CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

PUBLISH-SUBSCRIBE ARCHITECTURE FOR BUILDING NON-POLLING ASYNCHRONOUS WEB APPLICATIONS

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

By

David Khanaferov

May 2013

The thesis of David Khanaferov is approved:

Schwartz, DianeL, Ph. D.	Date
Mcilhenny, Robert D, Ph.D.	Date
Wang, Taehyung, Ph.D., Chair	Date

California State University, Northridge

ACKNOWLEDGMENTS

I would like to thank Professor Taehyung Wang for all the help he has extended during this project. His continuous guidance and direction has made this project possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

PUBLISH-SUBSCRIBE ARCHITECTURE FOR BUILDING NON-POLLING ASYNCHRONOUS WEB APPLICATIONS

By

David Khanaferov

Master of Science in Computer Science

Globalization of network topologies and resources has lead to the proliferation of information driven system based on the HTTP protocol. However, information driven systems have much to gain from a full duplex asynchronous approach to data delivery architectures. Asynchronous data push architectures allow information driven systems to utilize bandwidth and energy resources more efficiently, while reducing latency and data inconsistency across systems. At the time this paper is written, available asynchronous technologies are not mature enough to handle full duplex data communication required by distributed information systems. Recognizing these challenges, I propose in this thesis a publish-subscribe messaging framework for web applications. A framework which consists of a publish-subscribe server, to facilitate an asynchronous message passing interface, a web application library to access the message passing interface, and a client subscriber library. The proposed framework is designed along the multiple observer pattern [1] in software engineering to allow 1 to N and N to N message passing capabilities. This thesis consists of a theoretical part describing the proposed framework

and a practical implementation of an asynchronous web application interface for communicating with mobile phone systems over short message system protocol (SMS).

1 INTRODUCTION

1.1 Information Driven Web Applications

A well established fact that the world wide web (WWW) has, for the foreseeable future, been established as the universal medium of communication between remote network resources, e-commerce applications and consumers. Designed on the application level of the OSI [1]model, the web applications are on the highest level in the network stack. The driving engine of the web is the hypertext transfer protocol (HTTP). At the time of creation and after several revisions, HTTP protocol designers envisioned a light weight protocol capable of driving development of sophisticated web applications. Aggressive growth of HTTP's network topology, as well as exponential growth of data, resulted in the evolution of information driven web applications. According to a study done at Packetcom, between 1997 and 2008 the rate of growth of internet traffic has doubled every six months. Extrapolated over a period of ten years, the presented results yield an exponential growth curve [29].

Information driven web applications are based on content centric, exploratory access to information [27]. Rather than predefined dialog of information such as those in form-based interaction, information driven systems are inherently dynamic. A variety of information is exchanged between content producers and consumers in structured and unstructured forms [28]. In addition, the state of information is no longer simply dynamic, rather edging closer to real-time. Prior to the epoch of information driven systems, the state of information on the web was notoriously static. Even with

1 OSI - Open Systems Interconnection (OSI) model SO/IEC 7498-1

dynamically generated content of the Web 2.0 era, non-asynchronous architecture of the HTTP protocol prevented near real time communication. It is a well established fact that HTTP is not a good fit for low-latency services, such as VoIP, chat, and other real-time applications [27]. Although information driven web applications do not require real-time communication, the benefits of publishing events to subscribers in near real-time environments makes user interaction more responsive and agile.

Whether referring to software systems' inter-communication or user interaction with data driven applications, the underlying concept is the same, “we are living in the data age” [10]. Great examples of information driven applications are social networks, which are solely based on delivery of data to users and systems. Social networks' data changes constantly and rapidly, postulating continuous consumer change notification. Whether the consumer in this case is a user or an external system, using a data access application programming interface, the data state change notification remains request based. Social network applications are not capable of ubiquitously pushing data state changes to all consumers because of the inherent non-asynchronous nature of the HTTP protocol. Proprietary push notification technologies, such as those in mobile phone operating systems, as well as data polling techniques exist allowing consumers to work around the limitations of HTTP. However an open standard, real-time push notification of information remains unattained.

1.2 Asynchronous Server Push

Several architectures have been developed to work around the lack of asynchronous capabilities of HTTP. A non-exhausting list of available technologies includes: Comet, Service Streaming, Hidden iframe, XMLHttpRequest long polling, Script tag long polling, and raw TCP sockets with browser plug-ins [30]. Each technology solves the underlying issue at the application layer, however the transport layer remains locked into the request/response model. Some of the existing solutions are just creative ways to use the available technology to simulate asynchronous communication. While the *hacks* work on small size applications, they lack scalability and re-usability. In addition, due to the fact that the aforementioned techniques are non-standard solutions, there remains lack of ubiquitous support by the developer community and recommendations from World Wide Web Consortium (W3C) . In this thesis I propose a framework which works along with the currently available technologies at every layer: physical, transport, and application. To comprehend the necessity for a proposed framework I will discuss, in the following sections, some of the available asynchronous techniques and key missing aspects that

Push	Support	Secure	Resources*	Name
N	All	Y	1	Session Streaming
N	All	Y	1	Long Polling
Y	Latest [1]	Y	-1	WebSockets
N	All	Y	1	Hidden iFrame
Y	Fragmented [2]	N	0	Raw TCP sockets
Y	All	Y	-1	Proposed Solution

* Resources are rated between -1 and 1. The lower the number the lower the amount of required resources.

1 Latest refers to lack of support by older (legacy) browser software.

2 Fragment support refers to lack of cohesive standards for implementation.

Table 1: Survey of current asynchronous technology paradigm

yield the need for an improved solution.

Table 1, describes the results of a survey of current asynchronous server communication technologies. The table categorizes six different server push implementations, including the proposed solution. The *Push* category refers to whether server push is implicitly supported or simulated. The *Secure* category refers to whether or not security is seamlessly integrated with HTTPS[2] protocol. Finally, the *Resources* property defines a rating between -1 and 1 for amount of network and computation resources required for implementation. The lower the rating represents least required resources; higher ratings categorize more resource hungry algorithms. In the following sections I delve deeper into the most conspicuous of the surveyed technologies.

1.2.1 Service Streaming

Service streaming is one of the first and oldest attempts to simulate HTTP push technology. Service streaming is based on the XMLHttpRequest specification [11]. Defined by the W3C web standards specification as "an API that provides scripted client functionality for transferring data between a client and a server"[11], earliest drafts were first introduced around 2006. XMLHttpRequest specification allows for data to be transferred between server and client over an HTTP session without posting an HTTP request to update the entire page. However, unlike the standard short lived data transfer from the server, session streaming keeps the session alive indefinitely. The server script employs an unbounded loop to keep the session open and stream information to the client as data becomes available [5]. Similar loops are instantiated on the client side to check

2 HTTPS – layering the Hypertext Transfer Protocol (HTTP) on top of the SSL/TLS protocol, which adds the security capabilities of SSL/TLS

for available data in the data stream. From a high level view the server asynchronously notifies clients as data becomes available. The trouble with this approach is that the server must maintain the connection in open state regardless of whether there is data to be transferred. This approach lacks scalability as each additional client requires a small (but dedicated) amount of memory and processing power. When multiplied over a large set of clients, the dedicated resources begin to adversely affect server performance.

1.2.2 Long Polling

Comet, hidden iframe streaming, and XMLHttpRequest technologies are diverse in implementation yet based on a analogous principle called long polling [5]. Long polling consists of an open HTTP connection created by the client that remains open on the server for a specific period of time. During this period the server continues to send data to the client. The server closes the HTTP session session once the configured polling period expires causing the participating client to restart the polling mechanism by requesting a new session. The key difference between streaming and long polling is the timeout period. Preventing an indefinite HTTP session allows the server to manage processing resources efficiently. Of the above mentioned technologies, Comet is the most sophisticated and successful approach to HTTP push [5]. Comet defines a Bayeux protocol, which delineates a topic based publish-subscribe architecture. Using long polling as a transfer layer protocol Bayeux allows servers to publish events (messages) to all subscribing clients. From the application layer perspective, messages are pushed to listening clients asynchronously, eliminating client side polling of the HTTP session. The Bayeux protocol adheres to the HTTP 1.1 specification which limits the number of

concurrently open HTTP sessions per client to no more than two, thus allowing for a full duplex connection [3]. From the transport layer perspective the Bayeux protocol does not support true asynchronous HTTP push, rather a simulated long polling loop [6]. Moreover, web servers must be designed to understand the new protocol at the time this paper was written, many major enterprise web server implementations lack Bayeux compatibility. Lastly, because the message passing interface and the server are one entity, there is a significant increase in required processing power. An increase in processor load is attributed to the fact that the server is required to keep state of the open sessions in addition to its normal server tasks. A survey done by Engine Bozdag, Ali Mesbah, and Arie van Dueren on the comparison of push and pull techniques for AJAX shows that as the number of subscribing clients increases linearly, the mean server CPU usage increases exponentially [5]. As a result, in order to build distributed applications using Comet, there is an immediate need for a load balanced system to be able to serve a large number of clients.

Comet and the Bayeux protocol lack a key component from a publish-subscribe model: space decoupling is not inherently present in the comet implementation. The Comet server is actively participating in managing state of the HTTP

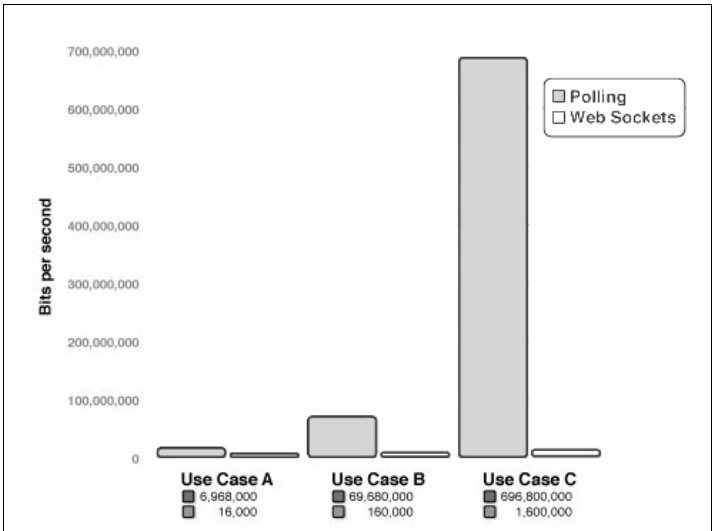


Figure 1: WebSockets vs. Polling bandwidth [12]

sessions and is thus directly aware of all subscribing clients. It is exactly this missing feature that causes the exponential increase in required processing power to handle large numbers of clients.

1.2.3 WebSockets Protocol

Part of the newest suite of technologies currently under active development is a bidirectional messaging protocol called WebSocket protocol. A draft of the proposed standard for a new protocol is defined in RFC[3] 6455 by the Internet Engineering Task Force in conjunction with research from Google Inc. as part of the HTML5 specification [13]. "Web sockets don't replace HTTP. Rather, much like BSD[4] sockets, they provide bidirectional, long-term communication between two computers" [14]. Achieved through a handshake, which facilitates a switch away from HTTP to the WebSocket protocol, the browser's JavaScript engine establishes a concurrent session with the server. Theoretically, WebSocket protocol does not define a limit on the number of open sessions, although various JavaScript engines place restrictions, attributed to performance reasons. A bidirectional nature of the WebSocket sessions allows the communicating nodes to send and receive data over the same connection [14]. In addition, the new protocol supports a publish-subscribe architecture for either topic or queue subscriptions. Under true publish-subscribe topic communication, several subscribing clients can simultaneously receive updates asynchronously, independent of the individual client's requests.

3 RFC – Request For Comments, memorandum published on behalf of the Internet Engineering Task Force, describing methods, behaviors, research to the workings of Internet-connected systems.

4 BSD – Berkeley Software Distribution is a Unix distribution developed by the Computer Systems Research Group at the University of California Berkeley

Research performed to identify resource utilization by asynchronous publish-subscriber architectures has shown dramatic results. Figure 1 shows a comparison of unnecessary bandwidth overhead in both polling and asynchronous applications. Defined in bits per-second, the figure shows the bandwidth required for protocol handshakes, during the exchange of information between server and client nodes. Compared at different bandwidth levels (use-cases A though C), the bits per-second of bandwidth required for overhead was recorded. The results are overwhelmingly in favor of the WebSockets protocol. Figure 1, shows the amount of unnecessary traffic as the number of clients grows linearly from 1000 (use-case A) to 100,000 (use-case C). The graph shows that the unnecessary traffic calculated for a polling architecture reaches upwards of 87 gigabytes per second, while WebSockets bandwidth remains around the 0.2 gigabytes mark [12].

WebSocket protocol will without a doubt change the way web applications are designed, however, lack of backwards compatibility with older generation web browsers will stagnate it's wide spread adoption for the time being.

1.2.4 Proposed Solution

WebSocket protocol is a lightweight and powerful solution for integrating server push technology into information driven web application architectures. However, lack of ubiquitous platform support across the industry renders this protocol not mature enough to serve as the basic server push architecture. This thesis proposes a server push architecture supported by all current and older generation client browser software. The design and implementation of the proposed framework is discussed in detail in the following sections. The new communication framework is fully generic, allows for

streamlined development, and can be integrated into any web application. Based on a messaging framework, the proposed solution consists of a set of libraries to simplify asynchronous communication between framework modules. Each module in the framework helps streamline development of application logic.

