.

- 3G operators Elisa and Sonera possess slightly different inactivity timer strategies, resulting in slightly different HB behavior and affecting overall energy consumption rates.
- In terms of energy efficiency for longer runs, WiFi ranks first, 3G UMTS second, and 4G LTE last.
- Regardless of the network type, the behavior of the mobile phone was virtually identical.

The results for all devices are displayed in this chapter with the following data: the HBI used, the total time for the run before Monsoon power measurements were stopped, average power, average current, expected battery life, and the success or failure of the run. The results will be divided into two sections: the longer lasting runs with unconstrained heartbeat intervals and the shorter runs with heartbeat intervals of less than 60 seconds. This is done because the measurements occurred at separate times and pose different implications that should be considered separately.

It is worth noting that the total time of each run includes all stages of the procedure mentioned in Chapter 5 including: starting from the idle state with the screen always on, opening the browser and WS chat application, entering in the username/message, and then waiting. The time of disconnection is in reference to the beginning of the Monsoon power measuring. Simply put, there is approximately an 8-14 second delay in each run before the WS connection is established and the chat application is running.

Therefore, the time of disconnection is not a direct representation of how long it took before the WS connection was terminated, as this delay must be factored in to be truly accurate. This delay cannot be easily determined as each run was unique because of the human factor involved. However, this does not detract from the overall results because the important values are the heartbeat intervals, the amount of energy consumed, and the outcome of the connection, particularly when the screen goes idle.

6.3 Long Heartbeat Interval Results

The results shown here are the measurements conducted after realizing that the WS connection can be extended for much longer periods of time than originally believed. Heartbeat intervals here last at a minimum of 5 minutes and extend up to 10 hours. Due to time constraints nearing the completion of the thesis, testing was only possible using the 3G UMTS Elisa network. After conducting many measurements on both LTE and WiFi, it is confidently

assumed that the behavior and performance on these networks would be similar to those of 3G, differing only in energy consumption. The Nexus One and Galaxy S3 were selected for longer testing because of their ability to maintain the WS connection in screen idle mode. The iPhone 4S could not do this and the Lumia 820 required a HBI of 29 seconds or less.

Device	Browser	HBI (mins)	Time (mins)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
Nexus One	Firefox	5	31.07	93.68	25.30	55.33	Successful run.
Nexus One	Firefox	15	31.65	75.96	20.52	68.24	Successful run.
Nexus One	Firefox	600	526.41	33.10	8.44	156.6	Lasted 21 hrs 35 mins total.
Nexus One	Opera	5	21.86	111.62	29.99	46.68	Fail. Does not respond to HB.
Galaxy S3	Firefox	5	31.57	137.37	36.91	56.89	Successful run.
Galaxy S3	Firefox	15	31.29	130.33	35.12	59.80	Successful run.
Galaxy S3	Firefox	70	71.89	64.48	17.37	120.88	Successful run.
Galaxy S3	Chrome	5	31.36	128.67	34.57	60.74	Successful run.
Galaxy S3	Chrome	15	30.80	113.09	30.47	68.92	Successful run.
Galaxy S3	Opera	5	15.62	152.41	40.95	51.28	Fail. Does not respond to HB.
Galaxy S3	Opera	15	32.74	107.75	29.03	72.34	Fail. Does not respond to HB.

Table 6.1: Longest WebSocket runs - 3G UMTS Elisa

6.3.1 Browser Support

Both Firefox and Chrome were able to maintain the connection virtually indefinitely without problems. In Firefox, it was found that modifying the HTTP keep-alive timeout value had no effect on maintaining the WebSocket connection. There were some issues with Opera Mobile, which were verified on both the Nexus One and Galaxy S3. During heartbeat exchanges, the mobile device would not respond to the server's PING message, eventually triggering the heartbeat client timeout and causing a disconnection to occur. This usually took place after the device had been idle for 5 or more minutes. This shows that the browser support is critical for handling the keep-alive mechanisms. Even with WebSocket support, there may be other factors affecting the ability of the browser to respond correctly during long periods of no activity which need to be considered.

6.3.2 5 Minutes

Runs with HBIs of 5 minutes or greater were successful using Firefox and Chrome only. The battery life estimates for both the Nexus One and the Galaxy S3 are between 55 and 60 hours, with average power consumption ranging from 93 - 137 mW. These values may be considered reasonable for some users, but could be significantly improved by extending the HBI.

6.3.3 15 Minutes

By increasing the HBI to 15 minutes, overall energy efficiency improved to a range of 75 - 130 mW for both devices. There is an 18 mW improvement in the Nexus One results, while only a 7-15 mW increase on the Galaxy S3. Battery life estimates did improve up to 68 hours, again confirming that the longer the interval between heartbeats, the more energy saved over time.

6.3.4 70 Minutes

A HBI of 70 minutes was tested using Firefox on the Galaxy S3. This run proved to be successful and consumed an average of 64 mW of power with an estimated battery life of 120 hours. This is a significant improvement compared to the other Galaxy S3 runs, where around half the power is consumed. Battery life estimates in the range of 5 days are quite excellent and can be considered a sustainable interval on a mobile device.

6.3.5 10 Hours

The longest-lasting run was tested using the Nexus One and the Firefox Mobile browser. The run was successfully tested for a total of 21 hours and 35 minutes and was still able to function after such a long period of being idle. A 10 hour HBI was used, suggesting that there are no middlebox timeout values within 10 hours over the Elisa 3G network. Even after long periods of no activity, the Nexus One was triggered instantly by a chat message sent by another connected user, suggesting that the real-time functionality works as promised. The Monsoon Power Monitor froze after 8.77 hours of measuring the energy, resulting in only a small sample of the entire run values. Still, the superior levels of energy consumption are made clear with power consumption of 33 mW and battery life estimates of 156 hours or 6.5 days. Maintaining the connection is not costly for the device in idle mode, especially with infrequent heartbeats.

6.3.5.1 Device Behavior

It has been confirmed by analyzing the packet captures that no TCP keep-alive or other unnecessary messages were sent during these long idle periods. The server eventually sent a HB packet to the mobile device after 10 hours. It took approximately 2.5 seconds for the Nexus One to wake up and respond to this packet, suggesting that the device is in IDLE mode and not PCH mode, as a new radio control connection was likely established to send the response. The device is able to maintain an IP address and the WebSocket connection alive, while still remaining in the ultra-efficient IDLE state. This is the ideal behavior for these types of connections.

This successful run reveals that the WebSocket connection can potentially last up to 24 hours or more without a single packet being transferred between the mobile and the server. This is impressive and means that the WebSocket can be considered reliable and robust for applications that require a persistent bidirectional communication channel.

6.3.6 Heartbeat Exchanges

Heartbeat packets triggered an energetic response from the Galaxy S3 lasting approximately 9.4 seconds. On the Nexus One, the tail energy lasted for approximately 7-8.5 seconds per heartbeat. These are reasonable and expected values given the nature of the 3G network. While heartbeat exchanges are quite wasteful given the small amount of data transferred, these transfers appear to be sustainable when spaced out in long intervals.

The results suggest that heartbeat intervals of 15, 30, or 70 minutes may be adequate for sustainable mobile use. Greater values can also be selected depending upon the nature of the application. This proves that great flexibility can be made when using persistent WebSocket-based applications. However, this is greatly dependent on the server configuration, especially regarding timeout values for closing persistent connections. The default values for this can be between 30 and 120 seconds, which would require the device to emit keep-alive messages within that time period to maintain the connection indefinitely. Let us now examine the results of such a server configuration.

6.4 Short Heartbeat Interval Results

The following results are for all devices when constrained by the 60 second server close timeout. At the time, it was believed that these results were optimal, and that premature disconnections were being caused by network

timeout values or buggy browser code. However, this is not the case. These results are still significant as they reveal the negative effects that frequent heartbeat intervals have on the battery life of these mobile devices. Also, these results indicate the use of default server timeout values, which is a common practice among developers. Therefore, these results illustrate the necessity to configure the server appropriately depending upon the mobile application's needs.

Each device did not perform identically when it came to maintaining the connection, even when conducted on the same cellular network. Table 6.2 reveals the best heartbeat intervals discovered for devices that were able to maintain the WS connection persistently, even with the screen off. Note that the iPhone was unable to achieve this and is not listed. It is logical that the optimal HB threshold would be at 59 seconds in this case, given the default configuration of 60 seconds. Table 6.3 reveals a sample of the power consumption for these runs while the screen was off, demonstrating the energy consumption differences during idle mode.

All tests conducted using 4G LTE took place on the Sonera network near Helsinki, Finland. All WiFi tests were conducted on the *aalto open* network at the Aalto University campus. All battery life estimations are made by Monsoon PowerTool software using the phone's original battery capacity as a parameter.

Device	Browser	HBI (secs)	Network	Time (mins)	Avg. Power (mA)	Avg. Current (mW)	Est. Battery Life (hours)
Nexus One	Firefox	56	3G (Elisa)	22.8	295.57	80.04	17.49
Nexus One	Firefox	59	WiFi	17	296.64	80.11	17.48
Lumia 820	IE10	29	3G (Elisa)	17.06	394.39	104.58	15.78
Lumia 820	IE10	28	3G (Sonera)	17.14	380.00	100.74	16.38
Lumia 820	IE10	29	4G LTE	16.83	576.14	152.75	10.80
Lumia 820	IE10	30	WiFi	13.51	300.49	79.66	20.71
Galaxy S3	Firefox	59	3G (Elisa)	12.29	314.21	84.42	24.88
Galaxy S3	Firefox	59	4G LTE	20.37	346.76	92.92	22.60
Galaxy S3	Chrome	59	3G (Elisa)	12.10	298.02	80.07	26.23
Galaxy S3	Chrome	59	4G LTE	17.59	370.01	99.15	21.18

Table 6.2: Successful runs using optimal HBI for all devices and networks.

The most obvious difference between Table 6.1 and 6.2 are in battery life. For longer HBI runs, the lowest estimate is around 55 hours of battery life, while for shorter runs, the longest battery estimate was around 26 hours. This is a significant difference and is definitely related to heartbeat frequency.

Device	Browser	HBI (secs)	Network	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)
Nexus One	Firefox	56	3G (Elisa)	176.02	101.54	27.50	50.92
Nexus One	Firefox	59	WiFi	183.95	32.97	8.90	157.24
Lumia 820	IE10	29	3G (Elisa)	182.70	199.83	52.97	31.15
Lumia 820	IE10	28	3G (Sonera)	181.44	237.32	62.92	26.22
Lumia 820	IE10	29	4G LTE	177.27	365.63	96.94	17.02
Lumia 820	IE10	30	WiFi	176.02	82.17	21.78	75.75
Galaxy S3	Firefox	59	3G (Elisa)	181.92	136.03	36.54	57.46
Galaxy S3	Firefox	59	4G LTE	319.91	330.75	88.63	23.69
Galaxy S3	Chrome	59	3G (Elisa)	179.85	102.91	27.65	75.96
Galaxy S3	Chrome	59	4G LTE	319.91	245.70	65.84	31.90

Table 6.3: Optimal run sample during display off period - demonstrating HB power consumption.

6.4.1 Results for Nexus One

The results for all significant runs on the Nexus One are found in Tables 6.4 and 6.5 below, separated according to 3G or WiFi connectivity. For all measurements, a constant voltage of 3.69V was applied, and all expected battery life estimates are calculated using a 1400 mAh battery. The setting to keep WiFi connections alive during sleep mode/screen timeout was enabled. The screen brightness was set to low, approximately 25%, and the screen timeout was set to 10 minutes. No other applications were open in the background other than the Google services which start up by default.

