Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Darshan Gajanan Puranik

Entitled Real-time Monitoring of Distributed Real-time Embedded Systems using Web

For the degree of Master of Science

Is approved by the final examining committee:

Dr. James H. Hill	
Chair	
Dr. Rajeev Raje	
Dr. Arjan Durresi	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. James H. Hill

Approved by: Dr. Shiaofen Fang 03/29/2013
Head of the Graduate Program Date

REAL-TIME MONITORING OF DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS USING WEB

A Thesis

Submitted to the Faculty

of

Purdue University

by

Darshan Gajanan Puranik

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2013

Purdue University

Indianapolis, Indiana

This work is dedicated to my family and friends.

ACKNOWLEDGMENTS

I am heartily thankful to my advisor, Dr. James H. Hill, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

I also want to thank Dr. Rajeev Raje and Dr. Arjan Durresi for agreeing to be a part of my Thesis Committee.

My gratitude is also extended to Mr. Dennis Feiock for helping me with implementation of WebSocket system.

Thank you to all my friends and well-wishers for their good wishes and support. And most importantly, I would like to thank my family for their unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Puranik, Darshan Gajanan. M.S., Purdue University, May 2013. Real-time Monitoring of Distributed Real-time and Embedded Systems using Web. Major Professor: James H. Hill.

Asynchronous JavaScript and XML (AJAX) is the primary method for enabling asynchronous communication over the Web. Although AJAX is providing warranted real-time capabilities to the Web, it requires unconventional programming methods at the expense of extensive resource usage. WebSockets, which is an emerging protocol, has the potential to address many challenges with implementing asynchronous communication over the Web. There, however, has been no in-depth study that quantitatively compares AJAX and WebSockets.

This thesis therefore provides two contributions to Web development. First, it provides an experience report for adding real-time monitoring support over the Web to the Open-source Architecture of Software Instrumentation of Systems (OASIS), which is open-source real-time instrumentation middleware for distributed real-time and embedded (DRE) systems. Secondly, it quantitatively compares using AJAX and WebSockets to stream collected instrumentation data over the Web in real-time. Results from quantitative comparison between WebSockets and AJAX show that a WebSockets server consumes 50% less network bandwidth than an AJAX server; a WebSockets client consumes memory at constant rate, not at an increasing rate; and WebSockets can send up to 215.44% more data samples when consuming the same amount network bandwidth as AJAX.

1 INTRODUCTION

Web 2.0 [1] technologies, such as *Asynchronous JavaScript and XML (AJAX)* [2], are revolutionizing how end-users interact with Web sites and Web applications. Instead of using many different pages and server callbacks to deliver content, Web 2.0 technologies enable Web sites to deliver content in real-time to Web clients while the end-user remains on the same web page. For example, it is possible to embed into an existing web page a real-time instant messaging widget that does not require the end-user to *refresh* the page, or visit a new page to send and/or receive messages.

Because of technologies like AJAX, web developers have open standards-based protocols built into the Web client that supports real-time monitoring capabilities via the Web. This is opposed to traditional methods that relied on embed applets, and required developers to design, implement, and integrate proprietary networking protocol manually. Within the AJAX realm, there are three primary patterns for asynchronous communcation: *polling* [3], where the Web client sends a request at regular intervals and the Web server sends a response immediately then closes the connection; *long-polling* [4], where the Web client sends a request and the Web server keeps the connection open for an extended period of time; and *streaming* [5], where the Web server keeps the connection open indefinitely and streams responses to the Web client until the Web client terminates the connection.

Although AJAX is addressing many shortcomings of traditional Web development, *e.g.*, static web pages and language dependency, AJAX can be resource intensive in both memory usage and network bandwidth—especially when streaming content in real-time. This is because the AJAX Web server uses indefinite loops to stream content in real-time. Likewise, the semantics of how content is streamed and delivered causes new content to be appended the existing content until the existing connection is closed and reopened.

WebSockets [6], which is an emerging technology that integrates socket-like communication mechanisms into the Web, has the potential to address many challenges introduced by AJAX. For example, WebSockets does not inherently append new content to existing content as done with AJAX. Likewise, WebSockets provides raw socket capabilities to the Web. It is therefore possible to build—from the ground up—custom protocols using WebSockets that best suites the target application domain. This is opposed to forcing an existing protocol to operate in an unfit application domain.

WebSockets, however, is a relatively new technology and not supported by many browsers [7]. Because of this, it is not well-known how WebSockets compares with AJAX, which is the most prominent technology that enables real-time communication via the Web [8], when enable real-time monitoring support for DRE systems over the Web. Based on this understanding, the main contributions of this paper are as follows:

- It provides an experience report for enabling a real-time monitoring support for DRE systems via the Web;
- It quantitative compares using AJAX and WebSockets to enable real-time monitoring by measuring both client- and server-side performance metrics, such as network bandwidth, throughput, and memory usage; and
- It provides lessons learned for implementing real-time monitoring support via the Web using AJAX and WebSockets.

We perform a quantitative study in the context the *Open-Source Architecture for Software Instrumentation of Systems (OASIS)* [9], which is open-source real-time instrumentation middleware for distributed real-time and embedded (DRE) systems. OASIS enables real-time instrumentation of DRE systems without *a priori* knowledge of metric structure and complexity. Likewise, instrumentation behavior can be modified at runtime to ensure minimal impact on software system performance. Finally, results from our study show that a WebSockets server consumes 50% less network bandwidth than an AJAX server; a WebSockets client consumes memory at constant rate, not at an increasing rate; and WebSockets can send up to 215.44% more data samples while consuming the same amount network bandwidth when compared to AJAX.

1.1 Organization of thesis

The remainder of this thesis is organized as follows: Chapter 2 compare our work with WebSockets and OASIS with other related works; Chapter 3 provides a brief overview of OASIS; Chapter 4 explains how AJAX and WebSockets are integrated into OASIS; Chapter 5 discusses the results of our comparative study; Chapter 6 provides concluding remarks and lessons learned; and Chapter 7 provide future research directions.

2 RELATED WORKS

2.1 Dakshita

Dakshita [10] is a web-based real-time web-based monitoring condition monitoring system for power transformers. Dakshita collects data from hardware sensors and stores collected data in an Oracle database. It then uses AJAX to stream content to a Web application. As per our work integrating both AJAX and WebSockets into OASIS, we have learned that their approach is pseudo real-time. This is because storing and retrieving data from the database is time-consuming, and increases the chance of retrieving stale, or out-of-date, data. Our performance testing also shows that WebSockets is a potential solution to resolving such issues that may arise.

2.2 Cara

Cara [11] is a web-based real-time remote monitoring system for pervasive healthcare that uses Flex (www.adobe.com/products/flex.html) and FluorineFx.Net (www.fluorinefx.com). Within Cara, sensors collects data and transmit it a gateway using Bluetooth or Wi-Fi. The gateway then streams the data to the Cara server using Adobe Flash. End-users can then view the data in real-time by logging in to the Cara server. Experiments were conducted to measure Cara's networking latency on different networks. The experiments revealed that Cara experiences high network latencies, which is attributed to high network bandwidth usage. Our experiments also show that AJAX, which is similar to Flex, has high network bandwidth usage. Lastly, WebSockets supports data fragmentations (*i.e.*, data can be divided into multiple frames and transferred independently), which can be useful for Cara's video streaming feature.

2.3 StreamWeb

StreamWeb [12] is a real-time web monitoring system with stream computing application domain that is developed atop of a stream computing system called System S [13–15] developed by IBM Research. Under the hood, StreamWeb uses AJAX to stream content to the Web application in real-time. StreamWeb, however, does not keep the AJAX connection open between multiple request for content in real-time. We believe this is one approach to reduce network bandwidth and memory consumption experience with AJAX, but it hinders stream content in real-time at high rates. We therefore believe that WebSockets can be used to address this design challenge, and enable updates at higher rates since the connection between the Web application and the server remains open.

Lastly, Websocket.org provides interesting results that compare the performance of WebSockets and Comet [16]. Comet is web technology that uses long-polling technique to achieve real-time behavior. According to the results, Websocket.org shows that WebSockets has better throughput and less network latency when compared to Comet. Our results not only complement and extend their experimental results, it increases support for using WebSockets (an emerging Web technology) to enable real-time behavior via the Web when compared to AJAX, and similar Web technologies.

3 A BRIEF OVERVIEW OF OASIS

OASIS is real-time instrumentation middleware for DRE systems that uses a metametics driven design integrated with loosely-coupled data collection facilities. Figure 3.1 presents an high-level overview OASIS's architecture. As shown in this figure, OA-

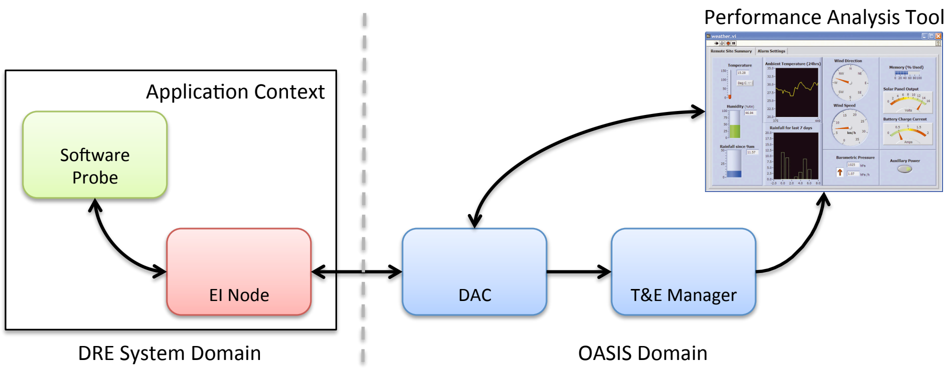


Figure 3.1. A high-level overview of OASIS architecture and middleware

SIS's architecture has the following key entities:

- **Software Probe.** The software probe is the entity that is responsible for collecting metrics from the DRE system under software instrumentation. Developers define software probes using the *probe definition language (PDL)*. The PDL is then use to generate base implementations for packing collected instrumentation data, and stubs for unpacking collected instrumentation data. System developers then have the option of inheriting the base implementation to define more domain-specific behavior for collecting instrumentation data, such as using system APIs to read the data points.

```
[uuid(ed970279-247d-42ca-aeaa-bef0239ca3b3); version(1.1)]
abstract probe MemoryProbe {
  uint64 physical_memory_avail, physical_memory_total;
  uint64 system_cache;
  uint64 commit_limit, commit_total;
  uint64 virtual_total, uint64 virtual_used;
};

[uuid(81DA0F4B-2712-4A7A-ABE4-F74C80A5C069); version(1.1)]
probe LinuxMemoryProbe : MemoryProbe {
  uint64 buffers, swap_cache;
  uint64 inactive, active;
  uint64 high_total, high_free, low_total, low_free;
  uint64 swap_total, swap_free;
  uint64 dirty, write_back;
  uint64 virtual_chunk;
};

[uuid(C78815F8-4A43-43BE-9E58-FE875E961B7D); version(1.1)]
probe WindowsMemoryProbe : MemoryProbe {
  uint64 page_file_total, page_file_avail;
  uint64 kernel_total, kernel_paged, kernel_nonpaged;
  uint64 page_size;
  uint64 commit_peak;
};
```

Above listing shows the PDL for a software probe that collects memory usage data from the host system. The base implementation for either the `LinuxMemoryProbe` or `WindowsMemoryProbe` is inherited to extract the data from

`/proc` or the Windows Performance Counters on Linux and Windows hosts, respectively. Lastly, software probes can be client-driven or active objects.

- **Embedded Instrumentation Node.** The Embedded Instrumentation (EI) Node bridges locality constrained abstractions with networking abstractions. When the EI Node receives collected instrumentation data as a data packet, it prepends its information (*e.g.*, UUID, packet number, timestamp, and host-name) to the data packet, and sends it over the network. The EI Node is not bound to a specific network communication protocol, or technology. For example, the EI Node can use an implementation of CORBA [17] (*e.g.*, The ACE ORB (TAO) [18]) or Data Distribution Services (DDS) [19] (*e.g.*, RTI-DDS (www.rti.com) and OpenSplice (www.prismtech.com/opensplice)) to send the fully packaged EI Node instrumentation data. This design approach allows DRE system developers to select the most appropriate networking middleware for their domain without impacting how OASIS packages instrumentation data. Lastly, there is one EI Node per application context (*i.e.*, an execution block, such as a for loop or conditional, an object/class, a component, or single application).

- **Data Acquisition and Controller.** The Data Acquisition and Controller (DAC) is responsible for receiving packaged data from an EI Node and controlling access to it. The DAC also manages *data handlers*, which are objects that act upon instrumentation data received from an EI Node. For example, an archive data handler stores collected metrics in a relational database, and a real-time publisher data handler allows clients to register for instrumentation data and receive it in real-time. This design approach allows OASIS to abstract away the data collection facilities from its data handling facilities, and places the data handling facilities outside of the DRE system's execution domain.

- **Test and Execution Manager.** The Test and Execution (TnE) Manager is a naming service for the active DACs. It is therefore the main entry point into OASIS for clients that want to access collected instrumentation data.

- **Performance Analysis Tools.** The performance analysis tools are clients that use instrumentation data collected by OASIS. Examples of performance analysis tools include, but is not limited to: real-time event processing engines and dashboards. Lastly, performance analysis tools can send signals/commands to software probes that alter its behavior at runtime. This design enables system developers, system testers, and performance analysis tools to control the effects of software instrumentation at runtime and minimize OASIS's overhead.

With the recent advances in Web technologies, such as AJAX and WebSockets, it is now possible to leverage the Web to monitor DRE systems in real-time. It, however, is unknown what impact such technologies have on this domain. Moreover, WebSockets is a fairly new technology when compared to AJAX. It is therefore unknown what technology is better for this domain. The remainder of this paper therefore discusses how AJAX and WebSockets are integrated into OASIS, and compares the performance of the two technologies.

4 INTEGRATING WEBSOCKETS AND AJAX IN OASIS

The previous chapter provided an overview of OASIS. As discussed in that section, the data handler is an integral part of OASIS. This is because the data handler processes instrumentation data received by the DAC outside of the DRE system undergoing software instrumentation. Because we want to integrate both AJAX and WebSockets into OASIS—as explained in Section 3—and compare its performance, the data handler is the best location to perform this integration because it ensures minimal impact on the DRE system's performance. The remainder of this chapter therefore discusses how we integrated an AJAX and WebSockets data handler in OASIS with the goal of comparing their performance and applicability in real-time instrumentation and monitoring of DRE systems.

4.1 Integrating AJAX into OASIS

Figure 4.1 provides an overview of how AJAX is integrated into OASIS. As shown in this figure, AJAX is integrated as a DAC data handler. When the DAC receives instrumentation data, it is forwarded to the AJAX data handler. The AJAX data handler then unpacks the instrumentation data, and writes it to a local file on disk in *JavaScript Object Notation (JSON)* [20] format. This is similar to writing the instrumentation data to a database.

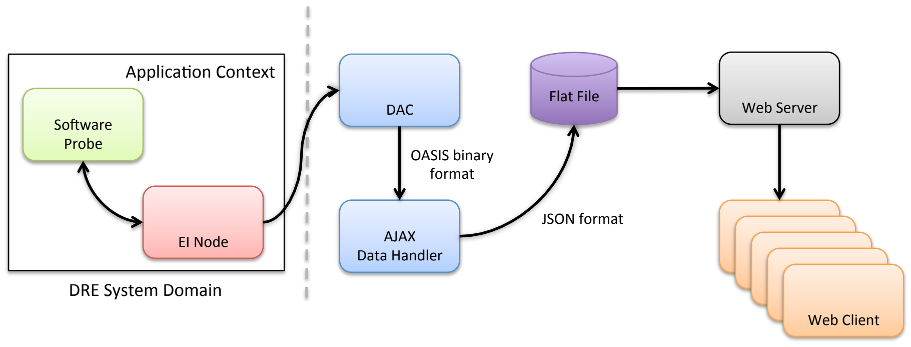


Figure 4.1. High-level overview of integrating of AJAX into OASIS

We then implemented a simple PHP (www.php.net) Web page that reads the values from the text file updated by AJAX data handler. The Web page is written in such a way that it executes an infinite loop while checking for updates to the text file. If an update is detected, then the Web page reads the values from the local file and streams it to the Web client. Because of streaming pattern, the Web server keeps the HTTP connection open indefinitely. Lastly, the Web page is hosted in an Apache Web server.

For this integrated version, we designed a simple Web application that uses the XMLHttpRequest object to open a connection to the AJAX data handler and receive instrumentation data in real-time. When the Web application receives a new JSON message it locates the last data sample received, and updates an HTML table with it.

4.2 Integrating WebSockets into OASIS

Figure 4.2 provides an overview of how WebSockets is integrated into OASIS. As shown in this figure, WebSockets is integrated into OASIS as a data handler. Unlike the AJAX integration, there is no intermediate step between the WebSockets data handler and the Web application. Instead, as the WebSockets data handler is forwarded instrumentation data by the DAC, it sends the instrumentation data to the Web application in the same binary format.

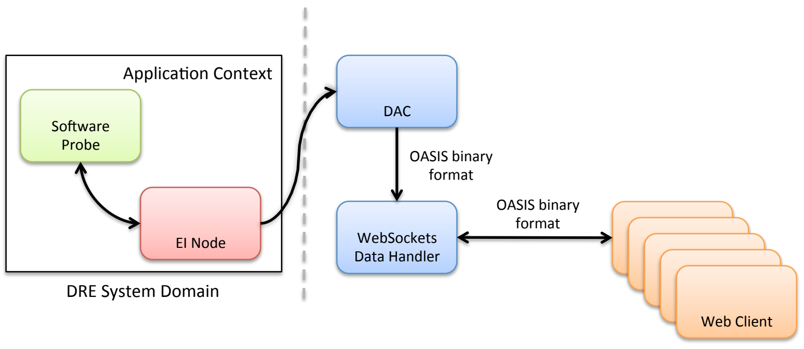


Figure 4.2. High-level overview of integrating WebSockets into OASIS

In comparison to the AJAX data handler (explained in Section 4.1), the WebSockets data handler's design and implementation is more complex as shown in Figure 4.3. We implemented the WebSockets data handler using the Adaptive Communication Environment (ACE) [21], which is a widely-used C++ framework for writing portable networked applications, and used heavily in DRE systems. We also used ACE to implement the WebSockets data handler because its Acceptor/Connector framework simplified many networking challenges such as reading/writing data in the correct byte order; reading/writing frames, which is an integral part of the WebSockets protocol; and managing connections between multiple Web applications.

As shown in Figure 4.3, the WebSockets data handler is composed of the following key abstractions that are designed to be used by any WebSockets client/server:

- **WebsockAcceptorTask.** The WebsocketAcceptorTask is an active object that extends the ACE_Task class in ACE. This object executes N number of threads that run an event loop of an ACE_Reactor object. The ACE_Reactor object is an object that dispatches events, such as input handle events and timeout events, to the task executing the reactor's event loop. This simplifies handling input events from the Web application.