The new framework decouples several aspects of web application engineering. Developing software business logic is an essential task that takes precedence over implementation details. Business logic encapsulates the application software requirements necessitating most design considerations, development, and testing resources. The proposed framework abstracts away the implementation details of data object transfer, permitting developers to focus on development of application logic. Developers can rely on the framework to deliver required data state changes to subscribing entities.

In Table 1, the proposed solution is compared against other competing technologies. The results show that the proposed solution supports native server push and security. In addition, the proposed framework is compatible with a majority of client software and requires a low amount of computation resources. Server push technology in the proposed solution is supported through a set of libraries that abstract an interface into a generic message passing system implementing a publish-subscribe architecture. There are many messaging framework products available, both proprietary and open sources, such as, Data Distribution Service (DDS), OpenDDS, and Windows Communication Foundation. In this thesis I advocate using open source software for all components. A cost-benefit analysis will easily show advantages for both development and production environments.

For instance, Apache web server is one of many open source products used throughout the industry. According to a study done on the economic benefits of open source software, the use of an open source Apache web server has saved society approximately 128 billion dollars when compared to costs associated with Microsoft alternatives [15].

A messaging framework could be any software facilitating a loose coupling between producers and subscribers of data events. As a concrete implementation, I decided to use the Java Messaging Service (JMS). JMS is a set of open standards for which multiple vendors provide implementations, thereby helping to avoid the dreaded *vendor lock-in*[5] problem. Advantages over using Comet with Bayeux protocol is the aforementioned loose coupling between messaging end points. Comet requires event producers to keep an explicit reference to subscribing endpoints and propagate notifications accordingly. In a loosely coupled messaging framework event producers and consumers maintain no reference to one another delegating the communication concerns to the framework.

The proposed framework relies heavily on the principle of delegation of concerns, thus forcing development of modular web applications. Delegation of concerns is a concept of separating a problem into a set of smaller, less complex and easily addressable problems. Existence of homomorphic relationships between system requirements and testable artifacts allows for delegation of concerns between system modules. In section 2.2 of this paper, I describe in greater detail the methodology behind separating the proposed framework into a set of loosely coupled modules.

5 Vendor Lock-in – organizations frequently choose to use open source software to reduce dependency on their software vendors. Organizations locked in to one vendor depend on all products and services, patched updates and support from the vendor. Open source software advocate support for open standards which promotes development of compatible products [16].

2 DESIGN

2.1 System Architecture Survey

A set of well known design patterns have been developed to address pressing issues in web application development. Web applications differ from non-distributed systems, therefore several aspects to highly distributed systems must be addressed. Distributed applications operate over a global network where individual clients serve one of two roles: producing data or consuming data. In general, web applications fit well into the above model. Though, data is the focal point of distributed applications, storage and content delivery are key outstanding issues that need to be addressed. Various types of data procurement and storage paradigms have emerged. Relational database management systems (RDBMA), advanced key-value (no-sql) storage engines, object oriented databases, and semantic databases are all examples of data warehousing techniques in existence today [10]. The power of distributed systems is the inherent ability to deliver data to any node on the network at any time. Caused due to inadequate network infrastructure, issues with bandwidth and latency, among other key factors, affect data state. Caching techniques allow applications to marginalize bandwidth and latency issues, while increasing the chances of propagating stale data. Improvements in synchronization techniques such as research done by engineers at Google Inc., allow globalization of data infrastructure. Spanner, Google's globally distributed database, uses advanced atomic clocks implemented over the global positioning network as well as a proprietary Google Time API, to synchronize database operations [17]. Globally-distributed databases such

as Spanner, provide several interesting features. Including dynamically controlled replication, dynamic data-center selection and application constraints. In addition, globally-consistent reads and writes across the database allow for control of durability, availability, and read performance. It is evident that a tremendous amount of research, time, and money has gone into the development of data storage techniques. Information driven web applications benefit most from such globally available data engines.

In this thesis I proposed developing software for web applications along a well known publish-subscribe architectural design pattern. In order to understand the reasoning behind said selection I present a set of other available design patterns, their strengths and shortcomings. To identify the best approach to developing asynchronous web applications, I surveyed a subset of software design patterns. Identifying the best design pattern proved to be a challenging, yet necessary starting task. Through my research and professional experience I was able to compile the following list of software design patterns most applicable to the problem at hand. Table 2 describes six patterns identified in my survey [30].

	Server Push	Latency	Complexity	Name
1	No	High	Low	Request based MVC
2	Simulated [1]	High	Medium	Asynchronous MVC
3	Yes	Low	High	RPC(Remote Procedure Call), RMI (Remote Method Invocation)
4	Simulated [1]	N/A [2]	Low	Observer Pattern
5	No	High	Low	Headless SOAP/REST
6	Yes	Low	Low	publish-subscribe

*1 - Simulated server push refers to achieving similar effect to server push though various implementations of long polling.

*2 - Observer pattern can not be implemented on it own. Depending on whether implemented with RMI, MVC or publish-subscribe the latency results can vary from Low to High

Table 2: Survey of software architecture design patterns

2.1.1 Model View Controller

Model View Controller (MVC), is an object oriented software design pattern which gained ground with the advent of SmallTalk[6] family of programming languages. MVC consists of three kinds of objects: a model to describe the data structure definition, a view which describes the user interface and a controller, encapsulating the business logic of the application. MVC decouples the three layers allowing for flexibility and increased reusability of code. The three layers communicate over an established publish-subscribe framework. Changes to any of the three layers trigger cascading updates throughout the system [18]. The above pattern fits perfectly into the distributed application architecture, hence the reason for wide adoption of this pattern into to web application engineering practices. Request based MVC is an adaptation of the model view controller pattern to the hyper text transfer protocol based web architecture. The model is commonly specified as an XML based domain specific language, strictly defined in an XSD document. This allows engineers to focus on developing decoupled user interface which utilizes HTTP methods to communicate with the controller. Due to the nature of the request response architecture, request based MVC lacks the ability to provide server push functionality. Theoretically, MVC design pattern specifies communication through publication of model change events, however request based MVC is locked out of receiving the events asynchronously.

Asynchronous MVC is an improvement over the fist surveyed pattern. Model changes can be propagated to the view and controller asynchronously, however only through one

6 SmallTalk – Generally accepted by the industry as the first attempt at purely Object-Oriented programming languages beginning with SmallTalk-80 made publicly available in 1980's.

of the available long polling implementations. As described earlier, long polling allows the view component to indefinitely maintain an open communication session. Polling the controller for model changes allows this software pattern to simulate server push. Unfortunately, polling results in higher resource utilization. In addition, the need for careful management of resources increases development complexity [18].

2.1.2 Remote Procedure Call

A stark contrast to MVC design pattern is the remote procedure call (RPC) pattern. A more modern RPC implementation, included with the core Java packages, is the remote method invocation (RMI). The two described patterns, allow remote clients to obtain a reference to a server object and invoke methods on the remote object nearly as easily as a local instance. Object data is marshaled across the network topologies using object serialization. Either XML or JSON serialization is used to send data between remote hosts. In addition, to execute code via remote proxies RMI, supports the dynamic procurement of dependency resources across the network [19]. XML-RPC is the first specification to use XML implementation of RPC for remote procedure calls. A robust protocol, simple object access protocol evolved from the earlier specifications [20]. Web applications can be developed using the RMI pattern by implementing SOAP communication pattern to execute remote methods. Implicit bidirectional remote method invocation is a major advantage to this design pattern [21]. Java RMI has greatly simplified the complexity involved with remote method invocation. Java's Remote interface handles much of the low level serialization and method invocation details. Application developers working with RMI, write the business logic of the application and

delegate communication logic to the system. However, the simplified development bring a certain level of fragility. SOAP web services require a web service definition language (WSDL) document to advertize the RMI interfaces and define a data model language. WSDL definitions use strict XML, thus modifications may require clients to rewrite major portions of their implementations. From the business perspective, inherent fragility of the SOAP web services may result in a maintainability nightmare.

2.1.3 Headless SOAP/REST

Unlike the previously discussed design patterns, which define end-to-end communication architectures, the Headless SOAP/REST pattern defines only the service end point. The term "headless" refers to lack of user interface or external system overhead. Typically, this design pattern is used to define a set of web services which provide an entry point to any authorized party and execute the application business logic. Good examples of this type of pattern are cloud API's available for web developers today from social networks, search engines, and other content providers. A set of web service calls is defined and published to developers of rich internet clients. Said web services grant to developers granular access to specific parts of the system. The simplicity of the above approach is appealing, however due to

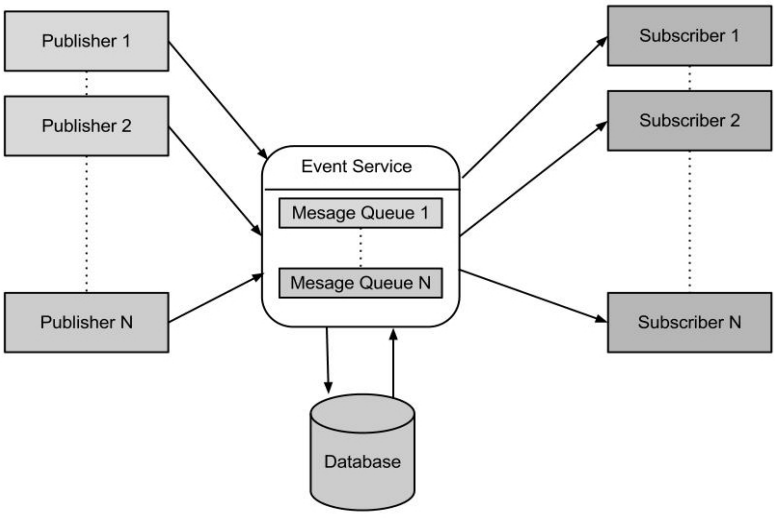


Figure 2: publish-subscribe Architecture

the fact that server push isn't easily implemented, long polling remains the only alternative. Web services designed along the above pattern can not afford to trust management of resources to clients, and must carefully monitor for abuse or negligence. As a result, long polling doesn't fit into the security model of the headless web service design.

2.1.4 Multiple Observer Design Pattern

The observer pattern was devised to reduce tight coupling of cooperating classes in a distributed system, and to increase reusability. The *Gang of Four*[7] define the observer pattern with two key relationships: subject and observer. Subject is an entity which may have a one-to-many relationship with dependent observers. When the subject undergoes a state change, all dependent observers are notified, each observer then queries the subject to synchronize its state [18]. The observer pattern allows the designers to vary subjects and observers independently. Reusing and modifying subjects independently of observers and vice versa allows for great flexibility in designing highly distributed modular applications [18].

Unfortunately, the observer pattern does not specify the mechanism for facilitating the notification other than maintaining a list of dependent observers in each subscriber's state. For large scale applications, the coupling between subjects, observers, and the notification mechanism turns into a performance bottleneck. A better approach is to decouple the messaging mechanism into a separate entity, thus allowing observers and subjects to scale independently. publish-subscriber pattern is an implementation of the

7 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch, authors of the book, *Design Patterns: Elements of Reusable Object-Oriented Software*

observer pattern with the addition of the independent messaging manager entity to facilitate asynchronous subject state change.

2.1.5 Publish-Subscribe Architecture

Traditional publish-subscribe models require decoupling along three distinct dimensions: space decoupling, time decoupling and synchronization decoupling. Space decoupling refers to interacting parties being unaware of each other. Publishers are tasked with creating events without directly being aware of which clients (if any) will be receiving the published events. Subscribers only know of their subscriptions, oblivious to the source of received data [2]. Time decoupling removes the restriction of requiring subscribers to be actively listening in order for publishers to produce events. Clients may or may not be listening to events at the time of the publications. Only currently listening subscribers will receive events and publishers will not know whether events were consumed [2]. The responsibility of tracking message delivery status is delegated to a third party entity called an *event service.* The last dimension, synchronization decoupling, states that publishers are not blocked from producing events while subscribers are consuming previous publications. With all three dimensions, a publish-subscribe model defines a distributed, asynchronous communication framework. Figure 2, shows a diagram of a traditional publish-subscribe model.

2.2 Proposed Web Application Framework

Having compared the available design patterns identified in Table 1, the publish-subscriber framework fulfills all the requirements for data driven web applications. publish-subscribe architecture allows for low latency, high availability, scalability, as well

as asynchronous server push and low complexity of implementation. The only area left undefined is the implementation of the asynchronous server push. As mentioned earlier, WebSockets protocol can be used to implement the required behavior. However, due to lack of backwards compatibility with older generation client software, WebSockets protocol is not a viable option. According to web statistics organization GlobalStatistics, at the time that this thesis was written *Microsoft Internet Explorer* and *Google Chrome* held over 70% of the web browser share [22]. Unfortunately, Internet Explorer, with the exception of the most current version 10, does not support the WebSockets protocol. In addition, not all legacy versions of the Chrome browser support the new protocol. When designing a web application a more widely supported technology shall be used.