6.4.1.1 3G UMTS Elisa

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
D2	75	94.61	649.43	175.89	7.96	Disconnect at 79 sec
D3	60	136.28	605.44	163.97	8.54	Disconnect at 128 sec
D5	58	141.44	597.48	161.82	8.65	Disconnect at 127 sec
F1	58	135.00	647.06	175.23	7.99	Disconnect at 129 sec
D6	57	204.01	550.06	148.97	9.40	Successful run
F2	57	192.89	550.58	149.10	9.39	Disconnect at 186 sec
F3	56	1368.39	295.57	80.04	17.49	Total success (22+ mins)
A4	25	77.57	704.10	190.71	7.34	Successful run
B4	10	74.01	795.02	215.33	6.50	Successful run
C1	3	63.93	953.00	258.11	5.42	Successful run
C3	1	59.44	915.02	247.83	5.65	Successful run

Table 6.4: Nexus One - Firefox - 3G UMTS Elisa

There is a definite relationship between the heartbeat interval and power consumption. During the runs, the power consumption stays between 400-

500 mW on average, before surging to around 900 mW or more for approximately 8 seconds following a heartbeat message. While the entire HB exchange between the client and server occurs within 1 second (based on packet analysis), the tail energy produced by the UMTS connection lasts 6-7 more seconds, which is wasteful and costly. This pattern is shown below in Figure 6.1 and Figure 6.2 with the screen idle.

In the case of HB messages every 1, 3, and 10 seconds, no large power surge is detected. Instead, the power draw remains consistent at over 800mW as the device remains in the costly DCH state constantly. This causes significant power consumption and is completely unsustainable for longer periods. For longer HBI exceeding 55 seconds, when the screen is off, the HB messages trigger the device to a level of around 600 mW.

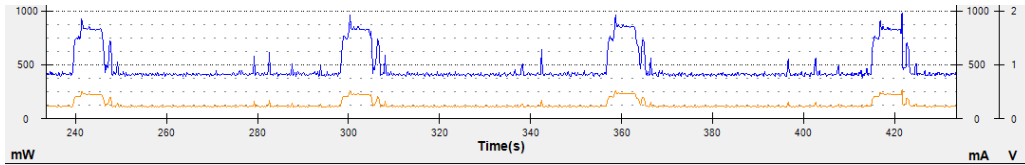


Figure 6.1: Run F3 - HB = 56 seconds. Screen is active.

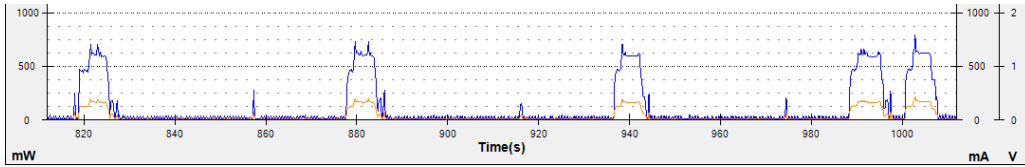


Figure 6.2: Run F3 - HB = 56 seconds. Screen is inactive.

6.4.1.2 WiFi

The WiFi results reveal that power consumption is significantly decreased compared to 3G, especially when the screen is active. However, over the long-run when the screen goes idle, the power consumption values become more similar to 3G, as the device remains mostly in a dormant low-power state. However, WiFi still proves to be the most energy efficient means to maintain a WebSocket connection, especially in the short run after the screen has timed out.

For runs where the screen was always active, HB intervals of 25, 10, and 3 seconds consumed an average power of 472 - 503 mW, which is much less than the range of 700 - 953 mW consumed during 3G measurements with

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
G1x	60	312.15	449.27	121.33	11.54	Disconnect at 306 sec
G2	59	1020.31	296.64	80.11	17.48	Disconnect at 977 sec
G3x	58	944.10	314.76	85.00	16.47	Disconnect at 879 sec
G4	57	694.40	412.94	111.52	12.55	Disconnect at 664 sec
G5x	56	996.90	296.38	80.04	17.49	Disconnect at 958 sec
G6	55	899.68	337.41	91.11	15.37	Disconnect at 854 sec
E1	50	163.91	472.34	127.92	10.94	Successful run
E2	25	135.77	472.54	127.97	10.94	Successful run
E3	10	135.20	478.07	129.47	10.81	Successful run
E4	3	135.40	502.62	136.12	10.29	Successful run

Table 6.5: Nexus One - Firefox - WiFi

the same values. This is expected as WiFi is known to consume less overall power than 3G connections. The HB messages trigger a power surge that lasts less than 2 seconds for the entire HB client-server exchange, compared to 3G with over 8 seconds.

The HB message exchange triggered by WiFi is barely noticeable in the power measurements compared to 3G. The radio for WiFi is always on and transmitting data, whereas the 3G radio must wake-up and transition into an active state before data can be transferred. This is one of the key reasons why WiFi is a more energy efficient connection medium from a behavioral perspective in the short term; there is far less tail energy than 3G. However, since WiFi is continuously transmitting data regardless of the activity state, it is possible that WiFi connections will consume more power over much longer periods of time than an efficient 3G connection.

6.4.2 Results for Lumia 820

The results for all significant runs on the Lumia 820 are found in Tables 6.4 and 6.5 below, separated according to 3G, 4G LTE, or WiFi connectivity. For all measurements, a voltage of 3.69 - 3.84 V was applied, and all expected battery life estimates are calculated using a 1650 mAh battery. The setting to keep WiFi connections alive with the display off was enabled. The screen brightness was set to low and the screen timeout was set to 5 minutes. No other applications were open in the background. The chat application was always opened via a Live Tile from the home screen for instant access.

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
D1	50	52.93	993.76	269.18	6.13	Disconnect at 46 sec
D3	35	48.70	1033.78	280.03	5.89	Disconnect at 42 sec
D4	30	73.20	976.71	264.56	6.24	Disconnect at 68 sec
F21	29	1023.48	394.39	104.58	15.78	Total success (17+ mins)
A3	25	100.64	906.74	245.61	6.72	Successful run
B1	10	64.05	1197.76	324.44	5.09	Successful run
C1	3	66.73	1631.30	441.92	3.73	Successful run

Table 6.6: Lumia 820 - IE10 - 3G UMTS Elisa

6.4.2.1 3G UMTS Elisa

The Lumia 820 consumes more power than the Nexus One on average, resulting in significantly less estimated battery life. This is most likely due to differences in hardware, screen types, and battery capacities. Average power draw ranged from 394 - 1631 mW, compared to 268 - 953 mW for the Nexus One over Elisa 3G. Battery life estimates ranged from 3.6 to 15.78 hours.

The tail energy is greater and lasts longer than the Nexus One, approximately 10-13 seconds, reaffirming that the device itself is a key factor in overall energy consumption. The power consumption for Run F21 is shown below in Figure 6.3, clearly depicting the HB intervals, as well as the difference in power consumption when the screen is idle. The tail energy exhibited is consistent regardless of the screen being active or not.

Interestingly, the Lumia 820 could not maintain the connection if the HB interval exceeded 29 seconds. This could be due to the fact that the Socket.io library does not officially support Internet Explorer 10 [20]. However, the WebSocket connection is established correctly during each run, and heartbeat messages do appear to sustain the connection indefinitely when this 29 second interval is used. This strongly suggests that there may be some browser code that is incompatible with Socket.io and is terminating the idle connection every 30 seconds. Currently, this heartbeat frequency will cause unsustainable battery life for Lumia devices using Socket.io-based WebSocket connections.

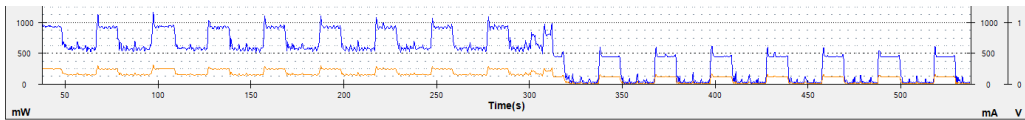


Figure 6.3: Run F21 - HB = 29 seconds. Screen goes idle at approx. 325 seconds.

6.4.2.2 3G UMTS Sonera

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
F8	30	71.71	1222.61	324.19	5.09	Disconnect at 69 secs
F12	30	138.86	1289.42	341.91	4.83	Disconnect at 131 secs
F11	29	378.86	635.68	168.53	9.79	Disconnect at 373 secs
F19	29	103.06	929.98	246.57	6.69	Disconnect at 98 secs
F19x	29	405.39	577.75	153.17	10.77	Disconnect at 374 secs
F20	28	1028.29	380.00	100.74	16.38	Total success (17+ mins)

Table 6.7: Lumia 820 - IE10 - 3G UMTS Sonera

Using the Sonera network, the power consumption levels were similar to the Elisa network on average. However, some important differences are noted. First, the optimal HB value proved to be 1 second less than that of the Elisa network. This is likely due to differences in the T1/T2 inactivity timer strategies used by both networks. Secondly, HB messages trigger a power surge lasting only 3.8-6 seconds during some exchanges, compared to 10-13 seconds on the Elisa network. This suggests that the IDLE to FACH transfer may be supported in the Sonera network or that the device is making use of Fast Dormancy. The overall result is less tail energy and energy savings over the long run. These faster state transitions are better equipped to handle low-data HB message exchanges efficiently and should be used whenever possible.

Interestingly, while the screen is idle, the device consumes similar levels of power consumption over 1000 mW during certain HB intervals, whereas on the Elisa network the device consistently reaches around 500 mW for each HB exchange. A comparison of Figure 6.3 and 6.4 demonstrates this relationship.

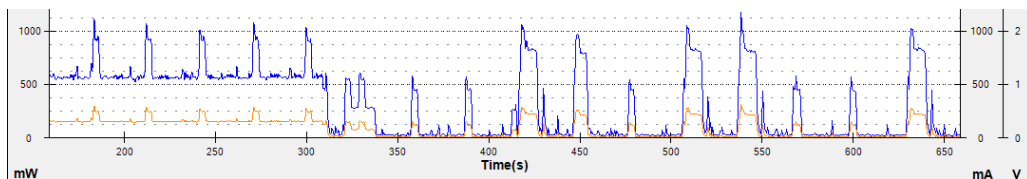


Figure 6.4: Run F20 - HB = 28 seconds. Screen goes idle at approx. 320 seconds.

6.4.2.3 4G LTE Sonera

The measurements performed on the Sonera 4G LTE network gives insight into overall energy efficiency compared to 3G UMTS networks. Some of these

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
F1	50	46.74	1087.04	288.97	5.71	Disconnect at 40, 41 secs
F3	35	74.71	1004.91	267.15	6.18	Disconnect at 71, 39, 70 secs
F4	30	378.36	840.48	223.43	7.38	Success, waited one addt'l HB msgs after screen timeout
F17	30	317.64	918.42	243.49	6.78	Disconnect at 311 sec after screen timeout
F17x	29	1009.57	576.14	152.75	10.80	Total success (16+ mins)

Table 6.8: Lumia 820 - IE10 - 4G LTE

runs were performed more than once to guarantee their accuracy. The LTE connection drew more power consistently than its 3G counterpart, fluctuating between 500 - 1000 mW on average during the Long DRX state where the radio is connected and waiting for data. For example, by comparing Run F11 on Sonera 3G and Run F4 on Sonera LTE, we can see the type of power difference. Both runs were 378 seconds, with LTE consuming on average 840 mW compared to 635 mW using 3G. The tail energy triggered by the HB message lasts 10-11 seconds in total, surging to over 2200 mW during LTE's Continuous Reception active state. Figure 6.5 below shows the power consumption after being triggered by a HB message when the screen is active.

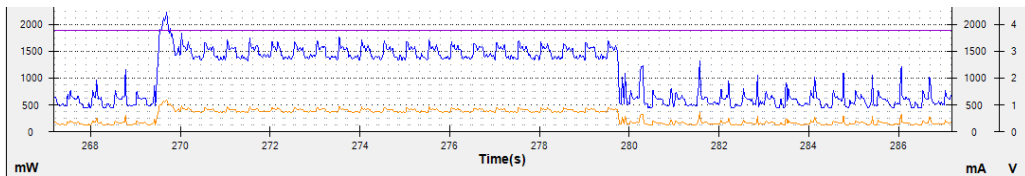


Figure 6.5: Run F17x - HB = 29 seconds. HB occurs at approximately 269.5 seconds.