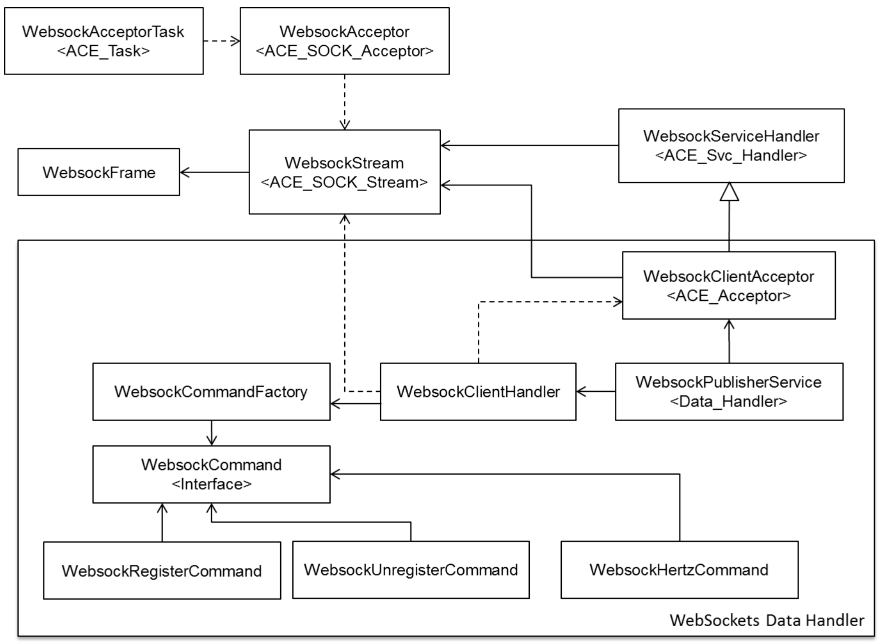

WebsockAcceptorTask <ACE_Task>
WebsockAcceptor <ACE_SOCK_Acceptor>
WebsockServiceHandler <ACE_Svc_Handler>
WebsockFrame
WebsockStream <ACE_SOCK_Stream>
WebsockClientAcceptor <ACE_Acceptor>
WebsockCommandFactory
WebsockClientHandler
WebsockPublisherService <Data_Handler>
WebsockCommand <Interface>
WebsockRegisterCommand
WebsockUnregisterCommand
WebsockHertzCommand
WebSockets Data Handler

Figure 4.3. Architectural diagram of the WebSockets data handler implemented using ACE.

- **WebsockAcceptor.** The WebsockAcceptor class extends a class in ACE called ACE_SOCK_Acceptor. The purpose of the WebsockAcceptor object is to listen for incoming connections from the Web application on a specified port. When the WebsockAcceptor is notified of an incoming connection, it accepts the connection and creates a stream for sending/receiving data to/from the Web application. This object then authenticates itself with the Web application as per the Websockets specification. If the authentication succeeds, the Websockets data handler can begin communicating with the to the Web application. If the authentication fails, then the connection is closed.

- **WebsockStream.** The WebsockStream class extends a class in ACE called ACE_SOCK_Stream. This object is responsible for sending/receiving data to/from the Web application. The WebsockStream also abstracts away the framing complexity of the Websockets protocol with simple send/receive methods that take a data buffer. The WebsockStream then use special data structures to package/unpackage the data accordingly to the WebSockets protocol.

- **WebsockServiceHandler.** The WebsockServiceHandler extends a class in ACE called ACE_Svc_Handler. This class is responsible notifying WebsockStream objects when data from a Web application is ready for reading.

- **WebsockFrame.** The WebsockFrame class is a helper class that builds frames according to Websocket protocol. It is primarily used by WebsockStream objects.

In addition to the generic abstractions discussed above, the following abstractions are specific to the Websockets data handler:

- **WebsockClientHandler.** The WebsockClientHandler extends a class in ACE called ACE_Service_Handler. This object adds an extra level of indirection to ACE's reactor design, but adds more flexibility when sending instrumentation data to the Web application. It is also responsible for handling commands sent from performance analysis tool.

- **WebsockClientAcceptor.** The WebsockClientAcceptor class extends a class in ACE called ACE_Acceptor. This object is a factory for WebsockClientHandler objects. When it creates a new WebsockClientHandler, the WebsockClientAcceptor registers it the system's reactor. This object is also responsible for managing the subscription status for instrumentation data for connected Web applications.

- **WebsockPublisherService.** The WebsockPublisherService implements the DAC's data handler interface. This object is therefore where OASIS integrates with Websockets. When the DAC receives instrumentation data, it is forwarded to this object. The WebsockPublisherService then forwards the instrumentation data to the WebsockClientHandler, which is responsible for distributing the data accordingly.

- **WebsockCommandFactory.** The WebsockCommandFacctory is a factory class which generates appropriate command object based on request from WebsockClientHandler.

- **WebsockCommand**. The WebsockCommand is an interface for concrete command objects.

- **WebsockRegisterCommand.** The WebsockRegisterCommand implements WebsockCommand interface and responsible for handling registration of probe information as requested by perfomance analysis tool.

- **WebsockUnregisterCommand.** The WebsockUnegisterCommand implements WebsockCommand interface and responsible for deleting registration information of probe as requested by performance analysis tool.

- **WebsockHertzCommand.** The WebsockRegisterCommand implements WebsockCommand interface and responsible for changing hertz rate of a particular probe as requested by performance analysis tool.

When Websockets sends instrumentation data to the Web application, it is in binary format and packaged according to OASIS's packaging specification. We therefore had to implement JavaScript classes that converted the OASIS binary data to standard types in JavaScript. This also included resolving byte order issues, if they were applicable. Once the Web application converts the received binary data to its equivalent JavaScript types, the Web application updates an HTML table with the latest values from the data sample—similar to the Web application used with the AJAX data handler.

5 COMPARISON OF AJAX AND WEBSOCKETS

This section discusses experimental results for integrating AJAX and WebSockets into OASIS to enable real-time monitoring of DRE systems as discussed in Section 4. The experimental results discussed in this section focus on the following three performance properties:

- **Web application memory consumption.** This property focuses on how much memory the Web application consumes while receiving instrumentation data from the DAC using either AJAX or WebSockets. We selected this performance property because the Web application is an integral part of real-time monitoring that must run for extended periods of time. The Web application therefore should run efficiently as possible on general-purpose computers (*e.g.*, laptops, mobile phones, and tablets). This will allow the end-user to take advantage of real-time monitoring from any place that supports an Internet connection.

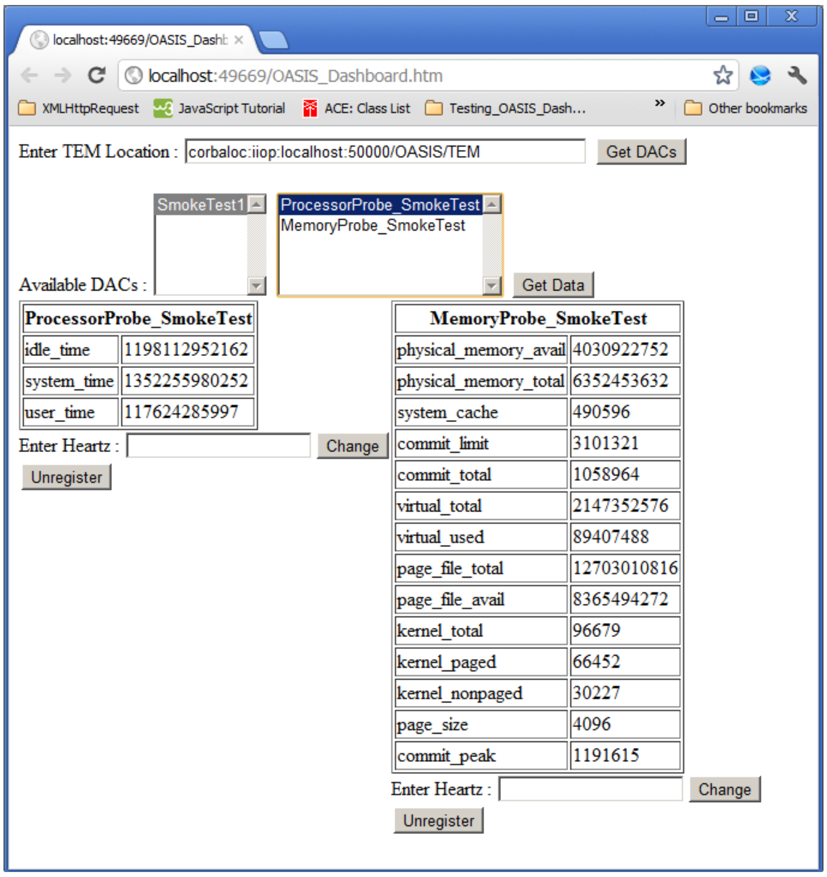
- **Network bandwidth consumption.** This property is concerned with evaluation how much network bandwidth the AJAX and WebSockets implementation use. We selected this performance property for two reasons. The first reason is because of economics. Network bandwidth is a costly, especially with mobile phone carriers now placing restrictions on network bandwidth consumption [22]. This implies that network bandwidth usage should be kept as minimal as possible to ensure that real-time monitoring is affordable. The second reason is because real-time monitoring is inherently data-intensive. This implies that network congestion can easily become a problem, and delay receipt of collected instrumentation data.

- **Data Throughput.** This property is concerned with evaluating how much data AJAX and WebSockets can handle when integrated into OASIS. We selected this performance property because it provides insight on their capacity and scalability.
- **Data Lag.** This property is concerned with evaluating how long it take to transfer data over network using Ajax and WebSockets. We selected this performance property because it provides insight on thier ability to produce data on client side. This property is important in application domain ,i.e. DRE systems, because of it's stringent time constraints.

We developed two sample Web applications for our experiments. The first application used AJAX to send instrumentation data in real-time from the DAC to the Web application (see Section 4.1). The second application used WebSockets to send instrumentation data in real-time from the DAC to the Web application (see Section 4.2). We used the System Probe Daemon tool, which is a tool provided with OASIS, to collect processor and memory information from each host in the experiment. Lastly, the Web applications were executed in Google Chrome 19 and displayed received instrumentation data in table format. Figure 5.1 shows a screenshot of the Web application without any instrumentation data.

All experiments were conducted in the System Integration (SI) Lab at IUPUI (www.emulab.cs.iupui.edu), which is powered by Emulab [23] software. Figure 5.2 provides high-level overview of testing environment we used for web application performance testing. Each experimental node in the SI Lab is a Dell PowerEdge R415, AMD Opteron 4130 processor with 8GB of memory executing 32-bit Fedora Core 15 (32 bit). *Boss* is Dell PowerEdge R415, AMD Opteron 4130 processor, 8GB of memory executing 32-bit FreeBSD 7.3.