In this thesis I propose using the Java Applet plugin, supported by all major browsers, to implement the asynchronous server push transport layer. Java Applets operate as embedded objects in the browser software. Applets are executed by the Java Virtual Machine software sending any user interface, events, notifications, or interactions with native environments to their container. Embedded into a browser, the Applet is capable of communicating with the JavaScript engine propagating events, to registered JavaScript event handlers. In this model the existing request based HTTP model is not replaced, rather extended with the asynchronous capabilities. The embedded Applet receives server push events generated by the messaging entity as described in section 2.1.5. All events are then sent to the JavaScript engine of the web application. The events passed between the messaging entity and the client subscriber are serialized into a JSON object prior to marshaling. Once received by the JavaScript engine, deserializing the received message

content is trivial using the *eval()* function defined in the ECMA-262 Specification[8] [23].

The web application middle layer will consist of data event producers, either a set of daemon services or a web server. The middleware, will enqueue events on the messaging entity by implementing message queues or topics. The messaging entity will be a decoupled independent application server tasked with ensuring delivery of the data events to the connected Applets. The messaging entity will implement persisted queues and topics using a persistency layer such an RDBMS or a key/value storage engine commonly referred to *NoSQL [24]*. Persistent queues and topics ensure fault tolerant delivery of messages. Although not a radical novelty, the idea of decoupling middleware into producers and an independent messaging entity allows for greater reusability of available resources.

An important point to understand is that, as applications grow in the number of consumers, the producers should not grow at a one-to-one rate. With the proposed framework, producers are not required to scale at a linear rate compared to consumers. Messaging entity can grow vertically not horizontally in the amount of required computing power. Vertical growth refers to the addition of low cost, commodity hardware to the pool of available resources, which increase the throughput of the entire system. In the proposed framework, adding an instance of a messaging entity is relatively simple. A basic load balancing algorithm, such as *round robin,* may be applied to a set of messaging entities. Queues and topics reside in a pool shared among all messaging entities, while the processing engines are load balanced across a cluster of servers. Middleware producers

8 ECMA-262 Specification defines the ECMAScript scripting language. The popular JavaScript language supported by all major browsers is an implementation of the ECMA-262 Specification [23].

are left unaware of the size of the cluster and continue to produce events to queues. Similarly, consumers are notified of data events with no insight into which messaging entity delivered the event.

The decision to use a Java Applet does not limit the middleware to any specific language implementation, because the serialization of data events into JSON objects is language agnostic. As a practical implementation of the proposed framework an application called TextBox was developed using the Java programming language. TextBox relies on the proposed framework and consists of two daemon services and a web application server in the middle layer. Message events are propagated to Applet consumers by a decoupled JMS broker. In this example the JMS broker serves as the messaging entity.

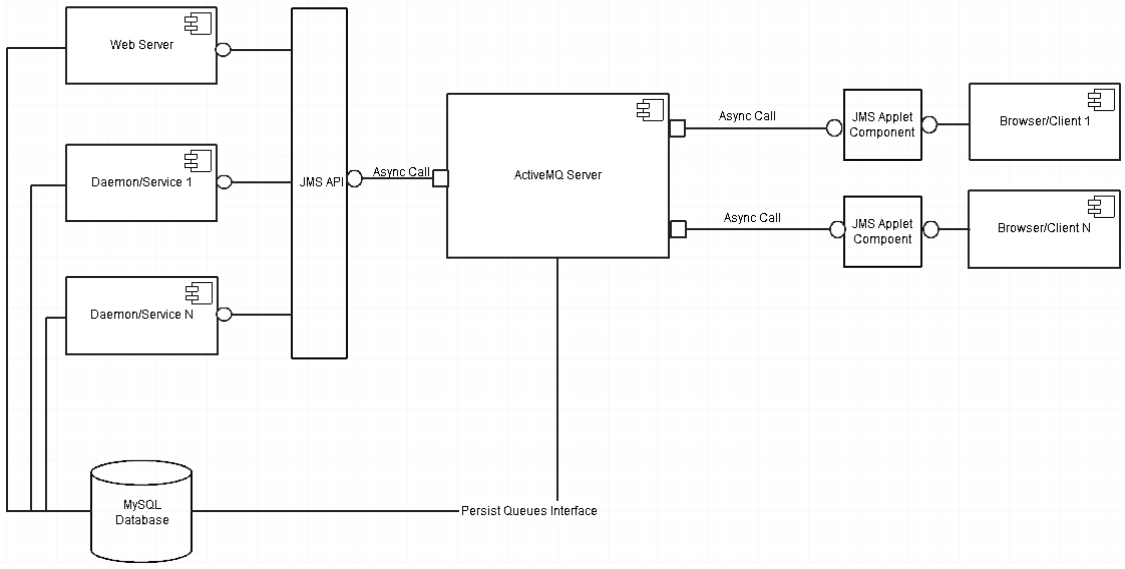


Figure 3: Proposed JMS Implementation of publish-subscribe framework with asynchronous server push

2.3 JMS Implementation

Figure 3, illustrates a high level component diagram of the framework implemented with

the JMS protocol. As mentioned earlier, a JMS broker is used to facilitate the publish-subscribe architecture. The JMS broker concept is an abstract entity defined in a set of interfaces in the JMS API; a concrete implementation of the interfaces is required to use the protocol. Several implementations of the JMS broker API exist, yielding a need to select the most appropriate implementation. The power behind defining brokers as a set of abstract interfaces allows any implementation to be plugged requiring no change on the producers and consumers.

The selection process of a JMS broker yielded a survey of the available products. SPECjms2007 is an industry standard benchmark for JMS, developed by Standard Performance Evaluation Corporation (SPEC) [8]. SPECjms2007 results show several JMS server implementations including: ActiveMQ[9], JBoss Enterprise Application Platform 5.1.2,[10] and HornetQ. The benchmark results published by SPEC, indicate that from the two open source JMS servers, HornetQ slightly outperforms ActiveMQ in almost all hardware and software configurations. The results also show that proprietary JBoss Enterprise Application Platform 5.1.2, delivers the highest JMS performance [9]. For my proposal I selected ActiveMQ due to several factors. ActiveMQ is an open source application from a highly reputable Apache Foundation. In addition, ActiveMQ is a much better documented and polished product. The difference in benchmark results between HornetQ and ActiveMQ is not significant enough to sway the decision for either product.

9 ActiveMQ is an open source Apache messaging and integration patterns server

10 Jboss Enterprise Application Platform and HornetQ are both Red Hat products. HornetQ is an open source messaging syerver while, the Jboss Enterprise Application Platform is suite of proprietary server products.

2.3.1 TextBox

TextBox is a data driven web application designed to use the proposed asynchronous application framework. The goal of TextBox is to define a web interface for sending and receiving data over the short message service protocol used in mobile wireless networks. At the highest level, TextBox application facilitates management of data message state across multiple protocols including, SMS and HTTP. The above definition qualifies TextBox as a data driven application. A data driven application design can be easily modeled using a data flow diagram (DFD). Although not part of the standard UML specification, DFD's are a great way to describe the flow of data between system components. Refer to Appendix D for a complete set of DFD diagrams for the TextBox application.

The asynchronous web application framework design for TextBox includes four top level components. A web server (with a built in web Servlet container), a JMS broker implementation, a Java Applet JMS client and a JavaScript library designed to interface with the Applet. In addition to the web server, the TextBox middleware contains two daemon services, used to delegate SMS gateway communication.The middleware of this web application is designed along the headless REST web services pattern described earlier; the user interface is designed along the publish-subscribe framework. Two sets of web service API's are defined as independent entities and provide access to the persistency layer though a JDBC[11] interface. The first set of web services exposes an endpoint into the TextBox system for a third party SMS Gateway partner responsible for

11 JDBC is a Java Database Connection interface which allows access to any ODBC compliant RDBMS software

SMS operations. The second set of web services is used by the TextBox rich web client application for user interaction.The user interface of TextBox is decoupled from the services layer which follows for a modular design. Modifications to the services layer do not affect the user interface and vice versa.

TextBox uses a custom built JMS connection pool, implemented using a ThreadLocal class available from the core Java library. Each Servlet request thread obtains an instance of a JMS connection through a JMSProviderFactory class. A private connection object, with thread scope, is instantiated and its reference is stored in the ThreadLocal container. Access to the provided connection object is restricted to the executing thread. Each JMS connection remains in memory until the thread is terminated or a method in the JMSProviderFactory class is called to explicitly close the connection. Appendix E contains the complete code of the JMSProviderFactory.

A wrapper to the *ActiveMQConnectionFactory*[12] class was designed to encapsulate most of the complexity in configuring persistent connections, sessions, and producer/consumer objects. Exposing a predefined set of functionality, the wrapper allows TextBox application to produce events to a queue or a topic. In addition, the wrapper allows a thread to subscribe to incoming messages from the JMS queues. Together the wrapper and connection pool class along with several helper classes define a framework for sending and receiving asynchronous messages though the JMS broker.

The TextBox application is designed to use JMS messages to send notification to the external user interface as well internal component. Internal components such as the

12ActiveMQConnectionFactory – Java class from the ActiveMQ API is the main entry point into the ActiveMQ JMS implementation.

services API and daemon threads use JMS to delegate business logic execution, which allows TextBox modules to be completely independent. Delegation of concerns is described further in section 2.4. Traditional web applications use a mixture of dynamically generated HTML on the server through either Java Server Pages, PHP, ASP.NET etc.TextBox, however, is built purely in JavaScript. HTML components are generated on the client side and all required data is retrieved from the web service endpoints. Furthermore, TextBox user interface embeds the Java Applet client allowing all messages from the JMS broker to be propagated asynchronously. Refer to Appendix F for a full listing of a simple JavaScript library developed to integrate the JavaApplet message events into the TextBox user interface. Upon initialization, the JavaApplet attempts to establish a connection with the JMS broker using a TCP/IP socket. Once successful, a JMS connection is created then the server push session begins.

Figure 4 shows a UML sequence diagram describing the sequence of function calls which occur when a send message operation is requested from the user interface. The request is initiated as an AJAX request and is received by the services API layer. Depending on how the web application is deployed, it's possible to run into the cross domain scripting problem with the proposed design. Several solutions exist as work arounds, however, this issue is out of the scope of this paper.

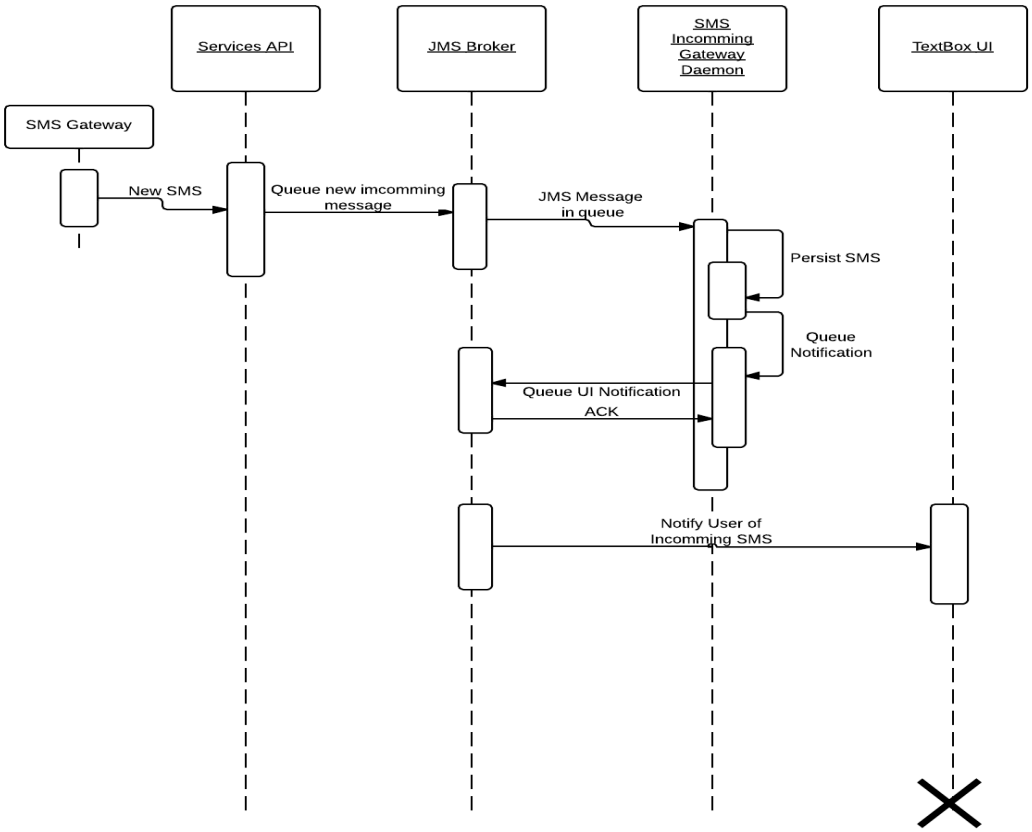


Figure 4: TextBox Sequence Diagram: receive message

Business logic of the application dictates that a new SMS message must be persisted in the data layer which is a synchronous (transactional) operation executed by the services API layer. Once the persistence transaction is committed, a JMS message is sent to a queue to notify the SMS Gateway service of a new outbound message. Earlier I discussed that the TextBox application was designed as a set of decoupled modules. This is an example of the aforementioned decoupling. The SMS Gateway service is an internal module; the services API module communicates with the gateway service through JMS queues. The final step in the sequence involves the SMS Gateway service processing the queue and sending an SMS message to the third party service.