The LTE optimal HBI is virtually identical to its 3G counterparts. When the HB interval exceeded 29 seconds, the connection would eventually be terminated while the screen was idle, sometimes after 2 or 3 additional HB cycles. This was a device-initiated disconnect, indicating that the browser could be the potential culprit. Overall LTE battery estimates ranged from 5.71 to 10.8 hours, which is 5-6 hours less than optimal connections using 3G and WiFi. This is a significant loss of battery life and can be attributed solely to the high-energy consumption rates of LTE and the frequency of the heartbeats. Figure 6.6 shows the power consumption patterns for the optimal LTE run, which surges as high as 2500 mW at times. Even in the idle screen state, the device reaches 1000 mW for each HB exchange.

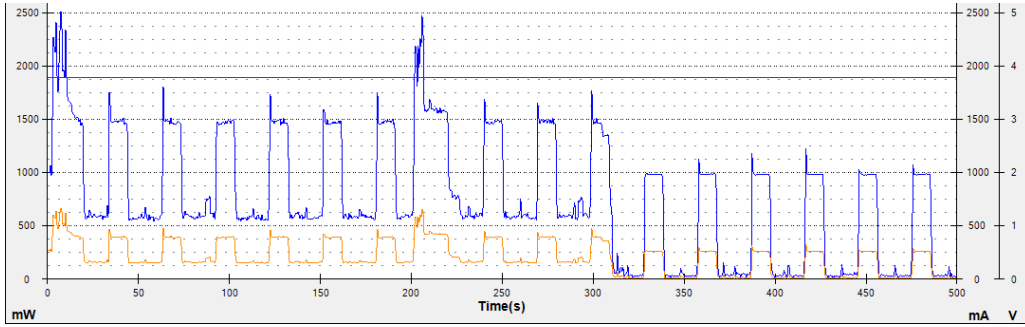


Figure 6.6: Run F17x - HB = 29 seconds.

6.4.2.4 WiFi

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
E1	50	43.72	744.39	193.78	8.51	Disconnect at 40 secs
E2	30	55.10	711.33	185.17	8.91	Disconnect at 48 secs
F16	31	105.04	632.74	167.75	9.84	Disconnect at 98, 66, 98 secs
F15	30	378.67	532.13	141.07	11.70	Total success (6+ mins)
F18	30	810.64	300.49	79.66	20.71	Total success (13.5+ mins)
E4	25	126.59	668.53	174.02	9.48	Successful run
E5	10	125.69	684.65	178.22	9.26	Successful run

Table 6.9: Lumia 820 - IE10 - WiFi

The WiFi results indicate that power consumption is significantly less than the 3G and LTE connections. Power draw remains fairly consistent for all runs regardless of HB interval, ranging from 664-744 mW. Battery life estimates range from 8.5-20.7 hours. The optimal HB interval of 30 seconds was found to maintain the WS connection successfully for more than 13.5 minutes. This value would need to be tested at greater lengths exceeding 20-40 minutes to determine if the connection would actually hold. Because the WiFi radio is always active, the HB interval does not affect average power consumption as much as 3G. For example, a HB interval of 10 seconds is similar to a HB interval of 25 seconds in terms of power consumption. Figure 6.7 displays the power consumption over WiFi. Note that the HB exchanges are virtually unnoticeable over WiFi, even when the screen is idle.

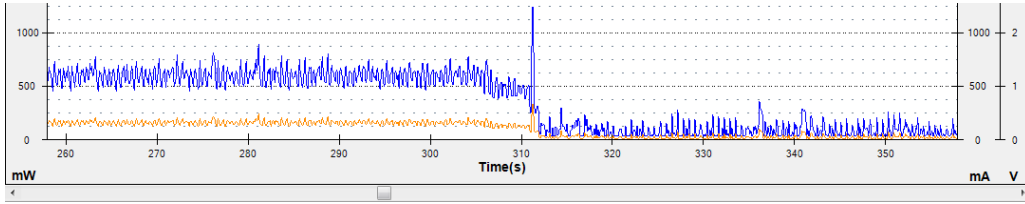


Figure 6.7: Run F18 - HB = 30 seconds. Screen transitions to idle. HB non-distinguishable.

6.4.3 Results for iPhone 4S

The measurements for the iPhone 4S were conducted over both 3G and WiFi. The battery life estimates are made assuming a 1432 mAh standard battery. The screen was always set at 25% brightness level, similar to the other phones. For all runs, the iPhone terminated the WebSocket connection as soon as the screen timed out. There was no setting to allow connections to continue during sleep mode or screen timeout. Therefore, the iPhone was unable to maintain a connection for prolonged periods of time and longer results could not be achieved to compare with the other devices. Still, HB intervals of 60 seconds or less were measured for both 3G and WiFi to provide a comparison in overall consumption to the other devices. Due to the similarities of the WiFi measurements to the Nexus One and Lumia 820, they are not shown below.

6.4.3.1 3G UMTS Elisa

Run	HB (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
A9	60	133.81	586.05	155.78	9.19	Disconnect at 128 sec
A10	59	134.17	588.84	156.52	9.15	Disconnect at 128 sec
A11	58	314.70	575.43	152.95	9.36	Disconnect at 310 secs
A8	57	316.25	555.94	147.77	9.69	Disconnect at 311 secs
A7	55	325.74	565.54	150.32	9.53	Disconnect at 317 secs

Table 6.10: iPhone 4S - Safari - 3G UMTS Elisa (5 min. screen timeout)

The longest HB interval tested for the iPhone was 58 seconds. However, the measurements reveal that HB intervals of 52 - 57 seconds were similar, if not better, in terms of overall energy efficiency. This shows that a minor discrepancy in the HB interval is not the sole determining factor in terms of overall energy efficiency, especially with the display active. There are other

variables to consider, mainly additional background and network resources that are unique for each run. Battery life is estimated to be around 9 hours, which is unacceptable for any persistent application. For the second round of measurements, the screen timeout was set to 2 minutes, and the HB interval values were brought down to 30, 25, 10, along with the optimal value of 58 as shown in Table 6.11.

Run	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
B4	58	136.61	583.44	155.08	9.23	Disconnect at 130 secs
B1	30	134.49	635.91	169.03	8.47	Disconnect at 129 secs
B2	25	134.43	706.57	187.82	7.62	Disconnect at 130 secs
B3	10	135.94	855.15	227.32	6.30	Disconnect at 132 secs

Table 6.11: iPhone 4S - Safari - 3G UMTS Elisa (2 min. screen timeout)

These results demonstrate that the HBI frequency affects the overall energy consumption of the device significantly. The shorter the HB interval, the more average power is consumed, resulting in a significant decrease in battery life. Meanwhile, if the HB interval is optimized to the server configuration, battery life is estimated to last approximately 0.5 - 3 hours more.

6.4.4 Results for Galaxy S3

The results for all significant runs on the Galaxy S3 are found in Tables 6.12 and 6.13, separated according to 3G and 4G LTE connectivity. For all measurements, a voltage of 3.73 V was applied, and all expected battery life estimates are calculated using a standard 2100 mAh battery. The setting to keep WiFi connections alive with the display off was enabled. The screen brightness was set to low (approx. 25%) and the screen timeout was set to 2 minutes. No other applications were open in the background. The chat application was tested using 3 browsers to better understand how browser support affects the WebSocket connection's longevity.

6.4.4.1 3G UMTS Elisa

Using Firefox and Chrome, the WS connection could be sustained for long periods. The longest runs for each of these browsers consumed almost identical amounts of energy at 299 mW over 11 minutes. Opera was unable to respond to server HB messages correctly after a certain threshold, eventually triggering the client timeout value. The HB exchanges generate tail energy lasting between 10-12 seconds consistently on all 3 browsers. However, the

Run	Browser	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
A5	Firefox	60	148.04	831.33	223.35	9.40	Disconnect at 114 secs
A4	Firefox	59	717.73	299.21	80.39	26.12	Total success (11+ mins)
B2	Chrome	60	135.68	837.53	225.02	9.33	Disconnect at 116 secs
B3	Chrome	59	726.38	298.02	80.07	26.23	Total success (11+ mins)
C2	Opera	60	144.79	820.56	220.46	9.53	Disconnect at 116 secs
C3	Opera	59	142.01	735.26	197.54	10.63	Disconnect at 115 secs
C4	Opera	58	318.49	394.35	105.96	19.82	Timeout after 4th HB.

Table 6.12: Galaxy S3 - All Browsers - 3G UMTS Elisa

Firefox and Chrome browsers differed slightly in their power consumption patterns during HB exchanges, suggesting that the way browsers handle HB message exchanges is not identical.

It is clear that Chrome is more efficient and generates fewer spikes in energy than Firefox. This suggests that Firefox is somehow over-reacting during the exchange and triggering unnecessary radio activity within the mobile. Regardless, the overall consumption is virtually identical over the long run, as shown in Runs A4 and B3. It is still interesting to note the subtle differences in behavior due to browser support, especially from an energetic perspective. This relationship is shown clearly in Figures 6.8 and 6.9.

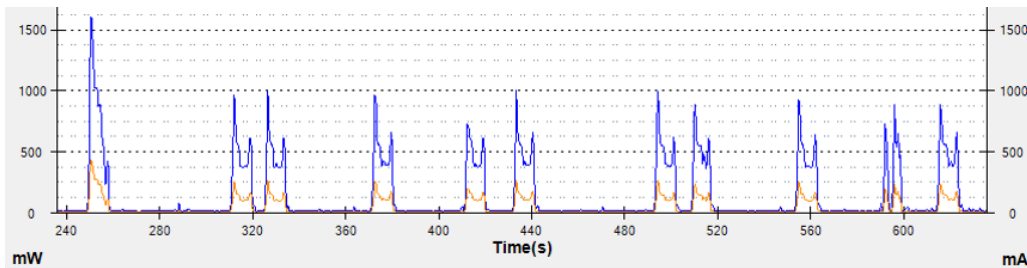


Figure 6.8: Galaxy S3 - Firefox - 3G Elisa. Note the double spike surges during HB exchanges.

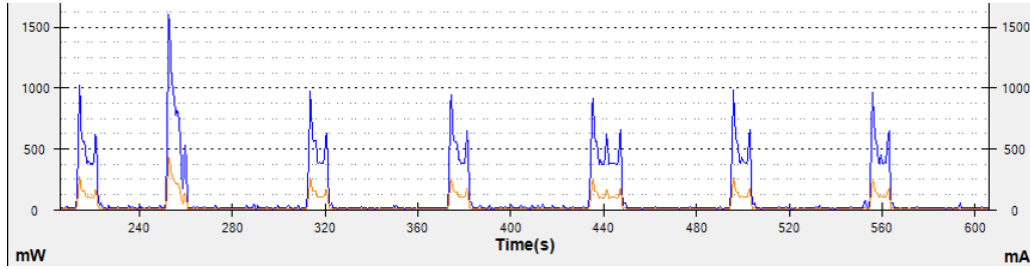


Figure 6.9: Galaxy S3 - Chrome - 3G Elisa. No unnecessary surges of power detected during HB exchanges.

Run	Browser	HBI (secs)	Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
D2	Firefox	60	137.65	1010.35	270.74	7.76	Disconnect at 115 secs
D3	Firefox	59	1222.55	346.76	92.92	22.60	Total success (20+ mins)
E2	Chrome	60	134.98	1184.80	317.50	6.61	Disconnect at 115 secs
E3	Chrome	59	1055.78	370.01	99.15	21.18	Total success (17+ mins)
F2	Opera	60	139.69	872.13	233.70	8.99	Disconnect at 117 secs
F3	Opera	59	356.94	514.68	137.92	15.23	Timeout after 4th HB.

Table 6.13: Galaxy S3 - All Browsers - 4G LTE Sonera

6.4.4.2 4G LTE Sonera

The results for the LTE runs were similar to those of 3G. Firefox and Chrome both succeeded, while Opera could not maintain the connection for longer periods. The phone would stop responding to server HB packets after the 4th HB, eventually triggering the close timeout. Again, we notice that the Firefox browser appears to be overactive compared to Chrome. The HB exchanges for Firefox and Chrome are demonstrated in Figures 6.10 and 6.11, producing around 10-12 seconds of tail energy.