For each experiment execution, the System Probe Daemon tool, DAC, and TnE Manager were deployed on their own experimental node. We only used one DAC in the experiments because we can not focus on scaling the OASIS architecture with



Enter TEM Location : corbaloc:iiop:localhost:50000/OASIS/TEM

Available DACs : SmokeTest1

ProcessorProbe_SmokeTest
MemoryProbe_SmokeTest

ProcessorProbe_SmokeTest	
idle_time	1198112952162
system_time	1352255980252
user_time	117624285997

Enter Heartz :

MemoryProbe_SmokeTest	
physical_memory_avail	4030922752
physical_memory_total	6352453632
system_cache	490596
commit_limit	3101321
commit_total	1058964
virtual_total	2147352576
virtual_used	89407488
page_file_total	12703010816
page_file_avail	8365494272
kernel_total	96679
kernel_paged	66452
kernel_nonpaged	30227
page_size	4096
commit_peak	1191615

Enter Heartz :

Figure 5.1. Screenshot of the basic web application used to display instrumentation data received in real-time using either AJAX or WebSockets.

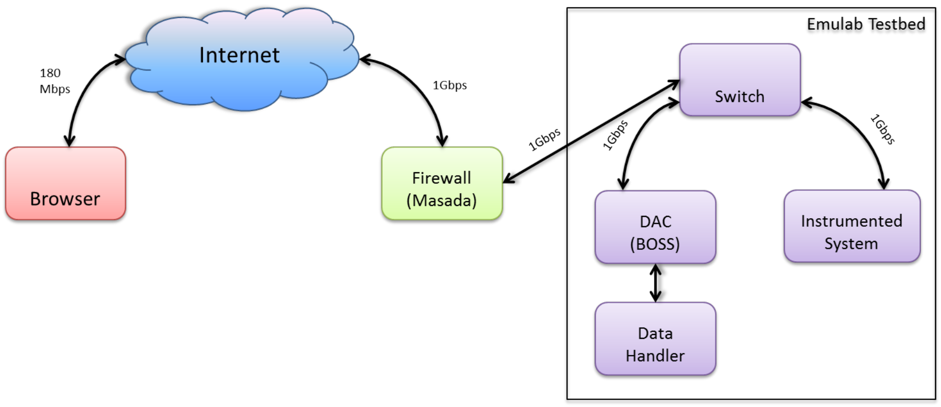


Figure 5.2. A high-level overview of performance testing environment

respect to streaming instrumentation data to the Web application. Finally, the Google Chrome web browser (*i.e.*, the performance analysis tools) was deployed on a Dell XPS 15z laptop with Intel Core i5 processor and 6 GB of memory executing 64-bit Windows 7 Ultimate. The laptop resided outside of the SI Lab, and the instrumentation data was sent over the public Internet using a WiFi connection. The remainder of this section discusses the results of our experiments evaluating the three performance properties discussed above.

5.1 Experiment 1: Web Application Memory Consumption Test

The goal of this experiment is to compare memory consumption on the client-side (*i.e.*, measure how much memory the web browser is using,) when using AJAX and WebSockets to monitor collected instrumentation data in real-time.

5.1.1 Experiment Design & Setup

Using the general experimental setup explained at the beginning of this section, we configured the System Probe Daemon tool to collect instrumentation data at 1 Hz. We selected 1 Hz because it allowed us to stream collected instrumentation data in real-time using both AJAX and WebSockets under similar operating conditions. When designing this experiment, We learned that if we collect instrumentation data at to high of a rate, then the AJAX data handler publishes data at a lesser rate than the WebSockets data handler. This is because the AJAX design has a ”middle-man” (*i.e.*, the flat file) that enables streaming, and the ”middle-man” introduces a delay that is not present in the WebSockets experiment. Finally, we executed the tests for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* [Appendix C] at 30 second intervals.

5.1.2 Experiment Results

Figure 5.3 shows the memory consumption results for AJAX and WebSockets when integrated into OASIS. As also shown in Figure 5.3, memory consumption for the Web application that uses AJAX increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. This is because AJAX implements with streaming pattern by appending new messages to previously received messages. This causes the response to increase in size over time and causes the Web client to consume more memory over time.

In case of WebSockets, each message is transferred in its own frame, or set of frames. The Web application that uses WebSockets therefore consumes an amount of memory that is consistent with the amount of memory that represents only the latest data sample. This analysis, however, disregards any memory consumed by the Web application in regards to storing and interacting with the received data. Finally, because of how we had to design the experiment to ensure fair comparison between AJAX and WebSockets, we received our first insight that AJAX Web servers cannot stream content as fast as WebSockets Web servers. This is illustrated in more detail in Section 5.3. we have conducted same experiment for different time durations and time intervals. We have included graphs for results of those experiments in Appendix A. We also compared memry consumption of AJAX web application implemented using

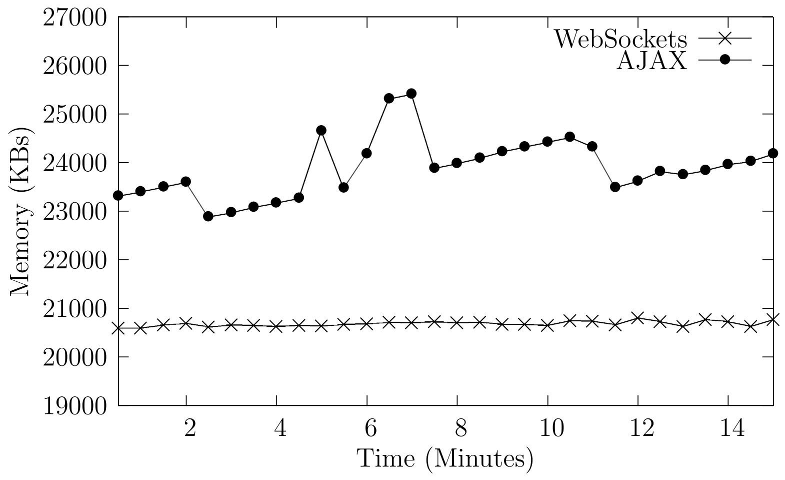


Figure 5.3. Results comparing Web application memory consumption between AJAX and WebSockets when integrated into OASIS.

long-polling [4] with WebSockets web application. We have included graphs for results of those experiments in Appendix B. Comparison results between AJAX long-polling pattern and WebSockets bolsters the conclusions we made above.

5.2 Experiment 2: Network Bandwidth Consumption Test

The goal of this experiment is to compare network bandwidth consumption between the AJAX and WebSockets implementation when integrated into OASIS to enable real-time monitoring of DRE systems.

5.2.1 Experiment Design & Setup

Using the general experimental setup explained at the beginning of this section, we configured the software probe to flush a fixed number of data messages. This is because we wanted to ensure that both the AJAX and WebSockets had the same amount of workload. If we allowed the software probe to run for a fixed amount to time, then the comparison would be unfair. This is because we learned from the previous experiment that WebSockets can operate at a much higher rate than AJAX, and the comparison of network bandwidth between both implementations would not be under the same operating conditions.

We used WireShark (www.wireshark.org), which is an open-source tool for monitoring packets on a network, in this experiment. More specifically, we used WireShark to monitor only the packets sent between the DAC and Web application by measuring the number of bytes associated with each packet (Appendix D). Finally, we executed the experiment 10 different times using 10 different number of fixed software probe flushes.

5.2.2 Experiment Results

Figure 5.4 shows the network bandwidth consumption results for AJAX and WebSockets when integrated into OASIS. As shown in this figure, the Web application that uses AJAX always consumes more network bandwidth than the Web application that uses WebSockets. There are two main reasons behind this observation. First, we learned that AJAX requires that at least 256 bytes of data per message. We therefore had to added 256 bytes of whitespace to every message if its original size was under 256 bytes. This requirement causes unnecessary bandwidth usage. Second, we observed that AJAX's header size is unpredictable, but it is always significantly greater than its equivalent in WebSockets.

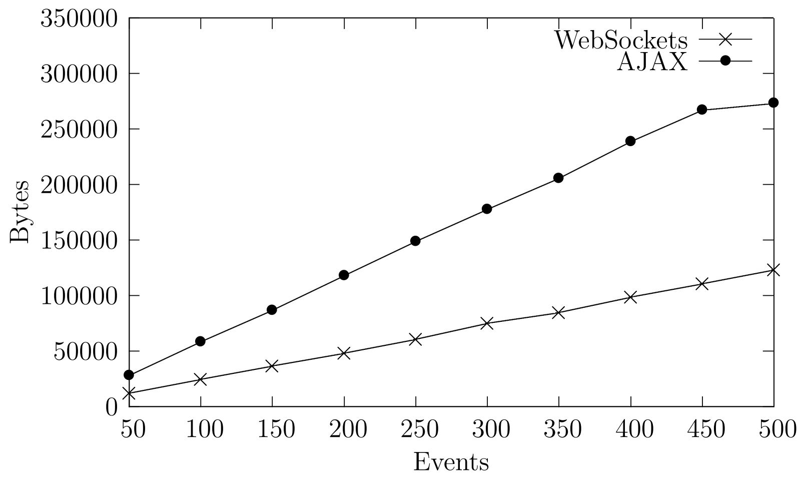


Figure 5.4. Results comparing network bandwidth consumption between AJAX and WebSockets when integrated into OASIS.

5.3 Experiment 3: Data Throughput and Data Latency Test

The goal of this experiment is to compare compare data lag between WebSockets and AJAX when integrated into OASIS and the System Probe Daemon tool is collecting instrumentation data at its maximum rate.

5.3.1 Experiment Design & Setup

Using the general experimental setup explained at the beginning of this section, we configured the System Probe Daemon tool to collect and send instrumentation data to the DAC as fast as possible, which was then sent to the Web application. We then counted the number data samples sent to the DAC and the number of data samples received by the Web client. We designed the experiment this way because we wanted to compare real-time performance of WebSockets and AJAX under extreme conditions. This experiment also allows us to establish maximum throughput for both technologies. Finally, each test was executed for 1 minute.