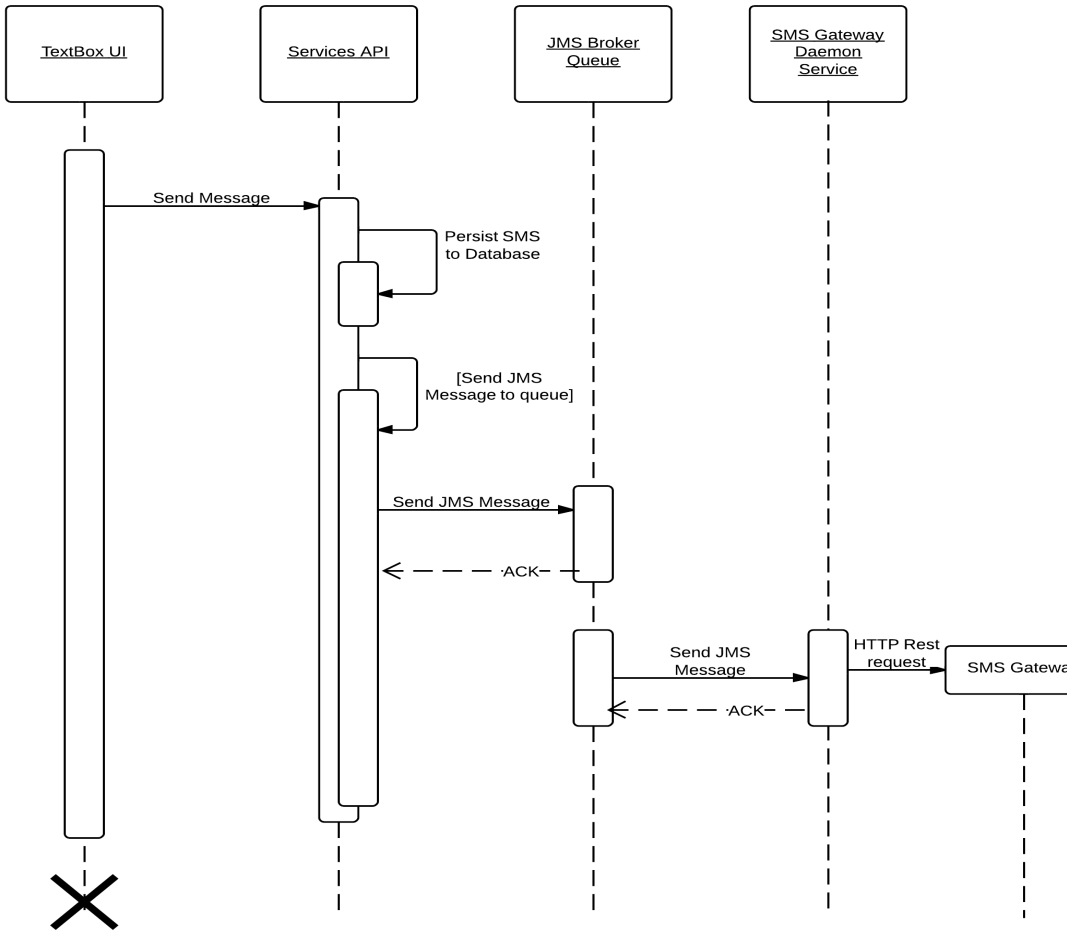


Figure 5: TextBox Sequence Diagram: send message function

Illustrated in Figure 5, is the sequence of events which occur when the external SMS gateway provider executes a web service call from the Services API. Figure 5, shows two events propagated by the JMS broker. The first event occurs when the internal daemon service receives a notification from the Services API layer. The second event occurs after the business logic is executed and a notification is sent to the JMS broker to propagate asynchronously to the TextBox user interface. JMS broker delegation to facilitate communication allows the ServiceAPI layer to be completely unaware of the daemon services both in the incoming and outgoing cases.

3 DESIGN CONSIDERATIONS

3.1.1 Business Logic Executor

In the previous section I presented sequence diagrams for incoming and outgoing message use cases. Each operation is considered an independent task in a chain of business logic steps propagated though the sequence using JMS messages. In the proposed framework I treat individual tasks as *operations*. An *operation* is a business entity; it encapsulates a specific set of short tasks (which could be transactional) and terminates with a result. Operations require an executer to process and return their result. Chaining the operations in specific order defines the application business logic. In addition, by defining exception and validation handlers developers can write clean, readable and easily reusable code.

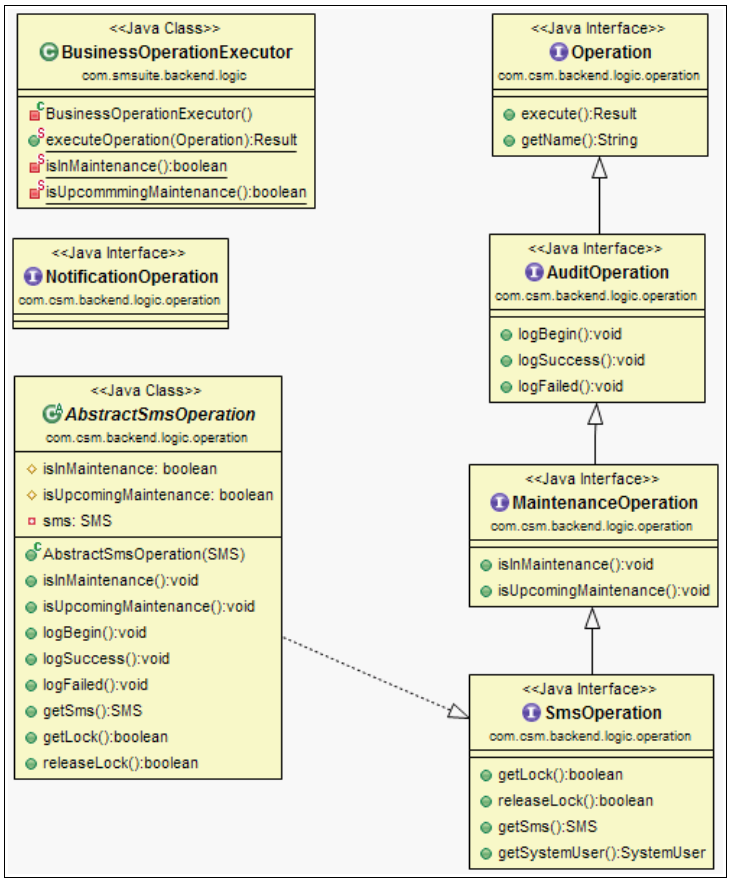


Figure 6: Class diagram: BusinessLogicExecutor

I created a class called BusinessOperationExecutor responsible for executing operations. Operations are Java objects decorated with a set of custom interfaces, defined in the proposed framework. Each custom interface follows an inheritance hierarchy, which

beings by extending the Callable interface from Java's concurrency library. Using polymorphism, the BusinessOperationExecutor class is able to determine the flow of execution for each individual operation object. For example, Figure 7 defines the activity UML diagram for the TextBox implementation of *BusinessOperationExecutor*. In Figure 7 we can see a complex flow of execution defined at each step by the *type* of operation. For example, in the first step the executer decides whether or not to log the operation by checking if the operation object implements the AudiOperation. Note that the details of how the operation gets logged are left for the operation object to implement. The executor does not define the logging mechanism, rather, using polymorphism the class is able to typecast the operation to the correct interface and execute the logBegin() method. Similarly other interfaces can be used to decorate operations: *SmsOperation* identifies if the operation requires a database lock and provides a method to obtain a lock ID. *BusinessOperationExecutor* will obtain the lock ID before executing the operation and *MaintenanceOperation* defines this behavior under system maintenance. Finally, since each operation is required to implement the Callable interface, the businessOperationExecutor is able to execute operations in a batch using Java's ExecuterService from the concurrency library.

Class diagram illustrated in Figure 6, defines a complete set of available decorator interfaces. One of the interfaces defined in the diagram is the *NotificationOperation.* This is one of the most important interfaces in the entire business logic framework. At a closer look, The *NotificationOperation* interface does not define any methods. Instead the said interface uses a set of custom Java annotations to turn implementing operation objects

into a JMS aware entities. Decorating an operation object with *NotificationOperation* forces to the BusinessLogicExecutor to delete JMS Notification logic upon successful operation result. BusinessLogicExecutor delegates this functionality to another service called, ProducerMessagingService, an interface that defines two simple methods: *sendMessageToQueue()* and *shutdownMessagingService()*.

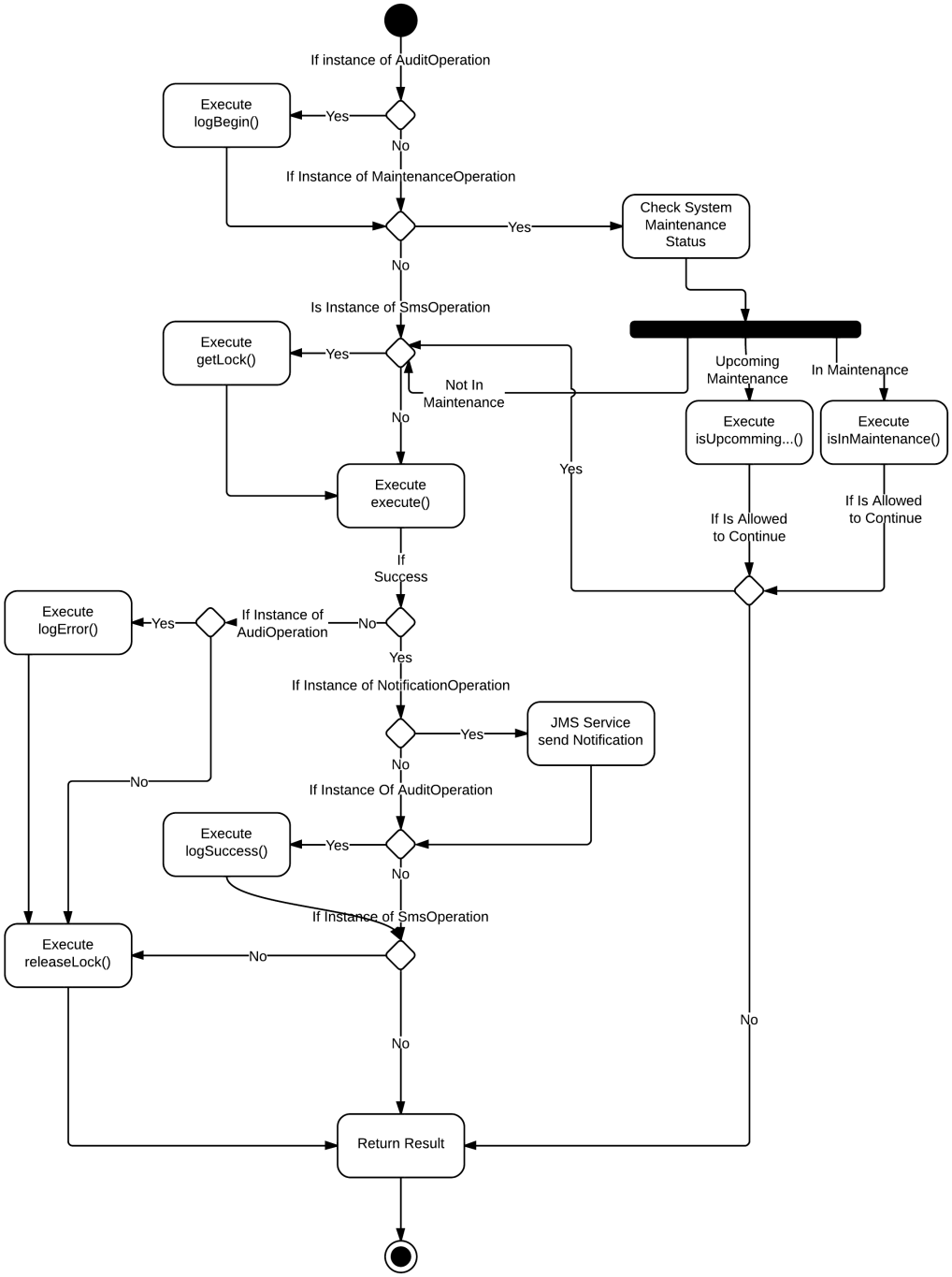


Figure 7: BusinessOperationExecutor activity diagram

Each operation decorated with the *NotificationOperation* interface must also be decorated with a set annotations [25] introduced to Java in version 1.5. The annotations, created as part of the proposed framework, allow the ProducerMessagingService to identify a destination queue name, as well as a serializable JMS message content. Additionally, the annotations dictate the encoding type for JMS message content. A set of encodings supported by the framework includes: XML, JSON, plain text, and key/value pair. Together, the business operation executer and the producer messaging service define a very robust framework for creating implicitly JMS aware operations. The TextBox business logic layer defines a set of operations based on the described framework.