6.5 Packet Behavior Analysis

By analyzing the packet captures for each of the runs, it was possible to gain a better understanding of what actually occurred between the device and the server when using the WS connection. Of particular interest was how the connection was setup, how the HB exchange process worked, and how the

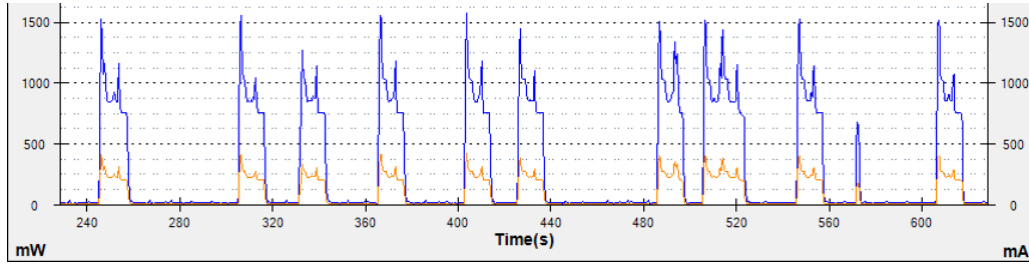


Figure 6.10: Galaxy S3 - Firefox - 4G LTE. Note the double spike surges during HB exchanges.

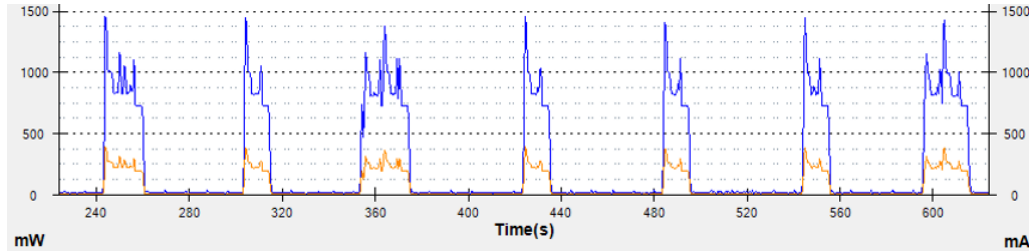


Figure 6.11: Galaxy S3 - Chrome - 4G LTE. No unnecessary surges of power detected during HB exchanges.

connection was terminated, particularly for runs that failed.

The packet behavior for all the devices is similar for both 3G, LTE, and WiFi. The HTTP Upgrade handshake to set up the WebSocket connection is quite lengthy. The duration from the first HTTP GET request from the client, to the server HTTP Switching Protocols response message ranges from approximately **1.3 to 2.1 seconds** for all runs, as shown in Figure 6.12. This means that all users would have to wait at least 1.3 seconds before the WebSocket connection was usable, which is quite long by today's standards. This setup process was identical across all devices. After the handshake is complete, the server transfers the *client.html* file to the client to be rendered on the mobile device. Note that the server IP address in the following figures is always **130.233.195.85**.

6.5.1 Heartbeat Exchanges

Heartbeat exchanges were similar across all devices, except for minor differences mainly in packet size and the presence of ACK packets. The HB packets from the server and client were of different sizes based on the device used. For example, from the server-side, 71 bytes for the Nexus One, iPhone

Time	Source	Destination	Protocol	Length	Info
5.865550	188.238.142.118	130.233.195.85	HTTP	350	GET / HTTP/1.1
5.868224	130.233.195.85	188.238.142.118	HTTP	232	HTTP/1.1 200 OK
5.943758	188.238.142.118	130.233.195.85	HTTP	408	GET /socket.io/socket.io.js HTTP/1.1
6.808376	130.233.195.85	188.238.142.118	HTTP	930	HTTP/1.1 200 OK (application/javascript)
7.104854	188.238.142.118	130.233.195.85	HTTP	384	GET /socket.io/1/?t=1362481965841 HTTP/1.1
7.107723	130.233.195.85	188.238.142.118	HTTP	235	HTTP/1.1 200 OK (text/plain)
7.203417	188.238.142.118	130.233.195.85	HTTP	428	GET /socket.io/1/websocket/stgpf7PNhilarprQEgr HTTP/1.1
7.205376	130.233.195.85	188.238.142.118	HTTP	183	HTTP/1.1 101 Switching Protocols

Figure 6.12: The WebSocket handshake over HTTP.

4S, and Galaxy S3, while only 59 bytes for the Lumia 820. This is likely due to the difference in browser engine types across those devices: both Chrome and Safari are Webkit browsers, Firefox is run by the Gecko engine, while IE10 is Trident-based. Both Webkit and Gecko are cross-platform while Trident is exclusively designed for Microsoft Windows operating systems, which could account for the packet size discrepancy. Otherwise, the server-initiated HB exchange behavior remained the same across all connection types.

860.490752	130.233.195.85	188.238.142.118	TCP	59	[TCP segment of a reassembled PDU]
860.893495	188.238.142.118	130.233.195.85	TCP	54	49201 > http [ACK] Seq=712 Ack=371
860.911113	188.238.142.118	130.233.195.85	TCP	63	[TCP segment of a reassembled PDU]
860.911143	130.233.195.85	188.238.142.118	TCP	54	http > 49201 [ACK] Seq=371 Ack=721
889.912516	130.233.195.85	188.238.142.118	TCP	59	[TCP segment of a reassembled PDU]
890.347185	188.238.142.118	130.233.195.85	TCP	54	49201 > http [ACK] Seq=721 Ack=376
890.348967	188.238.142.118	130.233.195.85	TCP	63	[TCP segment of a reassembled PDU]
890.348987	130.233.195.85	188.238.142.118	TCP	54	http > 49201 [ACK] Seq=376 Ack=730

Figure 6.13: Heartbeat exchange for Lumia 820. Same for 3G and 4G. In WiFi, the client ACK packet is missing.

Unfortunately, Wireshark was unable to provide much revealing information about what each client/server HB packet actually contained, describing them simply as *"TCP segment of a reassembled PDU."* The only messages captured classified under the WebSocket protocol was the username or the chat messages entered by the client. These packets were labeled as **WebSocket Text [FIN] [MASKED]**, with the payload being visible. By testing using a PC, it was discovered later that the 63 byte Lumia 820 response to a HB message is a WebSocket text packet with the payload of `"2:."`. This fits the profile of a small PONG packet and hints that the server packet is of a similar nature.

6.5.2 WebSocket Disconnection

During a successful run, the mobile browser would be closed manually on the device to terminate the WS connection. From the packet-level perspective, the process is shown for the Nexus One in Figure 6.14. Note that the 74-byte packet from the client actually triggers the server to initiate the connection

teardown by sending the first [FIN, ACK] packet.

1365.53550	84.231.182.229	130.233.195.85	TCP	74 [TCP segment of a reassembled PDU]
1365.53858	130.233.195.85	84.231.182.229	TCP	66 http > 44705 [FIN, ACK] Seq=451 Ack=811
1365.59552	84.231.182.229	130.233.195.85	TCP	66 44705 > http [FIN, ACK] Seq=811 Ack=452
1365.59556	130.233.195.85	84.231.182.229	TCP	66 http > 44705 [ACK] Seq=452 Ack=812 win=

Figure 6.14: Successful client-initiated termination of WebSocket connection

It was discovered that all runs that failed were terminated by the client due to the *close timeout* interval being set to 60 seconds. The only exception was the Lumia 820, which seems to be affected by the browser support. The disconnection would often occur during a HB exchange, where the client would respond with a special packet indicating that it wished to terminate the connection. For example, the Nexus One, iPhone 4S, and Galaxy S3 would each respond with a 72-byte packet rather than the usual 75-byte PONG message. On the Lumia 820, this termination response was a 60-byte packet. These responses would inform the server to immediately terminate the connection. This is illustrated in Figure 6.15 where the HB interval is 30 seconds, 1 second beyond the optimal successful interval.

291	291.585223	130.233.195.85	188.238.142.118	TCP	59 [TCP segment of a reassembled PDU]
296	292.282486	188.238.142.118	130.233.195.85	TCP	63 [TCP segment of a reassembled PDU]
297	292.282528	130.233.195.85	188.238.142.118	TCP	54 http > 49191 [ACK] Seq=271 Ack=541
318	322.285410	130.233.195.85	188.238.142.118	TCP	59 [TCP segment of a reassembled PDU]
323	322.499657	188.238.142.118	130.233.195.85	TCP	60 [TCP segment of a reassembled PDU]
324	322.499681	130.233.195.85	188.238.142.118	TCP	54 http > 49191 [ACK] Seq=276 Ack=547
335	322.503283	130.233.195.85	188.238.142.118	TCP	54 http > 49191 [FIN, ACK] Seq=276 Ac
336	322.520441	188.238.142.118	130.233.195.85	TCP	54 49191 > http [ACK] Seq=547 Ack=277
337	322.525453	188.238.142.118	130.233.195.85	TCP	63 [TCP segment of a reassembled PDU]
338	322.525466	130.233.195.85	188.238.142.118	TCP	54 http > 49191 [ACK] Seq=277 Ack=556
339	322.527293	188.238.142.118	130.233.195.85	TCP	54 49191 > http [RST, ACK] Seq=556 Ac

Figure 6.15: Premature client-initiated termination of WebSocket connection on Lumia 820.

Strangely, even after sending the 60-byte packet, the Lumia also sends the usual 63-byte PONG message, indicating its willingness to maintain the connection. However, by that point the server has already closed the connection and a RST message is sent alerting the client that the connection is no longer active. This reveals that heartbeat mechanism still operates independently from whatever caused the browser to initiate the disconnection. On the other devices, the 60 second close timeout was responsible for closing the connections.

The further implications of these results will be discussed in Chapter 8. For now, let us continue to the results for the Server-Sent Events measurements.

Chapter 7

Server-Sent Events Results

The results for all sets of Server-Sent Event (SSE) tests will be discussed now. Each device will be examined and analyzed using multiple browsers in terms of power consumption and performance in maintaining the connection. A better understanding of how the mobile device behaves and performs using an EventSource connection was the goal of these measurements. First, let us examine some of the key factors that played a significant role in obtaining these results.

7.1 Initial Results

As in the WebSocket measurements, an application-level keep-alive mechanism was not necessary to maintain the connection for long periods of time, provided that the server was configured correctly. However, the ability to accomplish long-lasting runs depended upon the browser support as well. The Node.js HTTP server possesses a default timeout value of 120 seconds for all connections, which is a standard value in many servers. Since the majority of measurements were made using this default server configuration, a keep-alive mechanism was required to maintain the connection. Events were sent by the server within the timeout interval acting as heartbeats, ensuring that the connection would remain active. Therefore, the event update interval (EUI) is the critical variable in these measurements, which is shown to have a significant effect on the overall energy consumption of the mobile devices.

Following the first set of measurements, it was seen that an interval exceeding 120 seconds would not keep the original connection alive, and thus events were not received and updated properly within the browser. This was caused by the default HTTP timeout on the server itself and was re-configured to be an unlimited value. After removing the timeout constraint,

certain devices and browsers were able to maintain the connection for longer periods without an event update acting as a keep-alive.

There are essentially two sets of measurements and results: those that are constrained by the 120 second HTTP timeout value and those that are not constrained by any timeout values. They will each be examined separately.

- **Short Event Update Intervals:** 5, 15, 30, 60, 119, 120 seconds.
- **Long Event Update Intervals:** 5, 15, 30, 60, 600 minutes.

7.2 Key Findings

Some of the key findings from these measurements are summarized here now:

- An application-level keep-alive mechanism is not necessary to keep the EventSource connection alive, granted that server timeout values are not set.
- A TCP keep-alive mechanism is used during periods of inactivity of more than 60 minutes. On the Elisa 3G network, the mobile device sends these keep-alives every 60 minutes.
- When HTTP server timeout values are present, an event must be sent by the server within that interval to preserve the connection.
- The lower the EUI, the more active the device, and the more energy will be consumed.
- With 120 second timeout values active, an EUI of 119 seconds or less must be used to successfully maintain the original connection.
- Without timeout values present, the EUI can be extended to far longer periods of 15, 30, and 60 minutes. The longest successful EUI tested was 600 minutes or 10 hours.
- Browser support is the crucial factor for maintaining the connection for prolonged periods of time.
- The only successful browser to maintain the connection indefinitely while updating properly was Opera Mobile.
- Most browsers support SSE activity correctly while the screen is on, but not when the screen times out.