5.3.2 Experiment Results

Table 5.1 and Table 5.2 presents results that measure both throughput and data lag, which we define as the percentage of events sent by the server that have not been received by the client over a period of time, for AJAX and WebSockets when integrated into OASIS. As shown in the Table 5.1, the WebSockets data handler

Table 5.1
Throughput results for WebSockets when integrated into OASIS

	Avg. Packet Size (Bytes)	**Samples Sent**	**Samples Received**	**Data Lag**
Processor Probe:	140	94,294	94,294	0%
Memory Probe:	236	67,344	67,344	0%
Total:	376	161,638	161,638	0%

sent a total of 161638 data samples (*i.e.*, processor and memory software probe data sample), and the Web application received all the data samples within the 1 minute time frame. This means that the WebSockets implementation has no data lag, but this does not mean the WebSockets implementation did not experience latencies. The latencies were low enough for our experiments that each data sample sent was received within the allotted time period.

Table 5.2
Throughput results for AJAX when integrated into OASIS

	Avg. Packet Size (Bytes)	**Samples Sent**	**Samples Received**	**Data Lag**
Processor Probe:	453	93,312	462	99.5%
Memory Probe:	733	68,011	859	98.73%
Total:	1,186	161,323	1,321	99.18%

The AJAX implementation, however, had extremely different results. As shown in Table 5.2, the AJAX data handler sent a combined 161323 data samples (*i.e.*, processor and memory software probe data sample), but the Web application only received 1321 of the sent data samples. For this experiment, the AJAX implementation has a 98% data lag for the 1 minute time period.

Based on our investigations, we believe the data lag in the AJAX the results is caused by two factors. The first factor is related to significant networking overhead. This is because the AJAX data handler receives data samples as packaged binary data and converts it to text-based data samples before sending it to the Web application. This conversion process negatively impacts its performance.

The second factor is related to how the data is received on the client-side. Although AJAX is sending only the latest data sample, it is appended to previously received samples. This means that the Web application must sift through all previously received data samples to locate the latest data sample, which will have at least a linear degradation on performance. One solution to addressing this problem is to open and close the connection continuously (*i.e.*, use a polling approach). This approach, however, will add more stress to the client and server, and reduces the overall throughput of data samples.

Lastly, Table 5.1 shows that WebSockets sends more data samples than AJAX for the same period of time. This can raise concerns that WebSockets can potentially use more bandwidth within a given time period when compared to AJAX. From Table 5.1 and Table 5.2, we can calculate that the total amount of data sent in the AJAX experiment was 1,566,706 bytes for 1,321 data samples. We can then use this amount to determine what is the equivalent number of data samples sent using WebSockets that will produce the same quantity of data sent, which is 4,167 data samples. Using this number, we can calculate that for the same amount of data, WebSockets sends up to 215.44% more data samples than the AJAX implementation. This also means that we can reduce the sending rate of WebSockets, and still send the same amount of data or more while consuming less networking bandwidth.

5.4 Experiment 4: Data Lag Test

The goal of this experiment is to compare data lag (*i.e.*, how much time data packet take to reach client from server over network,) when using AJAX and WebSockets to monitor collected instrumentation data in real-time.

5.4.1 Experiment Design & Setup

Using the general experimental setup explained at the beginning of this section, we configured the System Probe Daemon tool to collect instrumentation data at different frequencies ranging from 0.1Hz to 1Hz. In this experiment, we synchronized clock of machine running DAC and clock of laptop running web application with a dedicated time server in SI lab. The clock difference after synchronization was in microseconds, approximately between 2 microseconds to 100 microseconds, hence it was considered negligible in this experimental context. We used NISTIME, a TCP time client, to synchronize laptop clock to time server. We recorded time stamp and message number of every data message when DAC transfer data message over network. We also recoded time stamp and message number on web application side. We selected time stamps of packet number 50 to packet number 100 and took difference. Then we took an average of recorded differences to avoid extremities for all ten frequencies.

5.4.2 Experiment Results

Figure 5.5 shows data lag comparison between AJAX and WebSockets when integrated into OASIS. As also shown in Figure 5.5, data lag for AJAX application is significantly higher than that of WebSocket application. Reason behind this difference is the data file which act as a data storage in AJAX design. We come to conclusion that if application domain has stringent time constraints over data reporting and monitoring then WebSockets outperform AJAX. It is important to note that data lag results of AJAX implementation might change depending on design used.

Based on these experimental results, we can conclude that WebSockets is a better Web technology for real-time monitoring of DRE systems via the Web.

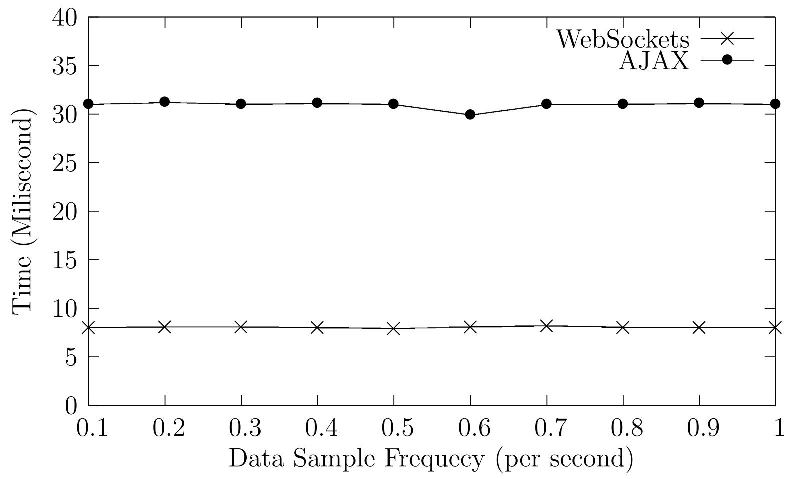


Figure 5.5. Results comparing data lag between AJAX and WebSockets when integrated into OASIS.

6 CONCLUDING REMARKS

The advent of Web 2.0 technologies, such as AJAX and WebSockets, is allowing the Web to be applied to application domains that had to use either *ad hoc* or custom solutions to realize the same capability. One such application domain is real-time instrumentation and monitoring of DRE systems. In this paper, we compared the performance of using AJAX and WebSockets to support real-time instrumentation and monitoring of DRE systems. Based on our results, we can conclude that WebSockets is a better fit for this domain because it provides higher throughput and better network performance when compared to AJAX.

Based on our experience in comparing performance of AJAX and WebSockets, we present following list of conclusions :

- **The advantage of AJAX streaming pattern.** During the data throughput and data lag experiment, we concluded that performance was degrading because AJAX was appending new data samples to previously received ones. This was causing the Web application to parse the entire response just to locate the latest data sample at the end. Although this can be viewed as a shortcoming for AJAX, it can be viewed as an advantage in application domains where historical content is displayed along with new content. When using WebSockets, the Web application has to manually implement this feature.

- **Suggestion to improve AJAX application performance.** During network bandwidth consumption test WebSockets protocol out performed AJAX streaming pattern because WebSockets use binary encoding for data transferred over network while AJAX use ASCII encoding. It will be worthwhile to come with AJAX pattern or use existing pattern to use binary encoding for data transfer.

OASIS, the AJAX data handler, and WebSocket data handler discussed in this paper are freely available in open-source format from the following location: oasis.cs.iupui.edu. Lastly, we are in the process of merging our generic Websockets abstractions into the ACE code base so it is available to the entire ACE community.

7 FUTURE WORK

This thesis provided two experimental implementations of web applications, one using AJAX and other using WebSockets. Web applications were implemented to provide real-time monitoring support for DRE systems. I gained numerous insights with respect to internals of both technologies and how they are used in the domain of real-time monitoring while implementing web applications. Real-time monitoring is inherently resource extensive domain which pushes technologies to their limits. This situation helped me learn and compare technologies in performance intensive environment. Based on my experience in implementing web applications and comparing the performance of AJAX and WebSockets, we present following list of lessons learned and future research directions.

7.1 Client-side programming laguages

In current world, client side devices such as desktop, laptop, phones have significant amount of processing power and memory availability. Application domain such as real-time monitoring which are inherently resource intensive can take advantage of that by shifting some part of processing from server-side to client-side. Currently, JavaScript is widely used client-side scripting language. There are many JavaScript libraries used by web application developers in real world. I observed that JavaScript was causing performance degradation in both the AJAX and WebSockets Web application during my experiments. I learned that JavaScript inherently makes it *hard* to use advance programming techniques to design and implement solutions that were originally design and implemented for a server. I believe, to take advantage of newly available performance capabilities on client-side, we need to improve client-side programming (or scripting) languages.

7.2 Client-side charting and graphing library

We all know picture worth thousand words. In real-time monitoring web application, best way to visualize data is in graphical format. In given implementations, I visualize data using simple html tables because intent of thesis was to compare AJAX and WebSockets. Originally, we implemented the WebSockets Web application to visualize data using *RGraph* (www.rgraph.net), which is an open-source charting library written in JavaScript that uses HTML 5 features, such as the Canvas element, to dynamically create charts on the client-side. I observed that the WebSockets implementation could transfer more than 3000 events/second. However, RGraph could not handle more than 100 events/second. From this observation, I believe, web application developers need improved client-side charting and graphing libraries which can operate in domains that have high throughput. Otherwise, web application developer willl not able to take advantage of technologies like WebSockets which can operate at high rates.