3.2 Delegation of Concerns

Throughout the previous sections I described the system architecture of the proposed framework. I defined modules to execute independently and communicate with one another using a messaging framework. Also known as “separation of concerns”, the concept of breaking down the complexity of a problem into loosely-coupled subproblems is the driving methodology behind the proposed framework design. Seen as a perennial quest for modularization of boundaries best representing original requirements, delegation of concerns is an inherent design idiom behind many software engineering design patterns [26]. The main problem giving rise to the concept of delegation of concerns can be summarized as the following: when the artifacts of a system become conceptually complex, methodological tools used to produce the artifacts fall apart due to scalability. Scalability of a system with complex requirements is the most prominent issue to significantly affect system design [26].

I am not attempting to make an argument for a modular software system design. The said concept is a fundamental part of modern the software engineering paradigm. However, I'm suggesting that implementing inter-module communication through the publish-subscriber architecture will alleviate issues with aggregation of module data into a cohesive result. There are two major questions which need to be answered when designing a modular system: which requirements can be modularized, and how to preserve relations between requirements in the modularized system [26].

The methodology used in identifying requirements delegation to independent modules was based on database transaction analysis. Distributed data driven applications often rely on the persistency layer to synchronize operations. Operations which persist data must ensure data consistency throughout the system accounting for possible failures. In many cases, partially committed data may result in violation of data constraints, and persistence layer corruption. Transactions must be carefully applied to rollbacks of unwanted data writes in case of unexpected failures. Naturally, I kept transactional operations in cohesive modules. For example, in the TextBox application, there are two daemon services which send and receive SMS messages. Prior to enqueuing a new SMS task to the send service queue, the SMS message is persisted by the core module (web servlet). The persistence requires a database transaction to insert records into three tables: *sms_inbound*, *chat_message*, *chat*. For a complete reference to database schema used by TextBox refer to Appendix C. Once persistence layer commits the database writes, TextBox sends an HTTP request to the third party SMS Gateway. According to the proposed methodology, sending the HTTP request can be delegated to an independent

entity because this operation does not belong in the same transaction as the persistence. Sender service asynchronously receives the message with results of the persistence serialized as a JSON object. Sender service spawns a thread to execute the HTTP post request. In this model, the web servlet is not concerned with anything other than persisting the new SMS object. Decorating the persistence operation with *NotificationOperation* interface allows automatic JMS notification. Classes implementing the servlet are only left with implementing the business logic. JMS notification and third party API communication are completely abstracted away. Any resources dedicated to the servlet by the web servlet container can be released as soon as the SMS is persisted. The sender services can be executing on any network connected machine distributing processor load throughout the system rather than creating a performance bottleneck.

4 FRAMEWORK PERFORMANCE

4.1 Test Methodology

In order to determine whether the proposed framework is an acceptable research avenue we must be able to test the server push technology for a high concurrency environment. Data driven applications operate in a highly parallel, distributed environment where the number of concurrent users and data requests may grow and shrink rapidly. The proposed framework must minimize latency with increased bandwidth without significant impact to performance. Thus, I devised a stress test scenario aimed at identifying bottlenecks and breaking points of the proposed framework.

Rather than performing stress testing on all system requirements I selected a set of use cases to target specific test goals. The TextBox application, as described in earlier sections, was built on top of the previously said push architecture, however due to complex business logic encapsulated in the persistence layer, executing a successful stress test was challenging. TextBox business logic requires the communicating parties to be authenticated users. Furthermore, a customer entity must own a phone number used in interaction with the third party SMS Gateway. Though it is possible to mock the outbound SMS Gateway interface, the number of standalone browser instances as well as the hardware required to perform a stress test on a large scale was impractical. What was needed was a way to isolate certain execution paths through the system from the web service endpoint to a remote client. Appendix A contains a complete system use-case diagram outlining all actors and request execution paths. A thorough analysis of the

available use-cases yielded the “Send SMS Message” a use-case accessed by the REST API Client actor as the most suitable test path for an end to end test. This use-cases involves a web service call to the headless REST container built for the TextBox application.

Web applications consist of many components all of which can contribute to performance bottlenecks and other failures. In this thesis I am not validating performance optimization techniques for web services, my stress test methodology shall focus on strictly testing the asynchronous push notification capabilities. For the said reason, all code that is not directly a part of the notification framework, had to be minimized. The result was an idempotent web service endpoint accepting parameter-less HTTP GET requests. The servlet request generated a unique message ID sequence statically synchronized across the JVM container of the web server. The above design guarantees that every concurrently executing servlet request thread atomically increments the sequence without side effects. The generated message is stored in an object along with a millisecond time-stamp of exact time the request is processed. Business operation executor, then generates a JMS message containing the message ID and time-stamp. As a result of stress testing the new web service endpoint generates a set of JMS messages containing a sequence of message ID's. Ideally, in an environment where no network errors, timeouts or data loss occurs, the subscribing Java Applet is expected to receive an equal number of JMS messages to the number of HTTP requests sent during the test. In addition, when read, the received messages shall contain a sequence of number with no gaps or missing values. Unfortunately due to network congestion, routing, hardware processing and memory

limitations etc, the ideal environment is difficult to obtain and various types of errors and data loss are expected.

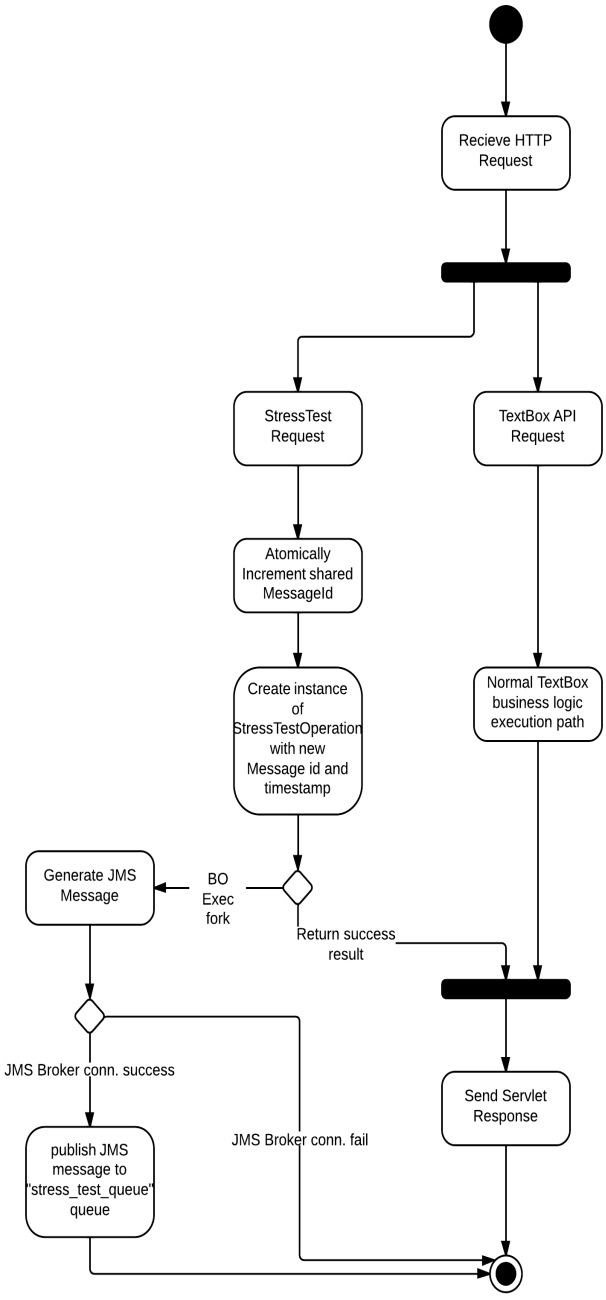


Figure 8: Stress Test Activity Diagram

4.2 Stress Test Implementation

The web service endpoint described in the previous section defines the entry point for a distributed stress test. Appendix H shows a complete listing of the servlet code and the Operation class designed to convert received HTTP requests into a JMS message sequence. This servlet was installed into the TextBox web-service container and deployed to a public domain. The idempotent nature of the stress test servlet guarantees that the received HTTP requests return successful results to the sender. In addition, the state of the system is unaffected by the stress test requests, therefore it poses no harm in integrating the stress test servlet into the TextBox application.

In order to complete the end-to-end test system, I created a modified version of the JMS Java Applet client. The original specification for the applet client contained validation of received JMS messages followed by execution of parent page JavaScript engine functions to render message contents. The test version of the applet client was stripped of all validation and JavaScript interfaces, in order to isolate the data transmission aspect under test. The test applet was redesigned to simply log the received message contents and the time-stamp of message receipt. Comparing the time-stamp long integer value from the message content and the time-stamp of message receipt on the applet a latency value measured in milliseconds was calculated and logged.

The generated log file was processed by a set of data analysis algorithms designed specifically for the devised stress test scenario. In sections 3.3 through 3.5, I further explain the algorithms used. The generated logs were used to calculate three performance metrics: data loss, latency, and overall system performance. Data loss refers to gaps in message ID sequence of received messages, which would indicate the ratio of lost data versus number of concurrent connections in the application. Latency was also measured with several metrics that focused mainly on the rate of increase compared to the rate of increase in number of concurrent connections. Ideally, the rate of increase in latency should be much slower compared to increase in concurrent requests. As predicted, the test results (explained below) show that latency increased at a significantly slower rate, indicative of a highly scalable system.

The final piece required to perform the stress test is a system of distributed nodes capable of sending a large number of concurrent HTTP requests to the newly developed stress test

servlet. Developing such a system would require a large number of hardware processing units capable of highly parallel independent network requests. Thus, a third party tool *Loader.io* was used to facilitate the task above. *Loader.io* is an open source, web based application allowing clients to stress test applications with little to no configuration complexity. The tool provides stress test capability of up to 50,000 concurrent nodes. Using *Loader.io* I conducted a set of four tests on my test architecture, by increasing the number of concurrent users with each test to gauge latency, data loss, and other error rate trends. The tests were divided into: 10, 100, 1000, and 10,000 concurrent users. For each test *Loader.io* was configured to incrementally increase the number of requests starting from 0 and ending with the test goal. Several metrics were collected by the third party tool during the execution of tests. These metrics included, number of successful vs error requests, number of network and HTTP timeout errors, and number of 400[13] and 500[14] errors. *Loader.io* presented the results in a well organized user interface which made test analysis much simpler.

4.2.1 Test Results

	Loader.io Report		Data Analysis			
Requests	Requests received	Request Errors	Latency	Data Loss	Data Gap Mode	Data Gap Avg
10	875	0	12.64118372 ms	0.00%	0	0
100	3744	0	69.26183311 ms	1.80%	1	3.35
1000	3188	65	454.7787939 ms	11.00%	1	12.73529412
10000	7192	3715	780.6827262 ms	16.50%	1	4.456953642

Table 3: Stress test results

13 HTTP 400 Errors: “Bad Request” The Web server thinks that the data stream sent by the client was 'malformed' or did not respect the HTTP protocol completely.

14 HTTP 500 Errors: “Internal Server Error”, The Web server encountered an unexpected condition that prevented it from fulfilling the request by the client for access to the requested URL.

Results of the stress test suite are presented in Table 3. The table is organized into two distinct sections: data analysis performed on applet log files and reports received from *Loader.io* stress-test tool. The report from Loader.io shows performance of the web service layer. We are less concerned with these values because the goal set for stress testing the framework was focused on testing the asynchronous messaging not the web service performance. It is interesting to mention, however, that because the request errors which begin to appear in the two highest load test cases are a more likely due to the

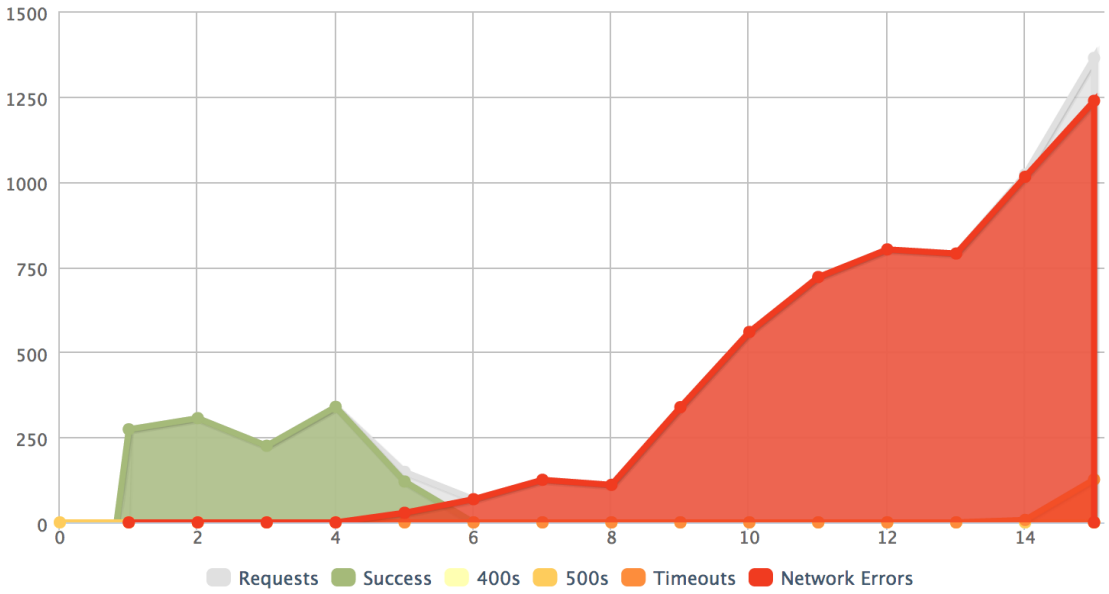


Figure 9: Loader.io 10,000 Concurrent User Test Case Error Report

testing hardware limitations. Most errors appear to be network errors as shown in Figure 9 and is an indication of a saturated data connection and not performance issues of an application. Data Loss Analysis

Data loss was analyzed using a simple algorithm executed on the log files produced by the JMS Java Applet client. Data loss is defined as messages generated by the stress test

servlet request from *Loader.io* which were not delivered to the client. The client knows how many messages it must receive because the messages are generated with a sequence of message ids synchronized across the JVM. The client is guaranteed to receive a set of messages starting with the message ID of 0 and ending with some arbitrary number. The log file produced by the client contains all received messages. A simple algorithm iterates through the log entries. Each iteration processes a single log entry into a map of message ID keys and the calculated latency values. If a value is missing from the message ID sequence it's considered a gap. The algorithm will count gaps to identify total missing data. In addition, the algorithm counts the most frequently occurring gap size (mode) and average gap size.