- Upon screen timeout, the connection is eventually terminated by the mobile device and updates are not processed correctly within the browser. (For all browsers except Opera Mobile)
- When the screen is reactivated, a new connection must be established and updates are then successfully pushed from the server once again.
- If parameters such as `Last-Event-ID` were to be leveraged during reconnection, the updates could resume successfully with the proper count being displayed.
- However, the mobile device would still not be correctly receiving push updates from the server in real-time while in the idle state, which violates the principle of push messaging.

7.2.1 Methodology

All devices used a screen timeout of **60 seconds** to ensure uniformity for all test runs. Following the screen timeout, events would continue to be sent by the server at specified intervals. The goal was to make sure that the phone was processing these events correctly in the browser while the device was in a low-energy state. After a suitable length of time, the phone was unlocked and the browser was examined to see if the updates had been processed correctly and were still actively updating. This was determined by examining the “Update Number” counter variable used to measure continuity. If the updates had stopped, were not the correct number, or were no longer being updated, the run was considered unsuccessful.

In total, more than 90 tests were conducted. Each run was evaluated according to several criteria, including the average power, average current, and estimated battery life. Additionally, the *active connection time* to maintain the event-stream was determined. The active connection time (ACT) is the total time between the first SYN packet of the active TCP connection, until the [FIN, ACK] packet that terminates that connection. This value can be seen as precisely accurate in comparison to the WebSocket disconnection times which were not precise.

7.3 Long Event Interval Results

Long-lasting runs with EUIs that exceeded 120 seconds will be discussed first. The server timeouts were disabled to allow for unrestrained persistent connections. It has been discovered that Android devices using Opera Mobile

are able to maintain an EventSource connection using an EUI of 10 hours without a problem. Events were processed correctly in the browser even after this long idle period. However, Firefox, Chrome, Safari, and the Android browser were not able to match this level of performance. With Firefox, no events were properly updated or processed following the screen timeout. With Chrome, Safari, and the Android browser, events could be processed and updated correctly up to a certain limit, usually between 5-10 minutes. At this point, the device would close the connection and the server would stop sending events. Therefore, the only two successful devices were the Nexus One and Galaxy S3 using Opera with their results shown in Table 7.1. All of the longest measurements were conducted on the Elisa 3G network only.

Device	EUI (mins)	Time (mins)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)
Nexus One	5	31.23	122.48	33.08	42.32
Nexus One	15	30.64	82.63	22.32	62.73
Nexus One	30	30.84	75.66	20.43	68.51
Nexus One	60	61.05	53.74	14.40	97.23
Galaxy S3	5	31.26	86.02	23.11	90.88
Galaxy S3	15	30.78	92.33	24.80	84.66
Galaxy S3	30	31.00	74.30	19.96	105.20
Galaxy S3	60	60.40	52.06	13.99	150.14
Galaxy S3	600	957.61	33.87	9.10	230.83

Table 7.1: Longest runs using Opera Mobile - 3G UMTS Elisa

7.3.1 5 Minutes

With events being sent every 5 minutes, the Nexus One performs reasonably well by consuming approximately 122 mW, suggesting around 42 hours of battery life. The Galaxy S3 consumed much less energy at 86 mW with an estimated 90 hours of battery life. It is important to highlight that battery estimates are made using the Monsoon Power Monitor software, taking into account each device's native battery capacity. In this case, 1400 mAh for the Nexus One and 2100 mAh for the Galaxy S3. This is the reason we see such a disparity between the devices, so it is important to look at each device individually to notice the actual energetic improvements. Overall, the 5 minute event interval is acceptable, but should be extended to lengths of

15 minutes or greater to truly gain considerable energy savings.

7.3.2 15 Minutes

In this case, the battery life improved by 20 hours on the Nexus One, consuming approximately 40 mW less than the 5 minute run. This is a significant improvement and could be considered an appropriate and sustainable interval for this device at 62 hours of expected battery life. On the Galaxy S3, the average power consumption actually increased by 6 mW. This is likely due to the fact that other background services such as push email were running at the time and used a data connection during the run. These processes were not easy to disable entirely, so they were left operational. Their presence remained consistent throughout all runs. Regardless, the Galaxy S3 still promised an impressive 84 hours of battery life using this interval, which is quite a sustainable value.

7.3.3 30 Minutes

This longer EUI demonstrated a further increase in expected battery life for both devices, at 68 and 105 hours respectively. Interestingly, the average power consumed for both mobiles was around 75 mW. Due to the capacity difference, we see the Galaxy S3's battery estimates to be much higher than the Nexus One.

7.3.4 60 Minutes

For this interval, both devices consumed approximately 53 mW of power. It is interesting to see that the devices start to converge in terms of overall energy efficiency as the EUI is increased. Since the devices are spending more time in the idle state, this is logical. Battery life estimates are between 97-150 hours for both devices, which is an excellent value.

7.3.5 10 Hours

The longest successful EventSource connection tested used an EUI of 10 hours on the Galaxy S3. The run lasted for a total of 15 hours and 57 minutes and the only event sent by the server was correctly received, processed, and updated within the browser. Since the device was almost exclusively in the idle state, the battery life estimates are impressive at 230 hours based on average power consumption of 33 mW. The most encouraging result is that the EventSource connection was able to last for such a long period. This

suggests that the Elisa 3G network does not possess firewalls or NATs that have timeout values that would hinder these types of long-lived connections. This may vary per operator but the potential still exists for mobile devices to enjoy the benefits of long-lasting, low-cost EventSource connectivity.

7.3.5.1 Device Behavior

In order to sustain the connection for this longer period, TCP keep-alive messages were sent from the mobile to the server precisely every 60 minutes. This was done for two TCP connections, so two keep-alive messages were sent each hour by the device. Since the server responded with a keep-alive ACK each time, the device continued to keep the connection open. This repeated itself successfully for almost 16 hours. Even after the event update was pushed by the server after 10 hours, the TCP keep-alive mechanism continued. Thus, it can be said that the TCP keep-alive interval is set to 60 minutes on the Elisa 3G network. If the server did not respond with an ACK, the device would send a few additional keep-alive probes before eventually disconnecting.

7.3.6 Event Update Exchanges

It took approximately 2 seconds for the mobile device to wake up and receive the event from the server after being idle. This indicates that the device was spending time in the IDLE state, where it can still retain its IP address without a radio control connection. The entire event update exchange lasted less than 3 seconds in all cases. The mobile device would receive the event, respond with an ACK, update the browser, and go back to sleep. An event update triggered 8-10 seconds worth of tail energy in both devices using the Elisa 3G network. This means there is about 6-8 seconds of unnecessary drain, but this value is not so significant provided that the EUI is long enough to be efficient.

7.4 Short Event Interval Results

The next set of results were made using the default HTTP keep-alive timeout setting of 120 seconds. This is a common value that is used in many servers, firewalls, and NATs today. Because of this 120 second restraint, the optimal EUI became 119 seconds in order to keep the EventSource connection alive. An EUI exceeding 120 seconds would cause the server to drop the connection and to stop sending event updates. The goal of the results are to reveal the

energetic effect that this lower EUI has on the mobile device. Additionally, each device and browser performance will be examined in greater detail. The overall results are shown in Table 7.2.

Device	Browser	Connection	EUI (secs)	Outcome
Nexus One	Firefox	3G Elisa	5	Fail on screen timeout
Nexus One	Opera	3G Elisa	119	Success (38+ min)
iPhone 4S	Safari	3G Elisa	119	Could not maintain past 682 sec.
iPhone 4S	Chrome	3G Elisa	5	Fail on screen timeout
Galaxy S3	Firefox	3G Elisa	5	Fail on screen timeout
Galaxy S3	Firefox	4G LTE	5	Fail on screen timeout
Galaxy S3	Opera	3G Elisa	119	Success (38+ min)
Galaxy S3	Opera	4G LTE	119	Success (18+ min)
Galaxy S3	Chrome	3G Elisa	60	Could not maintain past 426 sec.
Galaxy S3	Chrome	4G LTE	119	Could not maintain past 623 sec.
Galaxy S3	Android	3G Elisa	115	Could not maintain past 397 sec.
Galaxy S3	Android	4G LTE	119	Could not maintain past 362 sec.

Table 7.2: Overall results for all devices using optimal EUI across all browsers.

7.4.1 Results for Nexus One

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
A1	5	371.4	380.16	553.08	149.37	9.37	Success
A2	15	373.96	383.35	364.63	98.48	14.22	Success
A3	30	394.83	409.25	271.51	73.33	19.09	Success
A4	60	410.66	419.24	214.54	57.94	24.16	Success
A6	120	364.42	401.47	176.79	47.75	29.32	Fail. Only 2/3 updates received correctly.
A8	119	2291.72	2305.74	102.87	27.78	50.39	Success. 38+ mins.

Table 7.3: Nexus One - Opera Mobile - 3G UMTS Elisa

The Nexus One using Opera Mobile proved to be one of the most energy efficient devices, while also behaving correctly by maintaining the connection successfully. In this case, average power across the runs ranged from 102 - 553 mW, depending on the EUI and length of the run. As in all other cases,

when the EUI was increased, the device was able to enter an IDLE state more frequently, which significantly increases energy savings. The ideal EUI is 119 seconds and is able to maintain a persistent SSE connection while consuming an average of 102 mW for 38+ minutes. The Monsoon software estimates more than 50 hours of battery life with this configuration, which is significantly lower than extended runs with a greater EUI.

7.4.2 Results for iPhone 4S

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
A1	5	315.98	326.99	654.91	173.62	8.25	Success
A2	15	527.19	538.15	404.44	107.22	13.36	Success
A3	30	317.5	327.39	416.93	110.53	12.96	Success
A4	60	427.95	439.00	314.45	83.36	17.18	Success
A5	120	362.29	521.89	261.67	69.36	20.64	Fail. Only 1/2 update received correctly.
A6	119	683.79	828.19	225.51	59.78	23.95	Fail. Only 5/6 updates received correctly.
A9	90	680.10	836.92	245.43	65.06	22.01	Fail. Only 7/9 updates received correctly.

Table 7.4: iPhone 4S - Safari - 3G UMTS Elisa

Both Safari and Chrome were unable to successfully maintain the SSE connection for long periods of time. Chrome was not able to maintain the connection at all when the screen timed out. Therefore, the results for Chrome testing are not shown. For Safari, the connection could last for up to 11.3 minutes before the mobile would disconnect. This occurred regardless of the EUI, suggesting that there may be browser code within Safari or the iOS operating system that terminates idle connections after a certain period. An EUI of 119 seconds proved optimal for the iPhone in terms of energy performance. The average power consumption ranged from 225 - 654 mW. Without being able to maintain the connection for longer periods (i.e. 30+ minutes), it is unclear how much battery life could be expected under such circumstances.

7.4.3 Results for Galaxy S3

Using the Galaxy S3, it was possible to test the SSE connection on 4 different browsers: Firefox, Chrome, Opera, and Android browser. Similar to the

Nexus One results, Firefox was unable to maintain the connection once the screen timed out. Therefore, the three remaining browsers will be examined. It was possible to test these using both 3G and LTE connections, to determine if the network had a great effect on the outcome.

7.4.3.1 Opera Mobile Browser

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
A1	5	389.57	406.98	1001.78	269.86	7.78	Success
A2	15	381.51	392.86	636.96	171.58	12.24	Success
A3	30	395.19	404.16	539.45	145.32	14.45	Success
A4	60	378.95	389.18	367.19	98.91	21.23	Success
A19	121	123.62	212.55	460.42	124.04	16.93	Fail. No updates received correctly.
A9	120	360.45	411.36	363.78	98.00	21.43	Fail. Only 2/3 updates received correctly.
A23	119	2287.69	2295.69	156.71	42.21	49.75	Success. 38+ mins.