7.3 Comparison between WebSockets and other web technologies

In this thesis, I provided comparison between AJAX and WebSockets. I choose AJAX because it is an existing, well established and widely used web technology for real time updates in web application developer community. There are many other technologies used for real-time updates like Comet, Adobe Flex. It will be interesting to compare WebSockets with other web technologies. Results of comparison will be two fold, first we will gain more insight about WebSockets which will help to improve WebSockets. It is a good time to make improvements in WebSockets because WebSockets technology is still evolving. Second, these comparisons can shade light on strong holds of individual technology. Web application developer community may use results of comparisons to choose technology which is a best fit for their situation from wide range of technology. I tried to mimic AJAX application architecture used in real world as closely as possible while implementing experimental AJAX web application provided in this thesis. As it was an experimental web application, it had some deviations form real world architecture. The biggest and noticeable deviation was the use of flat files. I used flat file to store data coming from server. But in real world, database management systems are used for for data storage. Database management systems are optimized for data access activities in data intensive environment and preferred over flat files. Flat file and Flat file databases like Sqlite are used by small web applications and academia. Use of database management system might improve performance of web application which use AJAX streaming pattern. It will be interesting to evaluate AJAX application which uses database management system and compare the result with my implementation.

LIST OF REFERENCES

LIST OF REFERENCES

[1] T. O'Reilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & Strategies*, 1:17–38, 2007.

[2] A. T. Holdener III. *Ajax: the definitive guide.* O'Reilly, 2008.

[3] H. Takagi. *Analysis of polling systems.* MIT press, 1986.

[4] A. Russell. Comet: Low latency data for browsers. *alex. dojotoolkit. org*, 2006.

[5] A. Fecheyr-Lippens. A Review of HTTP Live Streaming. Technical report, Technical report, http://andrewsblog. org/a review of http live streaming. pdf, 2010.

[6] Internet Engineering Task Force (IETF). *The WebSocket Protocol*, Request for Comments: 6455 edition, December 2012.

[7] Wikipedia. WebSocket. http://en.wikipedia.org/wiki/WebSocket, 2012.

[8] BuiltWith Technology Lookup. AJAX Libraries API Usage Statistics. http://trends.builtwith.com/cdn/AJAX-Libraries-API, 2012.

[9] J. H. Hill, H. Sutherland, P. Staudinger, T. Silveria, D. C. Schmidt, J. M. Slaby, and N Visnevski. OASIS: An Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems. *International Journal of Computer Systems Science and Engineering, Special Issue: Real-time Systems*, April 2011.

[10] A. M. Wagle, A. M. Lobo, A. Santosh Kumar, S. Patil, and A. Venkatasami. Real time Web Based Condition Monitoring System for Power Transformers-Case Study. In *International Conference on Condition Monitoring and Diagnosis*, pages 1307–1309. IEEE, 2008.

[11] B. Yuan and J. Herbert. Web-based real-time remote monitoring for pervasive healthcare. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pages 625–629. IEEE, 2011.

[12] T. Suzumura and T. Oiki. StreamWeb: Real-Time Web Monitoring with Stream Computing. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 620–627. IEEE, 2011.

[13] B. Gedik, H. Andrade, K. L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[14] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM, 2006.

[15] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K. L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. *Middleware 2008*, pages 306–325, 2008.

[16] R. Gravelle. Comet Programming: Using Ajax to Simulate Server Push. *Webreference, WebMediaBrands Inc., http://www. webreference. com/programming/-javascript/rg28*, 2010.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 edition, January 2008.

[18] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[19] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, January 2007.

[20] D. Crockford. JSON: Javascript object notation, 2006.

[21] S. D. Huston, J. C. E. Johnson, and U. Syyid. *The ACE Programmer's Guide.* Addison-Wesley, Boston, 2002.

[22] D. Goldman. Sorry, America: Your Wireless Airwaves are Full. http://money.cnn.com/2012/02/21/technology/spectrum_crunch/index.htm, February 2012.

[23] J. Lepreau. The Utah Emulab Network Testbed.

APPENDICES

Appendix A: Additional results of memory consumption test

We repeated web application memory consumption experiment with different parameters and different settings. We have provided results for those experiments in following graphs. Following graphs bolstered the results we presented in Section 5.1.2.

Figure A.1 shows the memory consumption results for AJAX and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 1 minutes intervals.

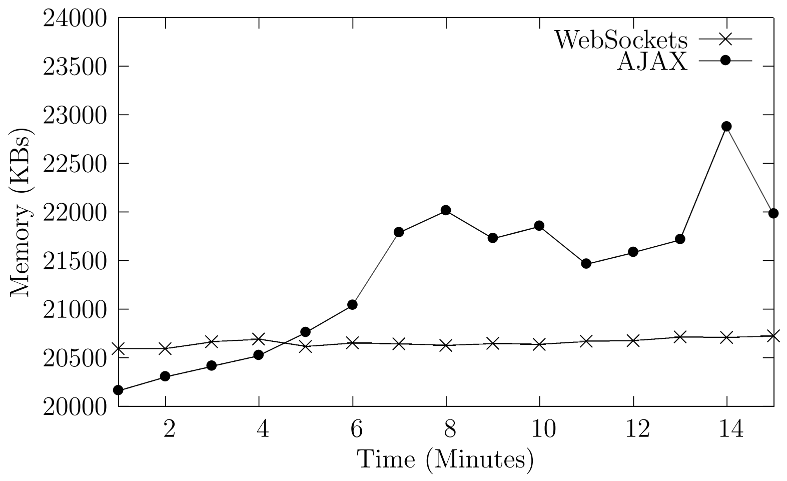


Figure A.1. Results comparing Web application memory consumption between AJAX and WebSockets when test duration set to 15 minutes.

As also shown in Figure A.1, memory consumption for the Web application that uses AJAX increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. In this experiment, we increased the time interval between samples being collected. We wanted to observe effect of time interval variation on experiment results presented in Section 5.1.2 but we didn't observe any significant difference.

Figure A.2 shows the memory consumption results for AJAX and WebSockets when integrated into OASIS. Test was executed for 30 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 2 minutes intervals.

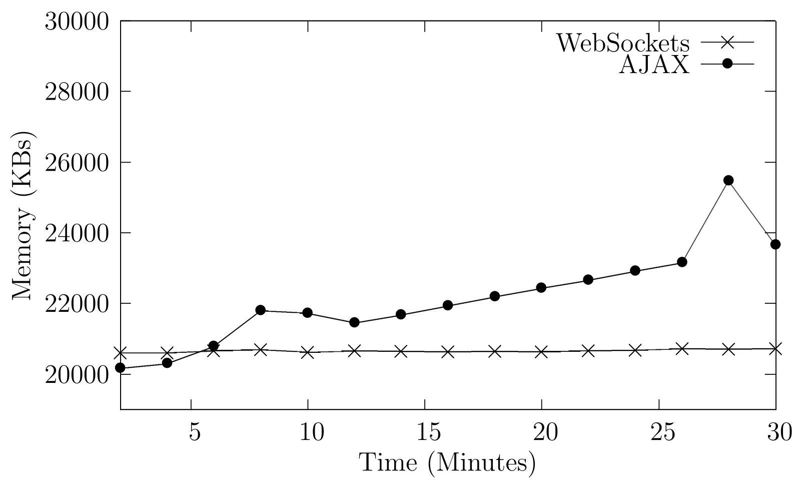


Figure A.2. Results comparing Web application memory consumption between AJAX and WebSockets when test duration set to 30 minutes.

As also shown in Figure A.2, memory consumption for the Web application that uses AJAX increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. In this experiment, we not only increased the time interval between samples being collected but also increased total duration of test execution. We wanted to observe effect of combination of change in testduration and time interval variation on experiment results presented in Section 5.1.2 but we didn't observe any significant difference.

Figure A.3 shows the memory consumption results for AJAX and WebSockets when integrated into OASIS. Test was executed for 60 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 2 minutes intervals.

As also shown in Figure A.3, memory consumption for the Web application that uses AJAX increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. In this experiment, again, we not only increased the time interval between samples being collected but also

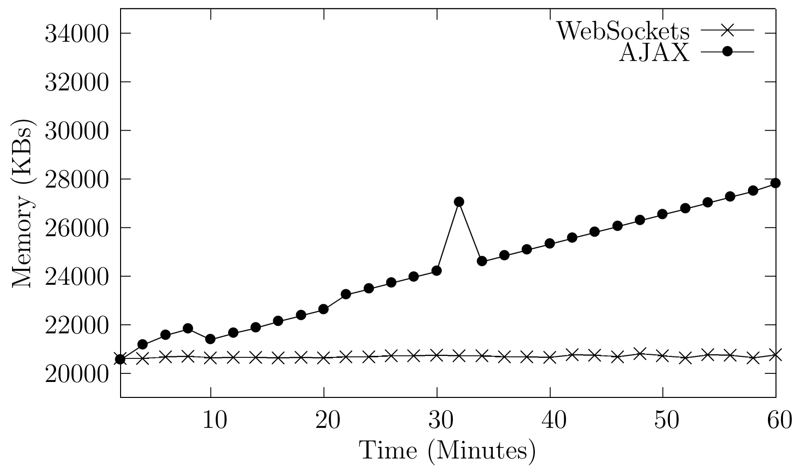


Figure A.3. Results comparing Web application memory consumption between AJAX and WebSockets when test duration set to 60 minutes.

increased total duration of test execution. We wanted to observe effect of combination of change in testduration and time interval variation on experiment results presented in Section 5.1.2 but we didn't observe any significant difference.

Figure A.4 shows the memory consumption results for AJAX and WebSockets when integrated into OASIS. Test was executed for 100 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 5 minutes intervals.

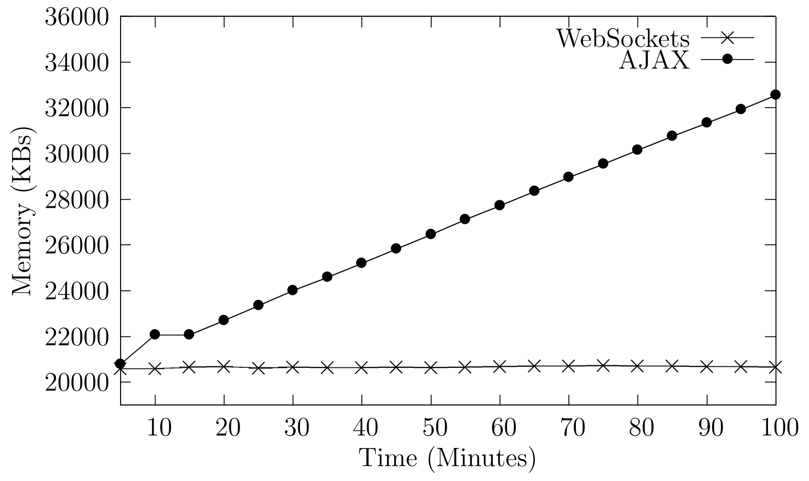


Figure A.4. Results comparing Web application memory consumption between AJAX and WebSockets when test duration set to 100 minutes.