Looking at the data in Table 3 we see that data loss is virtually non-existent in the first two test cases. In the third case with 1,000 concurrent connections an 11% data loss was observed. With gap mode of 1 and an average gap size of 3.35 we can see that under a high load the system started to drop messages. Furthermore, under the heaviest load, with 10,000 connections we start seeing more data loss. However, the data loss ratio increased only by 5% as as load increased by 1000%. In addition data gap mode remains at 1, which is an acceptable figure. Overall, we can see that data loss does occur under the heaviest load but remains manageable when compared to the number of received requests.

4.3 Latency Analysis

Latency is the most important metric which was measured by the stress test because latency is a direct indication of the perceived web application responsiveness. The testing model used in this thesis focuses on measuring the rate of increase in latency compared to the rate of increase in the number of concurrently connected users.

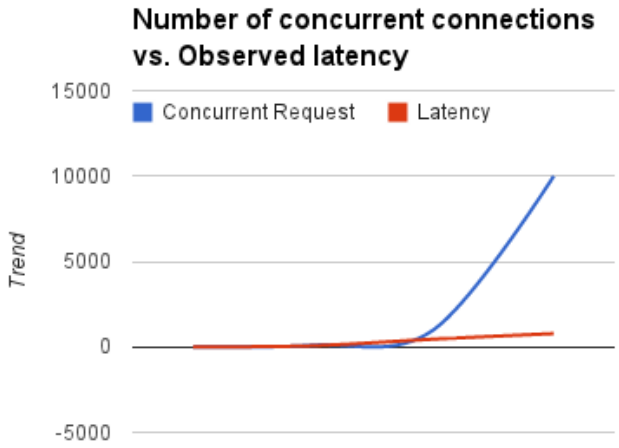


Figure 10: Stress Test Latency Analysis, number of connections vs latency increase rate

The *Loader.io* test suite started the number of concurrent requests at zero and increased until the test goal was reached over the configured test period. As described in the previous section, latency was calculated by comparing the time-stamp value of when the message was generated to the time-stamp of when the client received the message. For each test case, 10, 100, 1000, and 10,000 concurrent user, I measured the average latency of JMS message delivery for successful requests. Figure 10, shows two curves: the average latency in red and the number of concurrent requests in blue. The results are indicative of a system well suited for large number of users. It is evident that the latency curve increases at a slower rate than concurrent connections curve which was expected. If the two curves exhibited similar growth then the system would not be able to scale well under heavy load. In Table 3, we can see that for the 10 concurrent connections test case,

average latency was roughly 12 milliseconds and for the case of 10,000 concurrent connections, average latency was 780 milliseconds; an increase of about 6.5e100%. When compared to the gain of concurrent users, calculated at 1.0e100000%, the increase in latency is an acceptable figure.

It is impossible to avoid latency increase in a highly concurrent system because more CPU cycles are spent processing servlet request threads as more threads are spawned. However, it's important to note that the presented test results in this thesis are slightly skewed. TextBox application involved in the stress test was deployed on a virtualized hardware platform. Three virtual machines, running inside a single physical server deployed the database, web server, and the JMS broker messaging entity. Thus, increase in the required processing resources on one virtual machine inherently causes the same resources load increase for the other two virtual machines. The physical server on which the TextBox is deployed, was configured with four physical CPU's and eight hyper-threaded[15] virtual cores. The host operating system does not assign dedicated CPU cores to each virtual machine instance so computing resources are pooled throughout the entire system. As a result heavy load affects all applications evenly. A better approach for a stress test would require using a cloud computing platform or a set of separate physical servers.

15 Hyper Threading – Intel's proprietary multi-threading implementation at the processor level. Threads are executed on the same processing core by alternating processing instructions, instead of completing on thread's task then starting another. This technique improves multi-threading performance of some applications.

5 CONCLUSION

Research presented in this thesis denotes several aspects of web application development prominent in asynchronous data delivery. Pushing data across network resources at server's discretion rather than request from clients proves to be more efficient. Implementing data push technology is not enough to streamline development practices. A framework that abstracts away communication details and allows dedication of development resources to application logic proves equally as important. The proposed framework allows software architects to focus on designing responsive applications based on modularization principles. Trust in framework performance allows designers to focus on charting out complex distributed systems to deliver much touted data throughout network resources. Additional work is still needed to drill down more robust libraries and to abstract complexity, yet the groundwork has been laid out. Based on the proposed framework application developers can build responsive asynchronous web applications. The trend towards data centered applications is driving more research into data mining algorithms and delivery of data should not fall behind. Incredible leaps in the ability to process seeming insurmountable amounts of data require robust frameworks able to deliver data to clients. In this thesis I presented that it is possible to build an application consisting of several distributed modules to deliver a cohesive result. Using the power of publish-subscribe architecture and delegation of concerns I was able to develop a robust SMS client application taking full advantage of an open source messaging framework.

Further research in this field will yield web applications that were impossible to conceive just a few years go. More frameworks similar to the one proposed in this thesis are going

to feed the developers' hunger for solutions to help build distributed systems. It is an interesting time to be a part of an exciting industry developing around the world wide web today.

6 FUTURE RESEARCH

Several topics discussed still require future research. Adoption of WebSockets protocol into the system would be an essential future research task. Most JMS brokers, including ActiveMQ, support the WebSockets protocol as a means of delivering messages. Instead of embedding a Java applet client, which subscribes to JMS queues, it would be possible to implement asynchronous notification through WebSockets. As mentioned, WebSocket protocol is not supported by legacy web browser clients. To get around this issue, it would be possible to integrate the current Java Applet solution with the new WebSockets implementation. The client side JavaScript engine will decide whether or not the more robust WebSockets protocol is supported and will load the notification mechanisms accordingly.

Additionally on the server side, the proposed framework could be implemented with other popular web application languages such as: .NET framework, PHP, or Node.js. Although Java provides a simple integration with JMS brokers, other platforms can access the JMS broker taking advantage of JMS protocol's generic specification. Using platform specific extensions many JMS brokers allow integration with non-java applications. Modular design of the framework allows systems which are already built lacking the asynchronous functionality to be easily un-synchronized.

Automatic load balancing built into the framework is another topic which could be addressed in the future. The current design allows for load balancing on several levels including: load balancing the JMS brokers, daemon services, web servlet containers.

However, the proposed framework does not contain a way to automate load balancing on the need basis. For Java implementation, a JNDI integrated directory service could be plugged into the framework to delegate assignment of requested resources throughout the modules. Load manager objects can monitor the system performance and spin up new instances of modules. Load manager would then reconfigure the JNDI provider to assign resources accordingly. A system which can monitor its own performance and increase or reduce the amount of required computing power would be perfect for a cloud based hosting environment. Well implemented automatic load balancing could help reduce downtime and increase responsiveness of the web application.

In addition to load balancing, a JMS based built-in-test (BIT) library was originally devised to be a part of the proposed framework design. BIT is a library which allows system modules to communicate with each other as well as send statistics and health information throughout the life of the application. Highly modularized applications especially in a distributed architecture must ensure all modules are operating as expected. The system should be designed with fault tolerance as a main goal eliminating single points of failure. BIT modules would allow the system to administer itself, monitor modules for crashes, memory leaks, etc. Each module implements interface provided by the BIT library and monitor's its own health. Upon discovering faults, modules use JMS to notify the BIT controller which contains complex logic for dealing with faults. The controller can determine how to proceed with the reported faults and may request the faulted module to perform certain self-correcting procedures. Using the BIT module to control the distributed web application server system, could reduce downtown when

crashes and other faults occur. This module was left for future research to keep the scope of the project manageable. However, without a similar approach to fault tolerance a highly modular system is not practical for a production ready software application.

REFERENCES

[1] Gamma, E. (1995). *Design patterns, elements of reusable object-oriented software*. Addison-Wesley Professional.

[2] Patrick Eugster. , Pascal A Felber, , Rachid Guerraoui, , & Anne-Marie Kermarrec, (2003). The many faces of publish subscribe. *ACM Computing Surveys*,*35*(2), 114-131.

[3] B. Carpenter, "Hypertext Transfer Protocol -- HTTP/1.1" RFC 2616, June 1999. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html

[4] McCarthy, P., & Dave, C. *Comet and reverse ajax: The next-generation ajax 2.0*. First Press.

[5] Bozdag, E., Masbah, A., & van Deursen, A. (2007, October).*A comparison of push and pull techniques for ajax*. 9th IEEE internation workshop.

[6] Russell, A., Wilkins, G., Davis, D., & Nesbitt, M. (2007). *The bayeux specification*. Retrieved from http://svn.cometd.com/trunk/bayeux/bayeux.html. Retrieve date May 2013.

[7] Hapner, M., Burridge, R., & Sharma, R. (1999, November 9). *Java™ message service*. Retrieved from http://docs.oracle.com/cd/E19957-01/816-5904-10/816-5904-10.pdf. Retrieve date April 2013.

[8] Sachs, K., Kounev, S., & Appel, S. (2009).*Benchmarking of message-oriented middleware*. Third acm international conference on distributed event-based systems, New York, NY, USA. doi: 10.1145/1619258.1619313

[9] *All spec specjms2007@horizontal results published by spec*. (2011, June 30). Retrieved from http://www.spec.org/jms2007/results/jms2007horizontal.html. Retrieve date April 2013.

[10] Han, J., Kamber, M., Pei, J., Pei, F., & et al, F. (2012).*Data mining, concepts and techniques*. (3rd ed.). Waltham, Ma: Morgan Kaufmann.

[11] Xmlhttprequest. In (2012). J. Aubourg, J. Song & H. Steen (Eds.), *W3C Working Draft*. W3C. Retrieved from http://www.w3.org/TR/XMLHttpRequest. Retrieve date March 2013.

[12] Lubbers, P., & Greco, F. (n.d.). *Html5 web sockets: A quantum leap in scalability for the web*. Retrieved from http://www.websocket.org/quantum.html. Retrieve date December 2012.

[13] Melnikov, A. (2011). The websocket protocol. In *Request for Comments: 6455*. Retrieved from http://tools.ietf.org/html/rfc6455. Retrieve date March 2013.

[14] Lerner, R. (n.d.). At the forge: real-time messaging. (2013).*Linux Journal*, *2013*(225), doi: ISSN: 1075-3583

[15] Ruiping , X. (2008). *The economic interests and legal issues of oss*. In *2008 International Conference on Wireless Communications, Networking and Mobile Computing* (pp. 1-4). doi: 10.1109/WiCom.2008.2072

[16] Ven, K. (n.d.). Should you adopt open source software?. (2008). *Software, IEEE*, *25*(3), 54-59. doi: 10.1109/MS.2008.73

[17] Corbett, J., Dean, J., & Epstein, M. (2012, October).*Spanner: Google's globally-distributed database*. Osdi, Hollywood, CA.

[18] Gamma, R., Helm, R., Johnson, R., & Vlissides, J. (2009). *Design patterns: Elements of reusable objec-oriented software*. (37 ed.). Westford Massachusetts: Pearson Education.

[19] Dynamic class loading. In (2013). *Java Remote Method Invocation 3: System Overview*. Oracle Inc. Retrieved from http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmi-arch5.html. Retrieve date May 2013.

[20] Winer, D. (1999). Xml-rpc specification. In *XML-RPC*. Retrieved from http://xmlrpc.scripting.com/spec.html. Retrieve date February 2012.

[21] Rmi transport protocol. In (2013). *Java Remote Method Invocation: System Overview*. Oracle Inc. Retrieved from

[22] *Top 5 browsers from july 2008 to february 2013*. (2013, April 09). Retrieved from http://gs.statcounter.com. Retrieve date May 2013.

[23] Ecmascript language specification. In (2011). *Standard ECMA-262* (5.1 ed.). ECMA International. Retrieved from http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. Retrieve date March 2013.

[24] Stonebraker, M. (2010, April). Sql databases v. nosql databases. *Communications of the ACM*, *5*(4)

[25] Annotations. In (2013). *Java Standard Edition 1.5.0 Documentation*. Oracle Inc. Retrieved from http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

Retrieve date April 2013.