Table 7.5: Galaxy S3 - Opera - 3G UMTS Elisa

The Opera Mobile browser proved to be the only successful browser able to maintain the SSE connection persistently for longer periods of time. Again, the optimal EUI was 119 seconds. An EUI of 120 seconds would process 2 updates successfully before a server-initiated disconnect would occur exactly 120 seconds after the second update. For EUI values higher than 120, no updates were processed at all. The Monsoon software indicates potential battery life of around 49 hours, suggesting that maintaining the connection with this keep-alive frequency could be sustainable for some users. However, this estimate is only taking into account an idle phone with no display activity, which is not a likely real-world scenario for an average mobile user.

The average power consumption ranged from 156 - 1001 mW. This is higher than the Nexus One primarily due to the larger display and energy needed to power such a device with advanced hardware and software. In the optimal run of 38+ minutes, the Galaxy consumed approximately 156 mW of average power, compared to 102 mW on the Nexus One. While both of these values are fairly efficient, it would be far more beneficial to maintain the connection with an EUI that greatly exceeded 120 seconds. Battery life could be improved from 49 hours to up to 230 hours if this were the case.

Using LTE, the performance and behavior is similar to its 3G counterpart. Opera was successfully able to maintain the connection and EUIs exceeding 120 seconds failed to update properly after 1-2 events. The power

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
D1	5	305.24	315.43	1043.43	279.60	7.51	Success
D2	15	317.45	329.24	838.40	224.66	9.35	Success
D3	30	333.70	351.91	555.75	148.92	14.10	Success
D4	60	303.96	315.70	413.63	110.83	18.95	Success
D5	120	240.11	328.50	280.58	75.19	27.93	Fail. Only 1 update received correctly.
D6	119	1080.53	1093.50	276.77	74.16	28.32	Success. 18+ mins.

Table 7.6: Galaxy S3 - Opera - 4G LTE

consumption ranged from 276 - 1043 mW, which is slightly higher than 3G. Particularly for the optimal run of 18+ minutes, it appears that 276 mW is significantly higher than 156 mW in the case of the optimal 3G run. However, given that there is a 20 minute discrepancy between the two runs, we cannot compare these two so closely. The important thing to note is that Opera Mobile functioned correctly on both 3G and LTE for extended periods of time and was able to successfully maintain the SSE connection persistently using optimally-timed event updates.

7.4.3.2 Chrome Browser

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
B1	5	119.79	185.61	990.94	266.97	7.87	Fail. 23 updates received correctly.
B2	15	168.24	211.28	753.95	203.11	10.34	Fail. 11 updates received correctly.
B3	30	243.96	281.99	518.82	139.78	15.02	Fail. 8 updates received correctly.
B5	60	425.53	500.67	298.42	80.40	26.12	Fail. 7 updates received correctly.
B6	90	303.63	359.84	393.64	106.05	19.80	Fail. 3 updates received correctly.

Table 7.7: Galaxy S3 - Chrome - 3G UMTS Elisa

Using Chrome, the connection was unable to be maintained indefinitely beyond 7 minutes. Regardless of the EUI used, events were not properly updated within the browser while the screen was off. After several updates, usually within 3-12 seconds of an event being received, the mobile device would terminate the connection. This is likely due to the browser support and how Chrome handles mostly idle connections. However, without being able to access or examine the browser code itself, it is difficult to pinpoint

the exact reason for this behavior. It appears that modifying the EUI does have an effect on the active connection time. In this case, the optimal EUI of 60 seconds allowed the connection to last 7.1 minutes, while normally runs terminated usually before 5 minutes. The power consumption ranged from 298 - 990 mW.

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
E1	5	118.89	184.60	1025.66	274.85	7.64	Fail. 23 updates received correctly.
E2	15	169.50	224.49	835.36	223.86	9.38	Fail. 11 updates received correctly.
E3	30	244.26	284.57	586.74	157.24	13.36	Fail. 8 updates received correctly.
E4	60	427.17	470.95	342.29	91.73	22.89	Fail. 7 updates received correctly.
E5	120	482.98	539.43	280.91	75.28	27.90	Fail. Only 1 update received correctly.
E6	119	622.22	667.34	246.39	66.02	31.81	Fail. 5 updates received correctly.

Table 7.8: Galaxy S3 - Chrome - 4G LTE

Using Chrome over LTE proved to be similar in behavior, except that the runs were able to persist for slightly longer than 3G. Using an EUI of 119 seconds, the device was able to last 10.3 minutes. However, the mobile device still terminated the connection for reasons not entirely known. The power consumption ranged from 246 - 1025 mW. When comparing similar runs on 3G and LTE, it is clear that LTE consumes more power on average, as expected. The packet behavior was virtually identical to 3G runs, where client-initiated disconnections occurred after a certain amount of time passed. It is not clearly distinguishable as to what causes the disconnection, as some runs are longer than others, and this seems to be affected by the EUI.

7.4.3.3 Android Browser

The Android browser was also unable to maintain the connection indefinitely. The longest connection lasted for 6.6 minutes using an EUI of 115 seconds. The overall power consumption ranged from 228 - 576 mW. It is notable that using the Android browser consumes less energy than Opera and Chrome. This is especially apparent during the cases of 5 second EUI, where the average power exceeded 1000 mW for both Opera and Chrome. Using Android, the average power was almost half that value at 576 mW. If the connection were able to be maintained indefinitely, it seems that the Android browser

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
C2	5	304.48	310.22	576.86	155.39	13.51	Success
C3	15	305.51	319.71	433.02	116.64	18.00	Success
C4	30	307.91	315.19	347.92	93.71	22.41	Success
C5	60	363.43	395.04	244.07	65.74	31.94	Fail. 5/6 updates received correctly.
C6	120	241.71	283.48	278.01	74.89	28.04	Fail. 1 update received correctly.
C7	119	363.31	401.22	237.68	64.02	32.80	Fail. 3 updates received correctly.
C9	115	396.03	407.47	228.50	61.55	34.12	Fail. 3 updates received correctly.

Table 7.9: Galaxy S3 - Android Browser - 3G UMTS Elisa

would be the most energy efficient by far. This is likely due to it being the native browser for the device, where the browser developers were able to leverage their knowledge of the inner workings of the Android OS itself. The disconnection behavior was similar to that of Chrome, where updates would eventually stop being processed correctly within the browser, and a client-initiated disconnection would occur.

Run	EUI (secs)	ACT (secs)	Total Run Time (secs)	Avg. Power (mW)	Avg. Current (mA)	Est. Battery Life (hours)	Notes
F1	5	222.01	230.72	1056.66	283.15	7.42	Success
F2	15	361.80	393.43	717.64	192.31	10.92	Fail. 24 updates received correctly.
F3	30	361.46	407.64	434.45	116.42	18.04	Fail. 12 updates received correctly.
F4	60	361.10	399.89	361.85	96.96	21.66	Fail. 6 updates received correctly.
F5	120	360.18	397.31	247.22	66.24	31.70	Fail. Only 2 updates received correctly.
F6	119	361.97	396.02	306.96	82.25	25.53	Fail. 3 updates received correctly.

Table 7.10: Galaxy S3 - Android Browser - 4G LTE

The behavior using LTE was similar to 3G with the total active connection time not exceeding 6 minutes. As in the case of 3G, the Android browser proved to more energy efficient than Opera and Chrome. This is particularly notable during runs with EUI of 5 and 15 seconds, where the screen and mobile is in an active state more frequently. However, as the EUI became less frequent, the overall efficiency advantages are less noticeable. The overall power consumption ranged from 247 - 1056 mW, which is very similar to that

of Chrome using LTE. Approximately 1-5 seconds after an update being received, the mobile device would eventually terminate the connection, right around 6 minutes, regardless of the EUI. Given that the Opera browser was able to maintain the connection indefinitely, it seems that this is a browser support issue and not network related.

7.5 Heartbeat Analysis

Each event sent by the server triggered an ACK response from the mobile client during its sleep state. These responses could also be classified as the heartbeat period, since the event-update mechanism is being used as a keep-alive. The heartbeat period includes the head and tail energies incurred from signaling, along with the data transfer itself. Therefore, studying the heartbeat patterns across all devices was done to see if there are any major discrepancies or behaviors that stand out.

Device	Browser	Network	EUI (secs)	HB Time (secs)	Avg. Power (mW)	Avg. Current (mA)
Nexus One	Opera	3G UMTS	119	7.98	471.18	127.25
iPhone 4S	Safari	3G UMTS	119	4.81	577.78	153.17
Galaxy S3	Opera	3G UMTS	119	10.21	673.91	181.54
Galaxy S3	Opera	4G LTE	119	11.27	861.62	230.88
Galaxy S3	Chrome	3G UMTS	60	9.98	685.09	184.55
Galaxy S3	Chrome	4G LTE	60	11.89	866.91	232.30
Galaxy S3	Android Browser	3G UMTS	119	9.46	369.79	99.60
Galaxy S3	Android Browser	4G LTE	119	11.92	772.81	207.08

Table 7.11: Heartbeat period analysis for optimal EUI runs with screen off.

All heartbeat periods are not equal, even within the same test. Many factors can affect the overall power consumption during these periods, such as the amount of data to transfer, throughput, signal strength, background computation, among other factors. However, generally the heartbeat exchanges occur within a consistent time period that is correlated to the device and the network itself. For each device, heartbeat periods were examined to determine if there are any significant differences between browsers and/or devices. From this analysis, the following heartbeat durations were determined for each device:

- Nexus One: 8-10 seconds.
- iPhone 4S: 4.8 - 7.5 seconds.

- Galaxy S3: 9-12 seconds.

Interestingly, the iPhone was able to rapidly receive, process, and update the events in less than 5 seconds in some cases. This is likely because the iPhone appears to remain in a PCH state that consumes more power than the rest of the devices that remain in IDLE states. The iPhone peaks at around 900 mW before stabilizing and finishing the transfer at around 550 mW before returning back to PCH. This is shown in Figure 7.1.

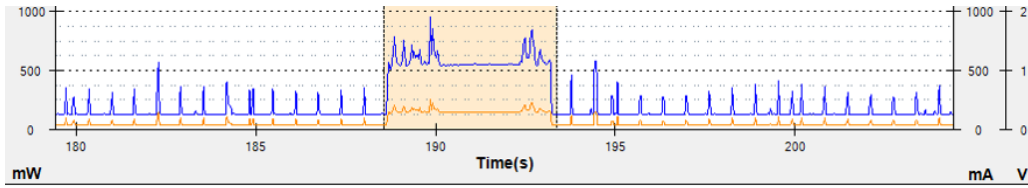


Figure 7.1: iPhone Heartbeat using Safari - 3G UMTS - 4.81 seconds

Another interesting case to examine is the difference in overall efficiency between the Android browser and Opera/Chrome in the Galaxy S3. The Android browser is clearly more efficient during its HB interval over 3G, consuming around half the power of Opera and Chrome. The HB behavior is shown in Figures 7.2 - 7.4. The Android transfer peaks at around 600 mW and stabilizes at 400 mW. For the others, the majority of the transfer consumes approximately 800 mW. Opera and Chrome perform almost identically during HB periods. It is interesting to note that native browser integration clearly has an energetic advantage over third-party browsers.