As also shown in Figure A.4, memory consumption for the Web application that uses AJAX increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. In this experiment, We not only increased the time interval between samples being collected but also increased total duration of test execution. We changed the values significantly in this experiment than previous experiment. We wanted to observe effect of combination of change in test duration and time interval variation with relatively higher values on experiment results presented in Section 5.1.2 but we didn't observe any significant difference.

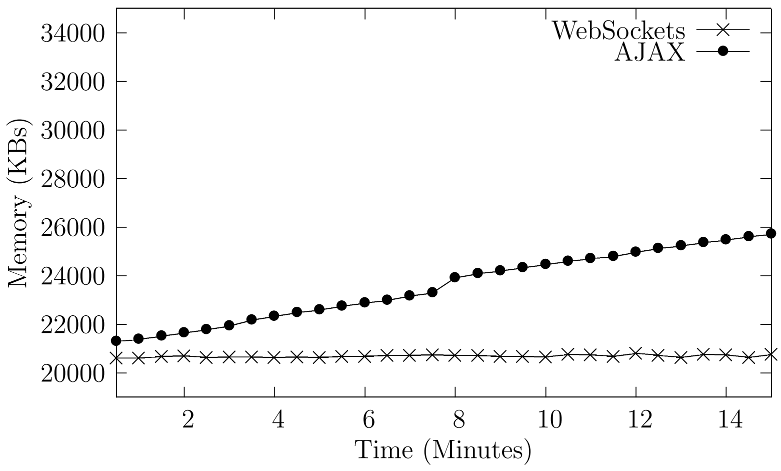


Figure B.1. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.1Hz.

Appendix B: Memory consumption test results with AJAX long-polling

We conducted web application memory consumption experiment with AJAX web application implemented using long-polling pattern. In case of this test, we dropped AJAX connection after every every 5 Seconds and resumed connection after 1 second. The reason behind doing this test was, we realized that AJAX streaming pattern buffers all previous messages so memory consumption of web application was increasing continuously. We implemented AJAX long-polling pattern to check it's impact on memory consumption test. My conclusion is, even though AJAX long-polling pattern doesn't append previous messages, that all the other conclusions we provided in my results section of all test are valid for AJAX long-polling pattern. We have provided results for those experiments in following graphs. Following graphs bolstered the results we presented in Section 5.1.2.

Figure B.1 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.1 Hz.

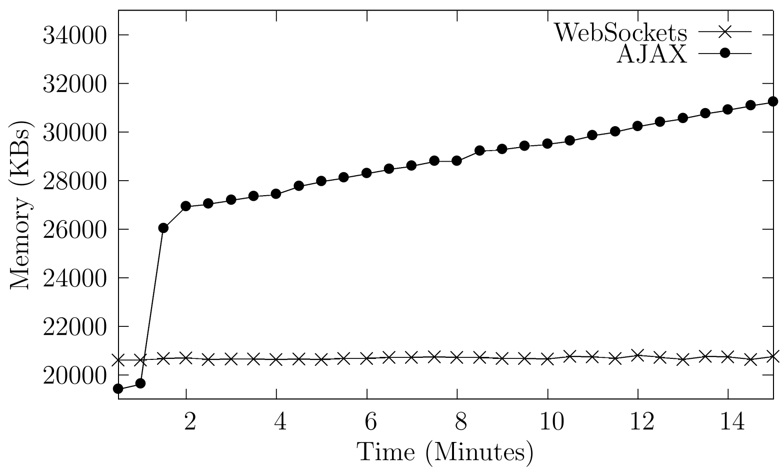


Figure B.2. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.2Hz.

As also shown in Figure B.1, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network.

Figure B.2 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.2 Hz.

As also shown in Figure B.2, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network.

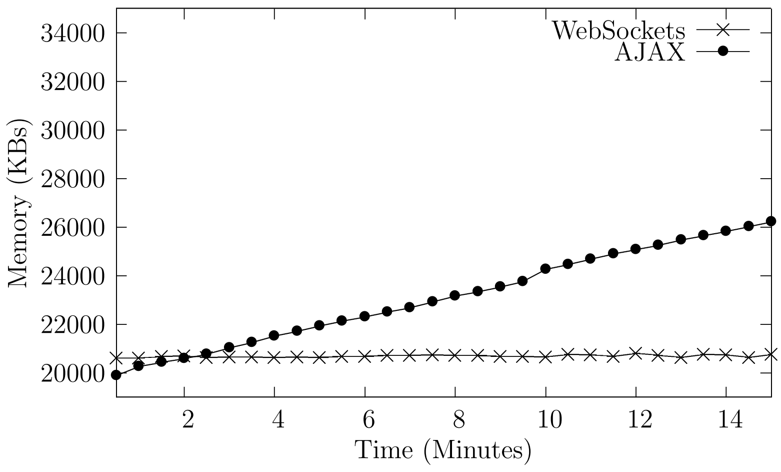


Figure B.3. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.3Hz.

We conducted this experiment with changed setting *i.e.* frequency was changed to 0.2 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.3 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.3 Hz.

As also shown in Figure B.3, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.3 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

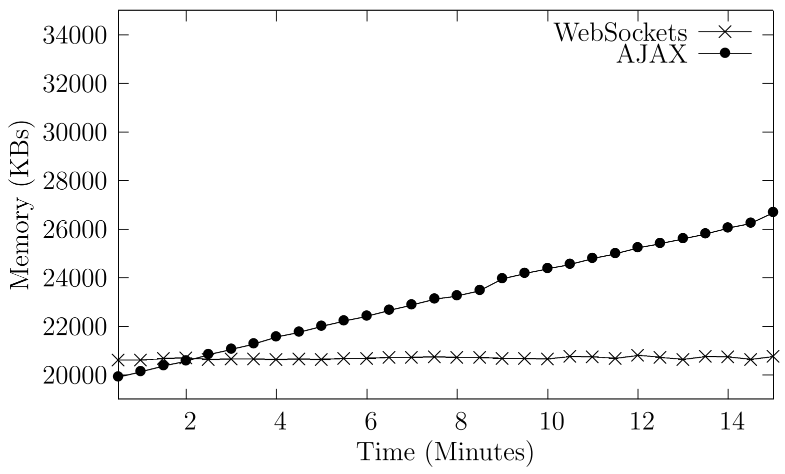


Figure B.4. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.4Hz.

Figure B.4 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.4 Hz.

As also shown in Figure B.4, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.4 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.5 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows

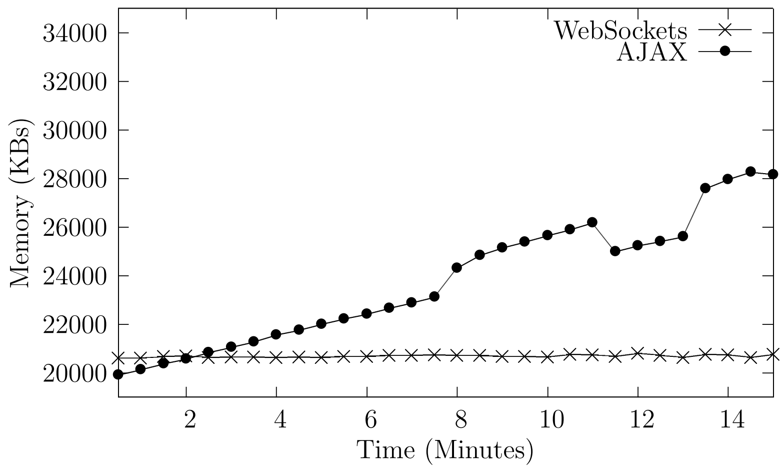


Figure B.5. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.5Hz.

command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.5 Hz.

As also shown in Figure B.5, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.5 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.6 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.6 Hz.

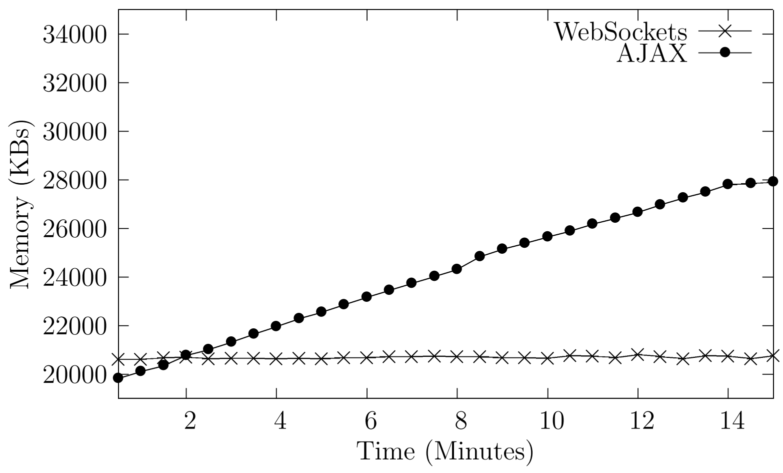


Figure B.6. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.6Hz.