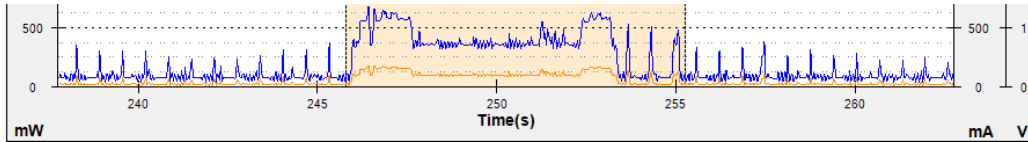


Figure 7.2: Galaxy S3 Heartbeat using Android - 3G UMTS - 9.46 seconds

7.6 Packet Behavior

The packet traces were analyzed to get a better understanding of what was taking place between the mobile device and the server, particularly during premature disconnections. The event-stream is created when the client requests the EventSource URL from the server. Following this, the server is able to stream events at its own will to the client. The server sends the data

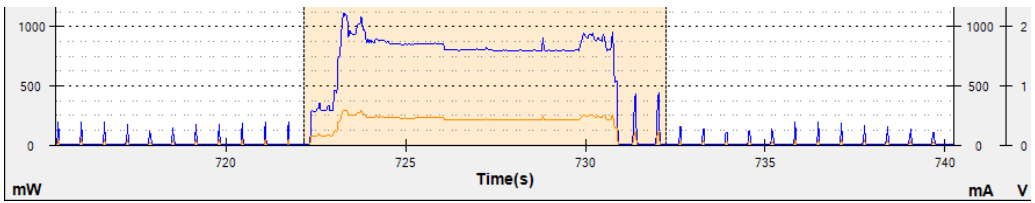


Figure 7.3: Galaxy S3 Heartbeat using Opera - 3G UMTS - 10.21 seconds

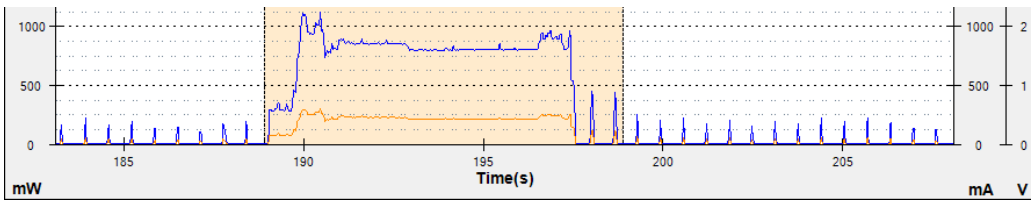


Figure 7.4: Galaxy S3 Heartbeat using Chrome - 3G UMTS - 9.98 seconds

in the format outlined in Chapter 4. The mobile device responds with an ACK to each event received by the server, but otherwise sends or requests nothing else. In these experiments, the event update process is displayed in Figure 7.5. Note that the server IP address is **130.233.195.85**.

970.163761	130.233.195.85	85.76.185.93	HTTP	85 Continuation or non-HTTP traffic
972.109651	85.76.185.93	130.233.195.85	TCP	66 14651 > http [ACK] Seq=564 Ack=97
972.109698	130.233.195.85	85.76.185.93	HTTP	134 Continuation or non-HTTP traffic
972.473186	85.76.185.93	130.233.195.85	TCP	66 14651 > http [ACK] Seq=564 Ack=10

Figure 7.5: Event Update / HB exchange between server-client.

When the server timeout was set to 120 seconds, an event update had to occur within this interval for the connection to be maintained. Otherwise, the server would initiate the disconnection. Using EUIs of 119 seconds or less for all browsers except Opera, the mobile device would eventually terminate the connection. The termination was achieved by sending a [FIN,ACK] packet for the active event-stream TCP connection. Further event-streaming would thus require the creation of a new EventSource connection, which was not initiated while the device remained in the idle state. This causes the device to miss out on event updates entirely, which defeats the purpose of persistent push messaging. Therefore, it is important to determine a feasible means by which the EventSource connection can be maintained persistently and efficiently over these other browsers, without being terminated by short timer values.

Chapter 8

Discussion

This section will discuss the overall implications of the results for the WebSocket and Server-Sent Events measurements. One of the major goals of this thesis has been to determine the overall readiness of these protocols for mobile usage. The emphasis will be on discussing their current state of usability from an energetic and performance perspective, followed by critical factors that must be taken into account.

A search through scholarly papers and peer-reviewed document databases revealed that mobile measurements of WebSockets and Server-Sent Events is almost non-existent. One set of measurements by Mandyam and Ehsan [33] included a comparison of the battery drain between AJAX and WebSocket using a 3G-connected laptop. However, these measurements were analyzed using the battery percentage indicator only and do not provide extensive results for the mobile readiness of WebSockets or SSEs across several platforms and browsers. Therefore, it is believed that the measurements made in this thesis are unique at this time, with the hope that they can be of use to developers and standardization bodies such as W3C.

8.1 Performance Implications

The results have revealed that both WebSockets and Server-Sent Events are suitable and ready for mobile use under certain conditions. By testing across the three most popular smartphone platforms, we have gained a better understanding of current WS and SSE performance and what affects the energy consumption and longevity of these connections. Let us now examine what contributes to the successful mobile performance of both WebSocket and Server-Sent Events.

The most critical factors to maintain a long-lasting, energy-efficient Web-

Socket and EventSource connection on a compatible mobile device include:

1. **Browser support.** This is the most important factor, as this will ultimately determine if successful performance is even possible. By testing across 6 of the most widely used mobile browsers, it is clear that the performance varies and there is room for improvement. For example, Opera Mobile was the only browser capable of successfully updating and maintaining an EventSource stream indefinitely without disconnecting. While Chrome, Safari, and the Android browser also received and updated events correctly during idle mode, they would eventually terminate the connection, rendering long-lasting push functionality impossible. Additionally, Internet Explorer 10 mobile does not support SSE at all. This demonstrates that browser support varies widely and must be improved and updated to make these types of connections more compatible with mobile devices.
2. **Supporting idle connections after the screen turns off.** The mobile device must be able to maintain connectivity when the display is off. For example, the iPhone 4S immediately terminated the WebSocket connection following a screen-lock. In the case of Firefox and SSEs, no events are processed and updated within the browser after screen timeout occurs. This type of behavior severely limits the potential usage of these persistent connections and must be addressed by updating the affected browsers and/or operating system platforms.
3. **Proper server configuration.** The server must be optimized to fit the application's needs and must take into account usage by mobile devices and their inherent energy constraints. Timeout values must be set accordingly, especially for use cases that require long-lasting connections with frequent periods of inactivity.
4. **Appropriate keep-alive interval and strategy.** Due to timeout values found within the server, middleboxes, and the browser itself, keep-alive mechanisms are often necessary to maintain the connection indefinitely. The heartbeat intervals must be carefully configured to meet timeout value thresholds, while also considering that the frequency of these heartbeats greatly affects mobile energy consumption.
5. **Network operator configuration.** The mobile network configuration plays a significant role in the success of these connections. The T1/T2/T3 timer values affect the amount of tail energy produced, especially during heartbeat exchanges. The support of Fast Dormancy

can be an asset during these exchanges, as the device can return to an idle state far faster if it knows it will not be receiving any more data. The TCP timeout values also play a role in how long these connections will wait before sending TCP keep-alive probes. Furthermore, the presence of firewalls, proxies, and NATs within the mobile network may be most critical of all. The timeout values here can potentially prevent long-lasting connections without the use of frequent heartbeats. In the case of 3G Elisa, it has been determined that no middlebox timeouts are preventing these long-lasting connections. However, this will certainly vary widely depending on the operator and country, so these types of results cannot always be expected.

If these conditions are met and given careful consideration, it is definitely possible to use both WebSocket and Server-Sent Events persistently and efficiently for a wide variety of mobile applications.

8.1.1 Application Types

The type of application used will greatly affect the overall server configuration and the type of connectivity required. WS and SSEs can be used as transport protocols within native, hybrid, and web applications for a variety of purposes. For instance, a periodic weather updating service may only need unidirectional functionality and would be best suited for Server-Sent Events. Weather updates could be sent whenever new data is available while a persistent connection runs in the background. However, a multiplayer game may require sporadic bidirectional communication and is ideal for WebSockets. There are many uses for both of these technologies and they should be implemented according to the application requirements.

8.1.2 Server Configuration

Configuring the server properly is extremely important when it comes to efficient mobile performance. Timeout values such as the *close timeout* must be identified and possibly lengthened considerably in order to create a lasting connection that requires infrequent heartbeat messages. During these measurements, I learned firsthand that not configuring these timeout values properly will drastically limit the longevity and performance of the connection. One must be aware of all factors that can affect the connection and this requires a detailed understanding of the server configuration parameters. Therefore, this is something all developers should recognize when implementing WS and SSE-based mobile applications.

8.1.3 Heartbeat Interval Performance Trade-off

In general, the less frequent the heartbeat interval, the more energy is saved in the mobile device. The measurement results confirm the conclusions of Haverinen et al. [22], namely that keep-alive intervals of a few minutes or less will lead to unsustainable battery life performance. Therefore, the heartbeat interval for any always-on mobile application must be extended to greater lengths to be truly usable. Our results have demonstrated that the underlying WS and SSE connections can be maintained without the use of heartbeats or event updates in certain scenarios. While longer heartbeat intervals of 10 hours may seem to be optimal from an energetic perspective, there is a crucial performance trade-off that must be considered.

It has been tested and confirmed that when a mobile WS connection is abruptly severed (ie. when power source is lost), the server will not be informed of this disconnection until the next heartbeat period. Therefore, even though the device is disconnected, the server continues to send all messages to the mobile as if it were still connected. This can be a significant waste of network resources, especially if the server is handling many clients. The server is unable to determine which clients still actively need to be allocated network resources. This serves as a reminder that keep-alives are also used to check on the status of a client and are not just for maintaining the connection indefinitely.

Therefore, a fundamental performance trade-off exists which must be taken into account. The heartbeat interval should be set long enough to allow for energetically efficient performance, while still frequent enough for the server to check if the client still needs network resources. This should be determined based on the application type and needs. Our results reveal that heartbeat intervals of 60 minutes or greater can lead to more than 150 hours of estimated battery life in some cases. However, intervals of 5 minutes or less are not sustainable for long-term mobile use. This suggests that an interval of at least 15 or 30 minutes should be used for most persistent mobile applications to ensure maximum energy performance, while still retaining the ability to check if the connection is still needed.

Chapter 9

Conclusions

This thesis has examined the energetic efficiency and overall performance of WebSocket and Server-Sent Events connectivity using modern mobile devices and browsers. The results have shown that performance is not uniform and varies greatly depending upon the browser and network configuration. However, under certain conditions, performance can be long-lasting and very efficient energetically, meaning that both of these protocols can be used for mobile web applications. Further improvements in device and browser support for WS and SSEs would greatly enhance the future success of these applications. In this section, future work opportunities will be discussed in order to potentially improve the mobile implementations of these communication technologies. This will be followed by the final conclusions of this work.

9.1 Future Work

Let us now examine some possibilities for future work in this area.

9.1.1 Mobility

One aspect of that these measurements did not address is the mobility of the device. All measurements were carried out in the same location, so handovers and cell-switching behavior was not accounted for. Additionally, losing service coverage entirely and then re-acquiring service is a scenario that has not been properly considered. In real-world usage scenarios, connectivity can be sporadic, so it is key to understand how mobility affects the performance and longevity of these connections.

9.1.2 Additional Testing

Testing across different application types that use WS and SSEs could be relevant and useful. For example, testing a WebSocket-based online game or SSE-based email system could highlight the differences in performance and behavior that application types create. Also, using a server other than Node.js and Socket.io could prove to be beneficial to understand how other libraries and server frameworks affect the overall performance.

An in-depth comparison of WS/SSE vs. AJAX/Comet real-time communication would also be very relevant to prove that there is indeed a tangible difference in efficiency between these methods. If concrete results are obtained in this area, especially if they denote a significant improvement in overall energy efficiency, this would prove to be very beneficial for the user and developer community at large. In general, all mobile testing using WS and SSEs will be valuable at this time while they continue to evolve and undergo their standardization.

Testing IE10-based devices using a fully compatible WS server becomes a priority to determine what needs to be fixed in the browser to extend the length of WS connections. Because Socket.io is not explicitly compatible with IE10, it cannot be assumed that the browser is fully responsible for the poor WS performance results on the Lumia 820. Yet the cause of this should be identified and corrected to ensure that Windows Phone devices can utilize WebSockets as a lasting mobile transport solution.

9.1.3 Connectionless Push

For Server-Sent Events, finding a way to test the *connectionless push* feature would be ideal. However, this would require intimate access to the mobile network environment, along with the ability to configure and test a push proxy to handle the offloaded connection. This is difficult but can be achieved with access to the right network resources. Ultimately, if the push proxy can be implemented correctly, this would serve as the most efficient method of maintaining push connectivity for idle mobile devices.

9.2 Future Potential

The evolution of HTML5 technologies promises great advancements towards the ideal of universal web access across all devices. These technologies are being designed with efficiency in mind and will serve billions of web users in the years to come. Our need for optimal communication protocols is

even greater given the exponential growth in mobile usage and widespread high-speed media consumption. These devices are energy-constrained at the moment and require special consideration in order to meet the needs of the masses.

The W3C is set to release a stable Recommendation version of HTML5 by the end of 2014 [47]. Currently these technologies are being tested for implementability worldwide, including for mobile devices. As more applications begin using WebSockets and Server-Sent Events to meet their real-time communication needs, traditional techniques such as polling and long-polling will become less popular. Mobile browser support will continue to improve, and we can expect that more mobile native, web, and hybrid applications will employ WS and SSEs as the means of transport.

Users and developers alike can expect greater simplicity and efficiency when implementing these technologies that are now native to the browser. This will continue to fuel innovative application design and the optimization of popular applications already in use. Ultimately, the WebSocket is going to become the bidirectional protocol standard of the future with its powerful capabilities. Server-Sent Events gives great promise at becoming an extremely versatile and highly scalable push solution for all devices. Given their optimization and the elimination of all unnecessary overhead, these technologies will provide significant energy savings for mobile devices in the years ahead. As browser support improves and evolves, it is likely that this vision will become a reality.

Bibliography

- [1] 3GPP TS 36.331. E-UTRA; Radio Resource Control (RRC) Protocol Specification, May 2008. Rel. 8, v8.2.0.
- [2] A. BERGKVIST, D. BURNETT, C. JENNINGS, A. NARAYANAN. WebRTC 1.0: Real-time Communication Between Browsers. W3C Editor's Draft, W3C, Mar 2013. <http://dev.w3.org/2011/webrtc/editor/archives/20130322/webrtc.html>. Accessed 01.4.2013.
- [3] ALESSANDRO ALINONE. Comet and Push Technology, 2007. <http://cometdaily.com/2007/10/19/comet-and-push-technology/>. Accessed 07.5.2013.
- [4] ANDREI POPESCU. Geolocation API Specification. W3C Proposed Recommendation, W3C, May 2012. <http://www.w3.org/TR/geolocation-API/>. Accessed 13.6.2013.
- [5] ANSSI KOSTIAINEN ET AL. HTML5 Media Capture. W3C Candidate Recommendation, W3C, May 2013. <http://www.w3.org/TR/html-media-capture/>. Accessed 13.6.2013.
- [6] ARVE BERSVENDSEN. Event Streaming to Web Browsers. *Opera Developers* (2006). <http://dev.opera.com/articles/view/labs-event-streaming-to-web-browsers/>. Accessed 01.4.2013.
- [7] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 280–293.
- [8] BONTU, C., AND ILLIDGE, E. DRX mechanism for power saving in LTE. *Communications Magazine, IEEE* 47, 6 (2009), 48–55.

- [9] BOZDAG, E. AND MESBAH, A. AND VAN DEURSEN, A. A Comparison of Push and Pull Techniques for AJAX. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on* (2007), pp. 15–22.
- [10] CARROLL, A., AND HEISER, G. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (2010), pp. 21–21.
- [11] CHARLAND, A., AND LEROUX, B. Mobile Application Development: Web vs. Native. *Communications of the ACM* 54, 5 (2011), 49–53.
- [12] CHEN, B., AND XU, Z. A framework for browser-based Multiplayer Online Games using WebGL and WebSocket. In *Multimedia Technology (ICMT), 2011 International Conference on* (2011), pp. 471–474.
- [13] CISCO. Cisco Visual Networking Index: Forecast and Methodology, 2012-2017, May 2013. <http://tinyurl.com/mh7t5gx>. Accessed 12.6.2013.
- [14] COLIN J. IHRIG. Server-Sent Events in Node.js, 2012. <http://cjihrig.com/blog/server-sent-events-in-node-js/>.
- [15] CORCORAN, P., MOONEY, P., BERTOLOTTO, M., AND WINSTANLEY, A. View- and Scale-Based Progressive Transmission of Vector Data. In *Computational Science and Its Applications - ICCSA 2011*, vol. 6783 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 51–62.
- [16] ETA DEVICES. Benefits for handset manufacturers. <http://etadevices.com/what-we-do/for-handset-manufacturers/>. Accessed 22.5.2013.
- [17] FABIO BUSATTO. TCP Keepalive HOWTO, 2007. <http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/index.html>. Accessed 07.5.2013.
- [18] GREGOR ROTH. HTML5 Server-Push Technologies, Part 1. *Java.net* (2010). <https://today.java.net/article/2010/03/31/html5-server-push-technologies-part-1>. Accessed 01.4.2013.
- [19] GSM ASSOCIATION. Fast Dormancy Best Practices - Version 1.0, Jul 2011. <http://www.gsma.com/newsroom/wp-content/uploads/2012/03/ts18v10tsgprdfastdormancybestpractices.pdf>.
- [20] GUILLERMO RAUCH. Socket.IO, 2013. <http://socket.io/>. Version used: 0.9.11.

- [21] HÄMÄLÄINEN, H. HTML5: WebSockets. *Aalto University, Department of Media Technology* (2011).
- [22] HAVERINEN, H., SIREN, J., AND ERONEN, P. Energy Consumption of Always-On Applications in WCDMA Networks. In *Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th* (2007), pp. 964–968.
- [23] HEINRICH, M., AND GAEDKE, M. WebSoDa: A Tailored Data Binding Framework for Web Programmers Leveraging the WebSocket Protocol and HTML5 Microdata. In *Web Engineering*, vol. 6757 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 387–390.
- [24] I. FETTE AND GOOGLE, INC. AND A. MELNIKOV. RFC 6455: The WebSocket Protocol, Dec 2011. <http://www.ietf.org/rfc/rfc6455.txt>. Status: PROPOSED STANDARD.
- [25] IAN HICKSON. Offline Web Applications. W3C Working Group Note, W3C, May 2008. <http://www.w3.org/TR/offline-webapps/>. Accessed 13.6.2013.
- [26] IAN HICKSON. The Web Sockets API. W3C Working Draft, W3C, Dec 2009. <http://www.w3.org/TR/2009/WD-websockets-20090423/>. Accessed 16.3.2013.
- [27] IAN HICKSON. Server-Sent Events. W3C Candidate Recommendation, W3C, Dec 2012. <http://www.w3.org/TR/eventsource/>. Accessed 19.3.2013.
- [28] IAN HICKSON. The Web Sockets API. W3C Candidate Recommendation, W3C, Sep 2012. <http://www.w3.org/TR/websockets/>. Accessed 16.3.2013.
- [29] IAN HICKSON. Web Workers. W3C Candidate Recommendation, W3C, May 2012. <http://www.w3.org/TR/workers/>. Accessed 16.6.2013.
- [30] JOYENT, INC. node.js, 2013. <http://nodejs.org/>. Version used: 0.8.18.
- [31] KOLDING, T., WIGARD, J., AND DALSGAARD, L. Balancing Power Saving and Single User Experience with Discontinuous Reception in LTE. In *Wireless Communication Systems. 2008. ISWCS '08. IEEE International Symposium on* (2008), pp. 713–717.

- [32] LIN, H., CHADLI, Y., FOURESTIE, B., AND FAYE, M. Moving the entire service logic to the network to facilitate IMS-based services introduction. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on* (2010), pp. 1–7.
- [33] MANDYAM, G. D., AND EHSAN, N. HTML5 Connectivity Methods and Mobile Power Consumption. *Qualcomm* (2012).
- [34] MAXIMILIANO FIRTMAN. Mobile HTML5, 2013. <http://mobilehtml5.org/>. Accessed 28.2.2013.
- [35] MICHAEL MUKHIN. node.js and socket.io chat tutorial, Sep 2011. <http://psitsmike.com/2011/09/node-js-and-socket-io-chat-tutorial/>.
- [36] MOBILENEWSDAILY. ETA Devices Promises Doubling The Handset's Autonomy, Nov 2012. <http://mobilenewsdaily.net/2012/11/10/eta-devices-promises-doubling-the-handsets-autonomy/>. Accessed 06.4.2013.
- [37] MONSOON SOLUTIONS INC. Power Monitor. <http://www.msoon.com/LabEquipment/PowerMonitor/>. Accessed 22.5.2013.
- [38] NITIN PURI. Flexible, foldable smartphones screens beckon. *ZDNet* (April 2013). <http://www.zdnet.com/flexible-foldable-smartphones-screens-beckon-7000013967/>. Accessed 19.6.2013.
- [39] NURMINEN, J. Parallel Connections and their Effect on the Battery Consumption of a Mobile Phone. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE* (2010), pp. 1–5.
- [40] NURMINEN, J. Energy Efficient Distributed Computing on Mobile Devices. *Distributed Computing and Internet Technology 7753* (2013), 27–46.
- [41] P. LUBBERS AND F. GRECO. HTML5 WebSockets: A Quantum Leap in Scalability for the Web. *SOA World Magazine* (2010).
- [42] PASCAL-EMMANUEL GOBRY. HTML5: Yes, It Will Replace Native Apps-But It Will Take Longer Than You Think. *Business Insider* (January 2012).
- [43] POHJA, MIKKO. Server push for web applications via instant messaging. *J. Web Eng.* 9, 3 (Sept. 2010), 227–242.

- [44] PUPUTTI, K. Mobile HTML5: Implementing a Responsive Cross-Platform Application. *Aalto University, Department of Computer Science and Engineering* (2012). <http://urn.fi/URN:NBN:fi:aalto-201210043228>. Accessed 06.4.2013.
- [45] R. FIELDING, J. GETTYS, T. BERNERS-LEE, ET AL. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, Jun 1999. <http://www.ietf.org/rfc/rfc2616.txt>. Section 8.1: Persistent Connections.
- [46] ROB MUELLER. HTTP keep-alive connection timeouts, Jun 2011. <http://blog.fastmail.fm/2011/06/28/http-keep-alive-connection-timeouts/>. Accessed 09.5.2013.
- [47] ROBIN BERJON ET AL. HTML 5.1. W3C Working Draft, W3C, May 2013. <http://www.w3.org/TR/html51/>. Accessed 12.6.2013.
- [48] STEVE KOVACH. There’s Already A Plan To Fix Your Smartphone’s Terrible Battery Life. *Business Insider* (Mar 2013).
- [49] SWEZEY, R., SHIRAMATSU, S., OZONO, T., AND SHINTANI, T. Intelligent Page Recommender Agents: Real-Time Content Delivery for Articles and Pages Related to Similar Topics. In *Modern Approaches in Applied Intelligence*, vol. 6704 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 173–182.
- [50] TELERIK. HTML5 Adoption Fact or Fiction: Developers Wade Through the Hype. *Kendo UI* (September 2012). <http://www.kendoui.com/surveys/html5-adoption-survey-2012.aspx>. Accessed 28.3.2013.
- [51] TIM GUTHBERTSON - GFXMONK. defer: taming asynchronous javascript with CoffeeScript, 2010. <http://gfxmonk.net/2010/07/04/defer-taming-asynchronous-javascript-with-coffeescript.html#1>. Accessed 07.5.2013.
- [52] W. RICHARD STEVENS. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Professional, 1993. Chapter 23: TCP Keepalive Timer.
- [53] W3SCHOOLS. AJAX Introduction. http://www.w3schools.com/ajax/ajax_intro.asp. Accessed 01.4.2013.
- [54] WESSELS, A., PURVIS, M., JACKSON, J., AND RAHMAN, S. Remote Data Visualization through WebSockets. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on* (2011), pp. 1050–1051.