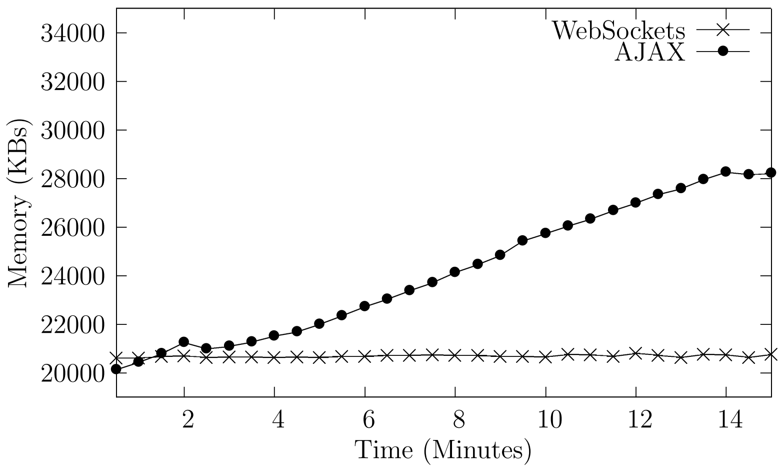


Figure B.7. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.7Hz.

As also shown in Figure B.6, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.6 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.7 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.7 Hz.

As also shown in Figure B.7, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX

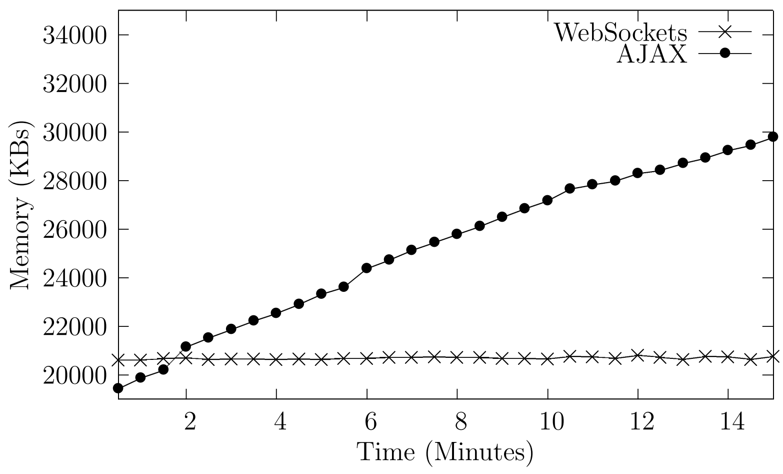


Figure B.8. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.8Hz.

web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.7 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.8 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.8 Hz.

As also shown in Figure B.8, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network.

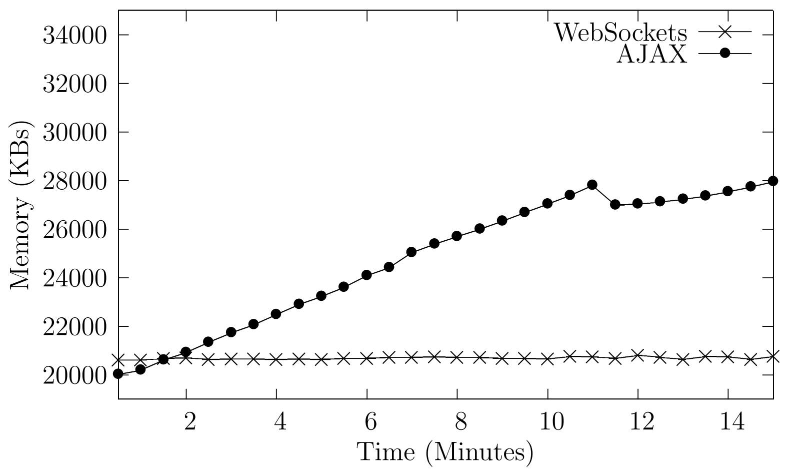


Figure B.9. Results comparing Web application memory consumption between AJAX long-polling pattern and WebSockets at 0.9Hz.

We conducted this experiment with changed setting *i.e.* frequency was changed to 0.8 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Figure B.9 shows the memory consumption results for AJAX long-polling pattern and WebSockets when integrated into OASIS. Test was executed for 15 minutes and collected memory consumption metrics for the Web application using Windows command-line tool named *Tasklist* at 0.5 minutes intervals. Frequency of data collection of probes was 0.9 Hz.

As also shown in Figure B.9, memory consumption for the Web application that uses AJAX long-polling pattern increases over time, and Web application memory consumption for the WebSockets implementation remains relatively constant. AJAX web application consumes more memory than WebSockets web application because of bigger headers and bigger data sections in a message transferred over network. Header and data sections are bigger because of ASCII encoding of data over network. We conducted this experiment with changed setting *i.e.* frequency was changed to 0.9 Hz to check how AJAX long-poling pattern performs with above change and bolster results of memory consumption test further.

Appendix C: Script to record memory used by process

Following Script was used in first experiment Web application memory consumption. This script was written to record memory usage of particular process. In this case, process was tab of chrome browser under which web applications were running.

```
@echo off
echo. > C:\Users\Darshan\Desktop\memory_log.txt
FOR /L %%a in (1,1,30) do (
tasklist
/FO CSV /FI "PID eq 1972" /NH
>> C:\Users\Darshan\Desktop\memory_log.txt
ping localhost -n 30 > nul
)
```

Appendix D: Sample packet monitored using WireShark

Following verbatim provides all the data captured in one packet by WireShark. I used WireShark in second experiment Network bandwidth consumption. Packet data may vary from packet to packet. So, this verbatim is there just to provide inside look to data contained in captured packet and structure of data contained.

```
No.     Time         Source         Destination
43    20.154750   134.68.136.11  192.168.1.143
Protocol Length Info
 HTTP     194   Continuation or non-HTTP traffic


Frame 43: 194 bytes on wire (1552 bits),
    194 bytes captured (1552 bits)
    Arrival Time: May 13, 2012 22:39:53.240444000
                  US Eastern Daylight Time
    Epoch Time: 1336963193.240444000 seconds
    [Time delta from previous captured frame: 0.002504000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 20.154750000 seconds]
    Frame Number: 43
    Frame Length: 194 bytes (1552 bits)
    Capture Length: 194 bytes (1552 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ip:tcp:http:data]
    [Coloring Rule Name: HTTP]
    [Coloring Rule String: http || tcp.port == 80]
Ethernet II, Src: Cisco-Li_d2:7d:2f (c0:c1:c0:d2:7d:2f),
                  Dst: IntelCor_22:c9:00 (88:53:2e:22:c9:00)
    Destination: IntelCor_22:c9:00 (88:53:2e:22:c9:00)
```

```
        Address: IntelCor_22:c9:00 (88:53:2e:22:c9:00)
        .... ...0 .... .... .... ....
        = IG bit: Individual address (unicast)
        .... ..0. .... .... .... ....
        = LG bit: Globally unique address (factory default)
    Source: Cisco-Li_d2:7d:2f (c0:c1:c0:d2:7d:2f)
        Address: Cisco-Li_d2:7d:2f (c0:c1:c0:d2:7d:2f)
        .... ...0 .... .... .... ....
        = IG bit: Individual address (unicast)
        .... ..0. .... .... .... ....
        = LG bit: Globally unique address (factory default)
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 134.68.136.11 (134.68.136.11),
                             Dst: 192.168.1.143 (192.168.1.143)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default;
               ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..00
= Explicit Congestion Notification:
                     Not-ECT (Not ECN-Capable Transport) (0x00)
    Total Length: 180
    Identification: 0xbf4a (48970)
    Flags: 0x02 (Don't Fragment)
        0... .... = Reserved bit: Not set
        .1.. .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
    Fragment offset: 0
    Time to live: 50
    Protocol: TCP (6)
    Header checksum: 0xb872 [correct]
        [Good: True]
        [Bad: False]
    Source: 134.68.136.11 (134.68.136.11)
    Destination: 192.168.1.143 (192.168.1.143)
Transmission Control Protocol, Src Port: http-alt (8080),
        Dst Port: 54641 (54641), Seq: 1, Ack: 1, Len: 140
    Source port: http-alt (8080)
    Destination port: 54641 (54641)
    [Stream index: 6]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 141    (relative sequence number)]
    Acknowledgement number: 1    (relative ack number)
    Header length: 20 bytes
    Flags: 0x018 (PSH, ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgement: Set
        .... .... 1... = Push: Set
        .... .... .0.. = Reset: Not set
        .... .... ..0. = Syn: Not set
        .... .... ...0 = Fin: Not set
    Window size value: 8212
    [Calculated window size: 8212]
```

```
        [Window size scaling factor: -1 (unknown)]
        Checksum: 0xbdea [validation disabled]
            [Good Checksum: False]
            [Bad Checksum: False]
        [SEQ/ACK analysis]
            [Bytes in flight: 140]
    Hypertext Transfer Protocol
        Data (140 bytes)
         Data: 827e00884549534101016c440b9949b0aa17a24c8806e893...
         [Length: 140]

88 53 2e 22 c9 00 c0 c1 c0 d2 7d 2f 08 00 45 00   .S."......}/..E.
00 b4 bf 4a 40 00 32 06 b8 72 86 44 88 0b c0 a8   ...J@.2..r.D....
01 8f 1f 90 d5 71 2d fe d2 6c 4e dd e4 9a 50 18   .....q-..lN...P.
20 14 bd ea 00 00 82 7e 00 88 45 49 53 41 01 01    ......~..EISA..
6c 44 0b 99 49 b0 aa 17 a2 4c 88 06 e8 93 29 96   lD..I....L....).
37 88 59 59 00 00 8c 31 03 00 01 00 00 00 00 00   7.YY...1........
00 00 38 00 00 00 19 00 00 00 50 72 6f 63 65 73   ..8.......Proces
73 6f 72 50 72 6f 62 65 5f 53 6d 6f 6b 65 54 65   sorProbe_SmokeTe
73 74 00 00 00 00 00 00 00 00 56 3e 7a 00 00 00   st........V>z...
00 00 81 0e 01 00 00 00 00 00 02 4c 01 00 00 00   ...........L....
00 00 1c 00 00 00 00 00 00 00 c8 29 00 00 00 00   ...........)....
00 00 99 36 00 00 00 00 00 00 ee 33 00 00 00 00   ...6.......3....
00 00                                             ..
```