[26] Mili, H., Mcheick, H., Elkharraz, A., Lounis, H., & Sahraoui, H. (2006). *Concerned about separation*. In *FASE'06 Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering* (pp. 247-261). doi: 10.1007/11693017_19

[27] Lucian, P., Ghodsi, A., & Stoica, I. (2010). *Http as the narrow waist of the future internet*. In *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (p. 6). doi: 10.1145/1868447.1868453

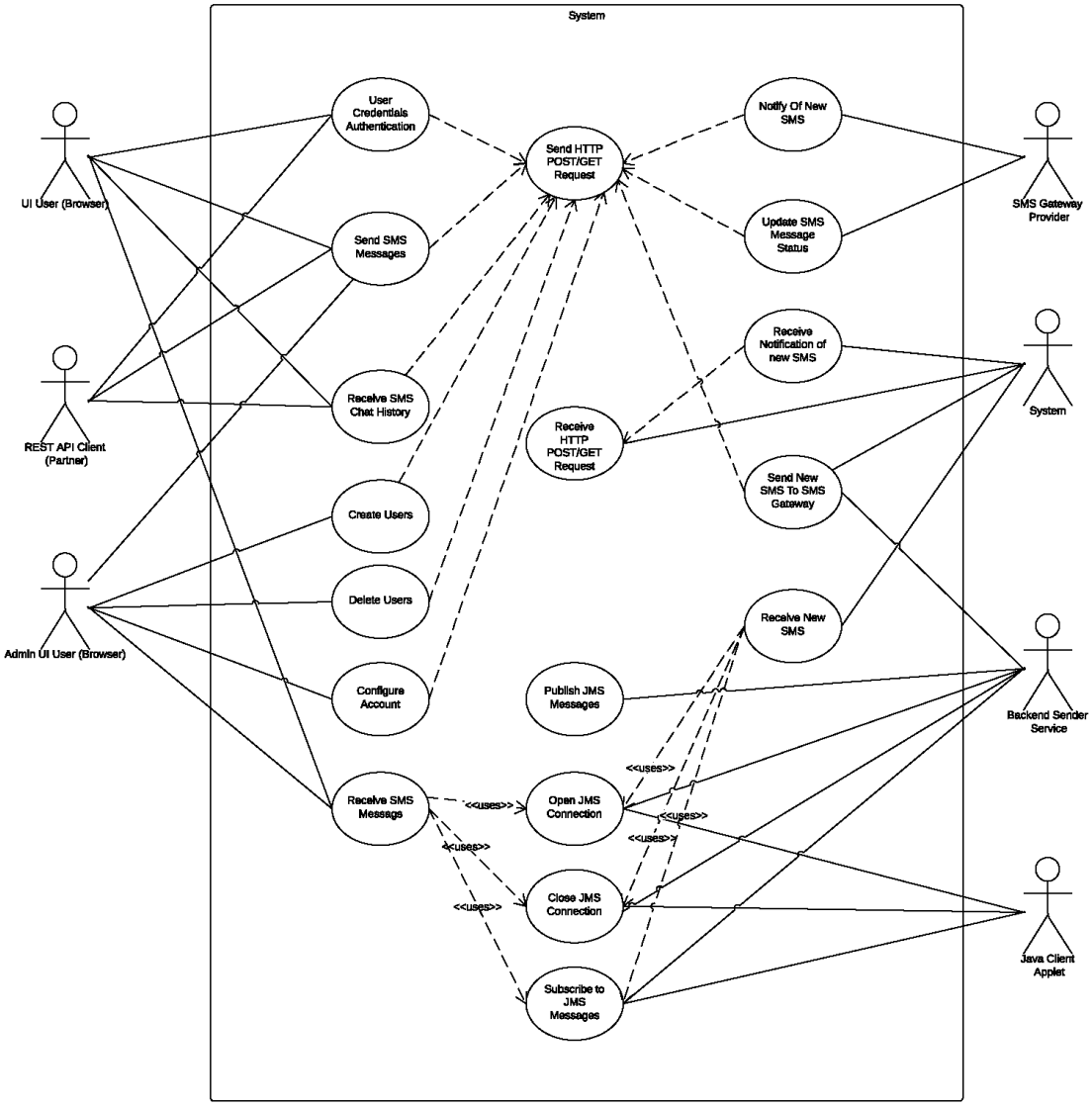
[28] Fraternali, P. & Paolini, P. (2000, October). Model-driven development of web applications: the autoweb system. *ACM Transactions on Information Systems (TOIS)*, *18*(4), 323-382

[29] Roberts, L. G. (2000, Janurary). Beyond moores law intenet growth trends. *IEEE Computer Society: Computer*,*33*(1), 117- 119

[30] Loreto S., Saint-Andre P., Salsano S., (2011). Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP In *Request for Comments: 6202*. Retrieved from http://tools.ietf.org/html/rfc6202. Retrieve date December 2012.

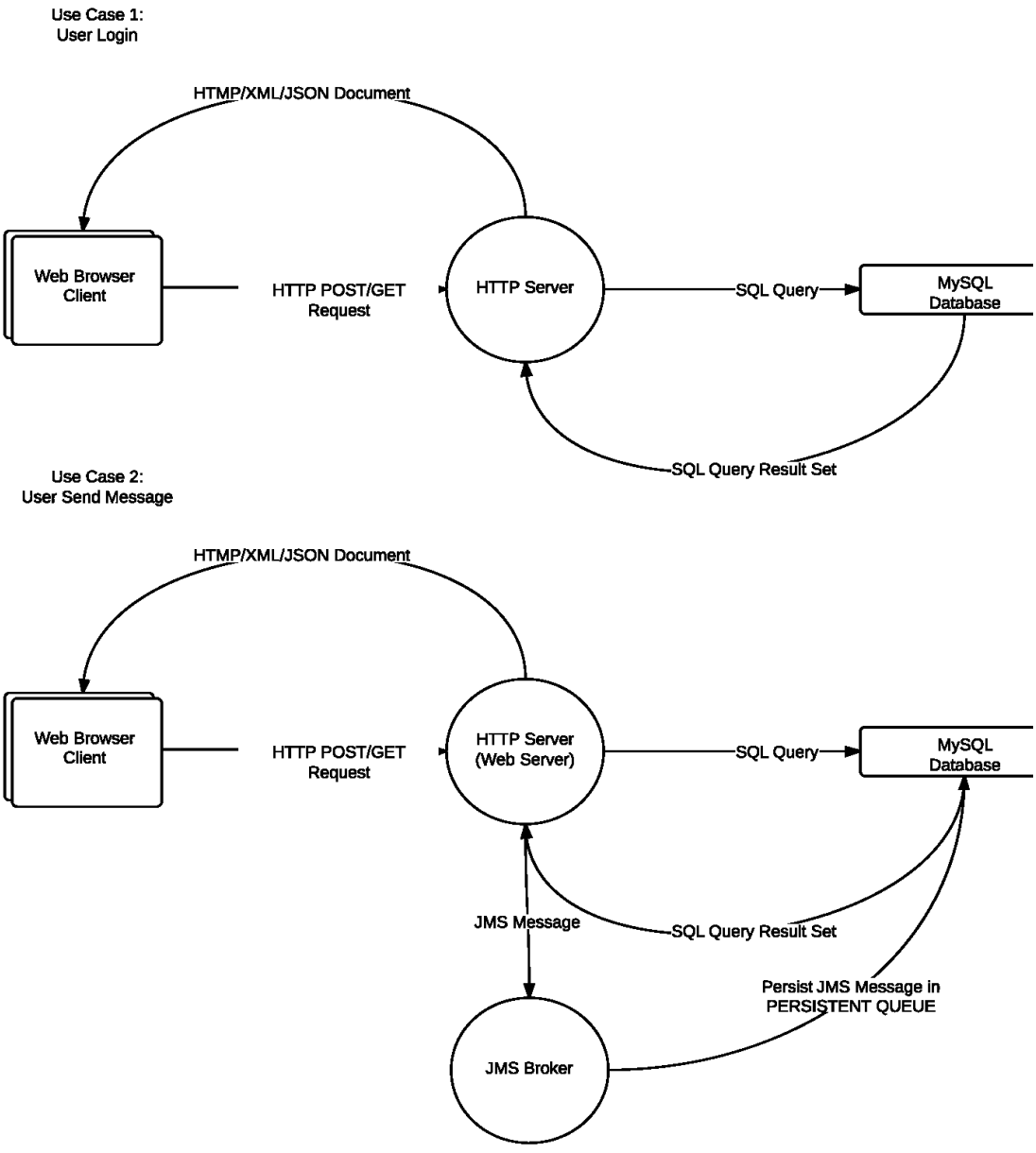
APPENDIX A

System Use Case Diagram:

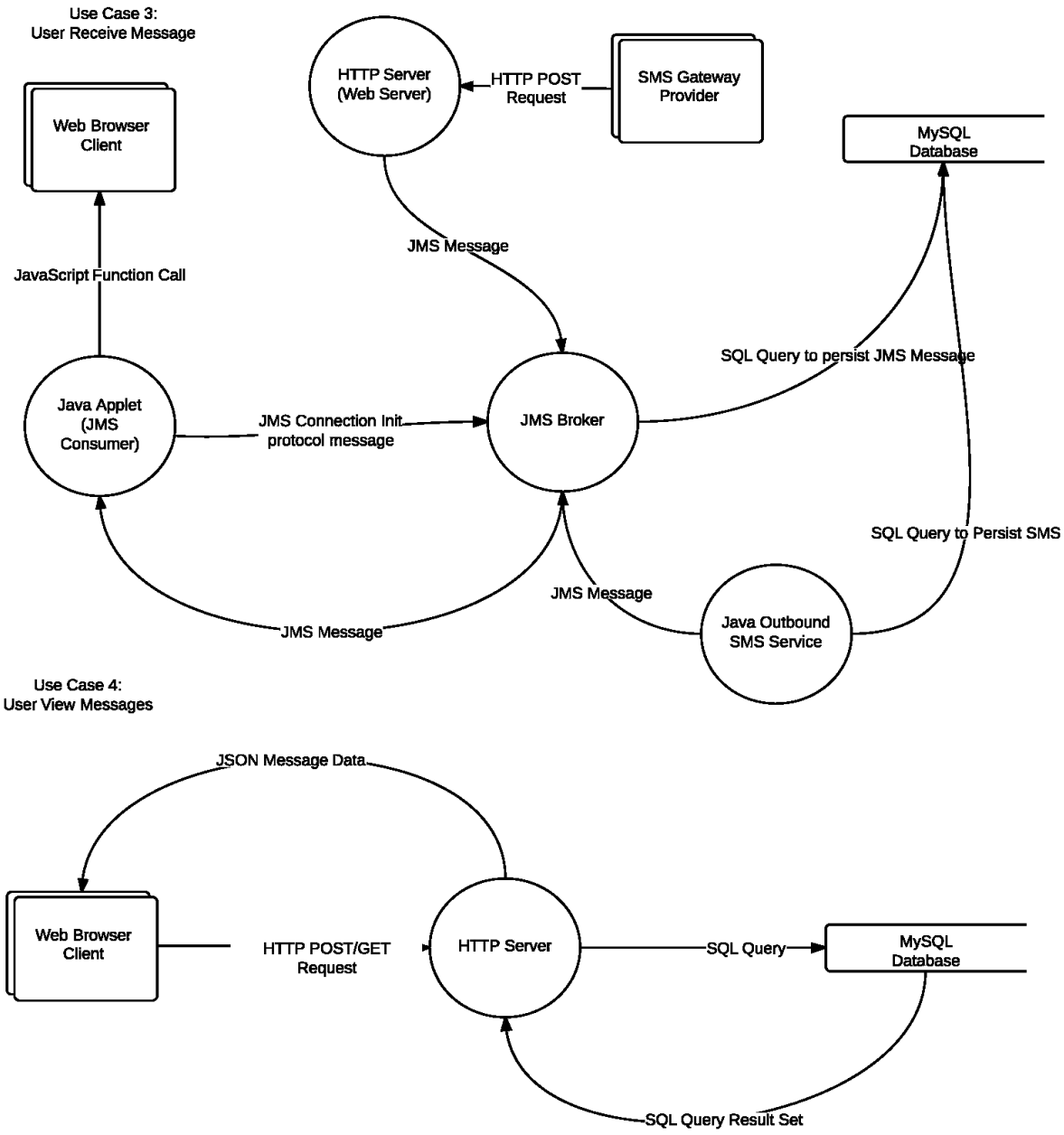


APPENDIX B

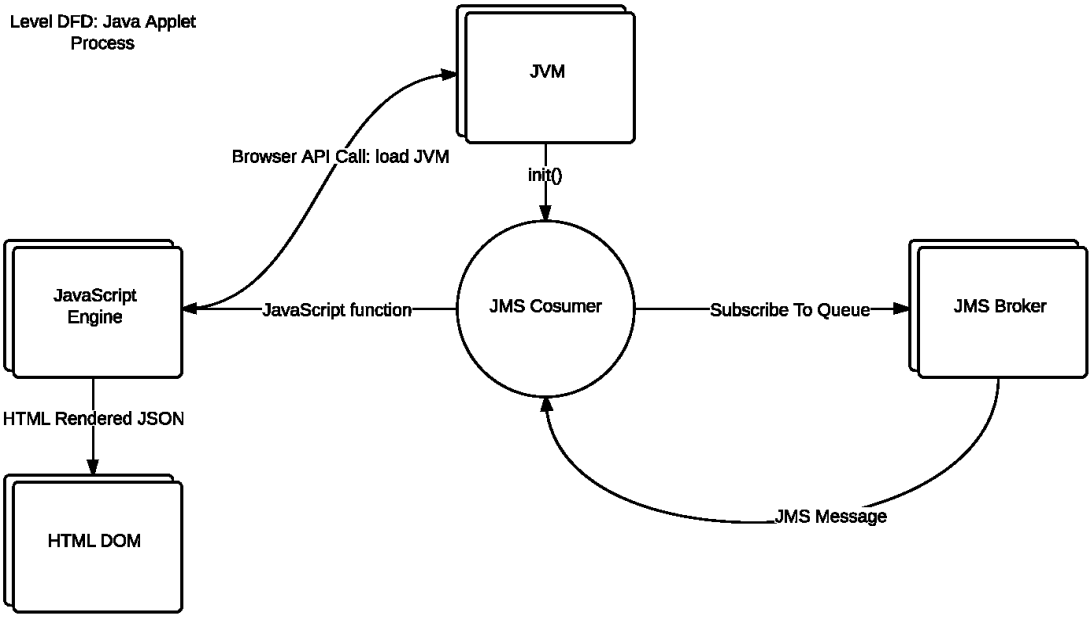
Data Flow Diagram: Use Case 1



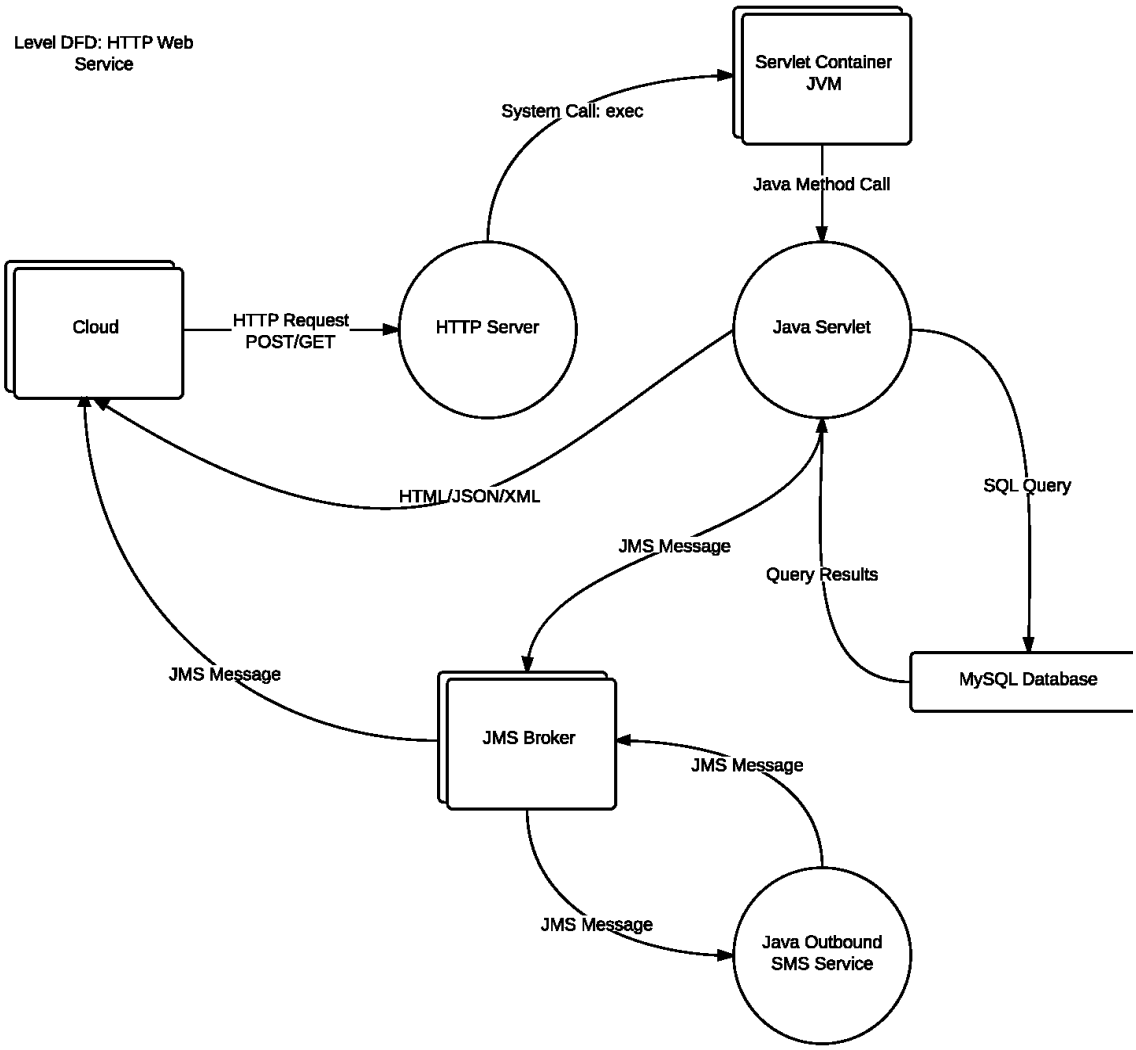
Data Flow Diagram: Use Case 2 and Use Case 4



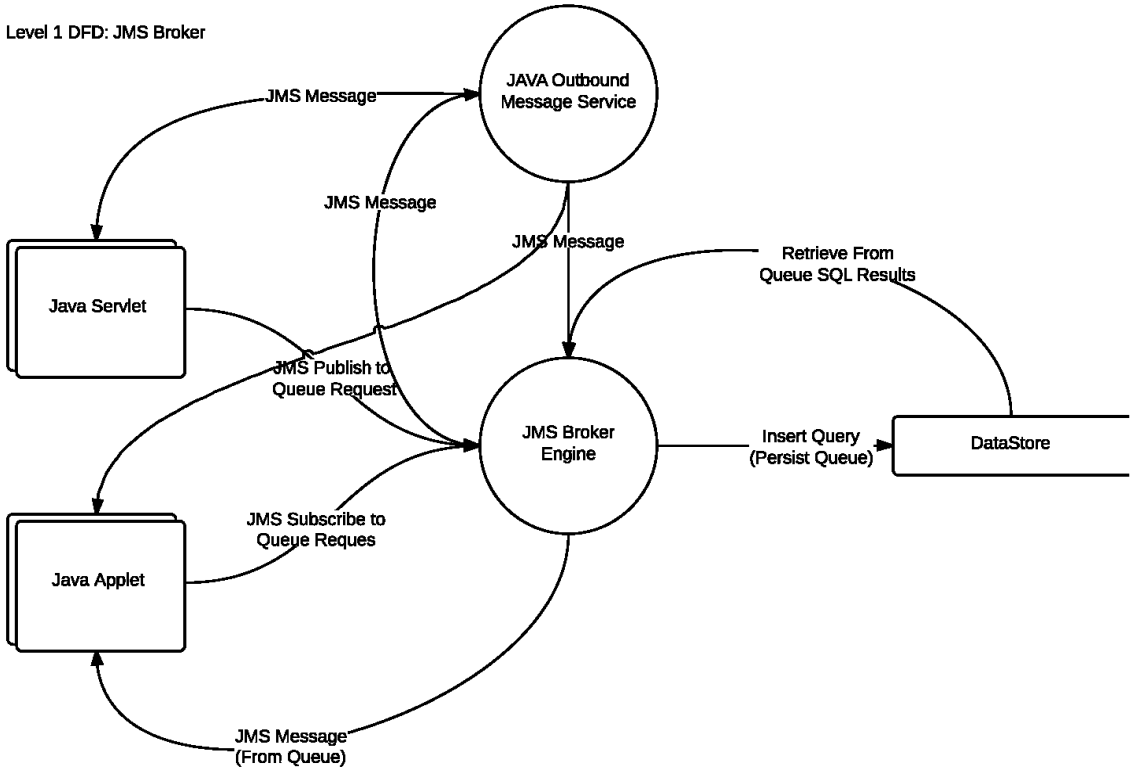
Data Flow Diagram: Java Applet



Data Flow Diagram: HTTP Web Server

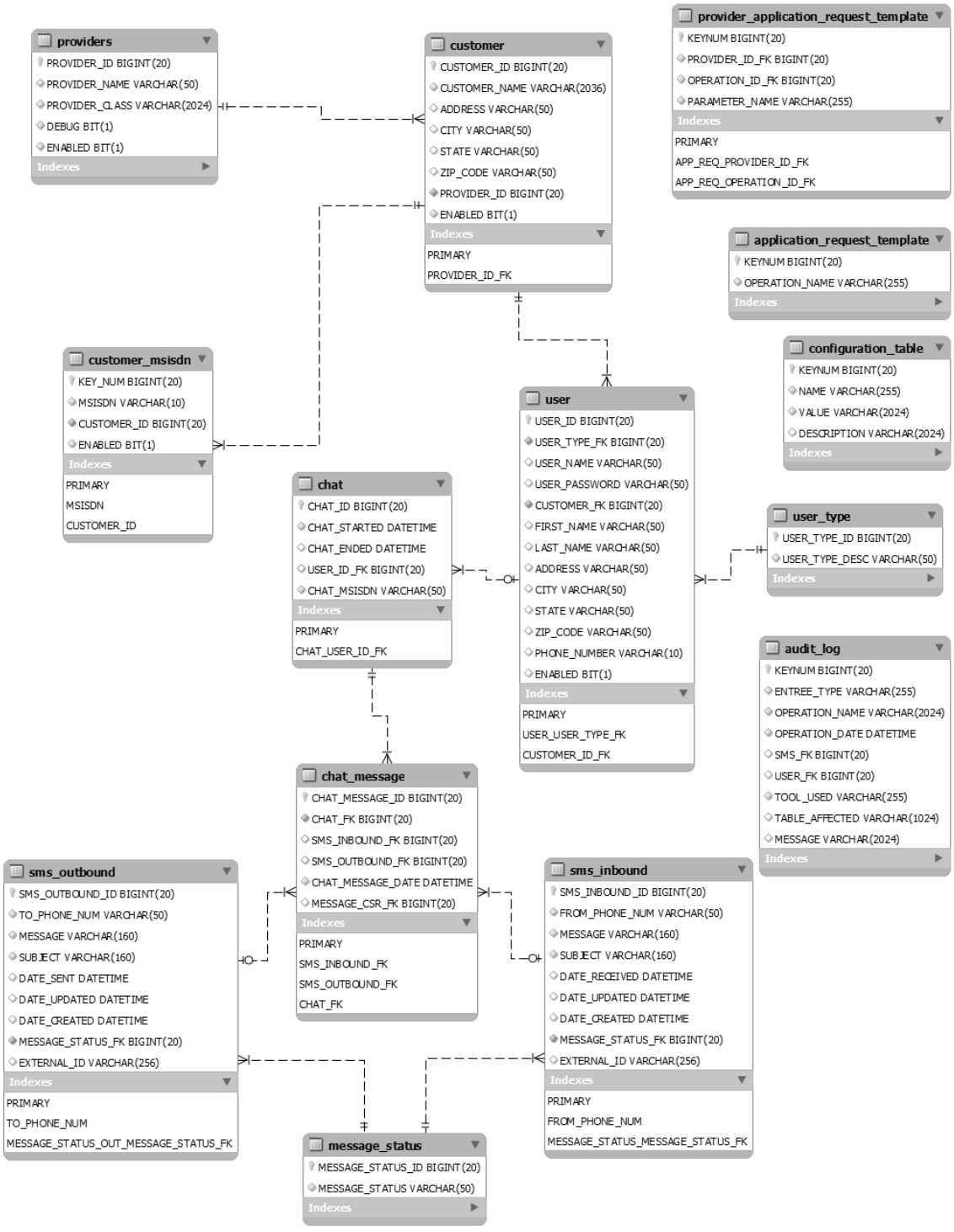


Data Flow Diagram: JMS Broker



APPENDIX C

Database Schema:



APPENDIX D

JmsClient Java Class Source Code

```
package com.smsuite.client.transport;
import java.applet.Applet;
import java.awt.HeadlessException;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.net.InetAddress;
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import netscape.javascript.JSObject;
import org.apache.activemq.ActiveMQConnectionFactory;
/**
 * Implementation of a JMS client Applet which establishes a JMS connection with
 * a broker over TCP/IP network and subscribes to a queue/topic to recieve
 * messages asynchronously pushed from the server.
 * <p>
 * This client communicates with the JavaScript engine of the window which
 * embeds the applet. The JavaScript engine is used to obtain JMS server
 * configuration as well as to display error notifications.
 * <p>
 * The JavaScript engine is required to implmenet a set of functions used by
 * this client in order for this applet to work properly.
 * <p>
 * Additionally the JavaScript engine is notified of progress of this client
 * through a set of events which get fired by the client.
 *
 * @author David Khanaferov
 *
 */
public class JmsClient extends Applet {
          private static final long serialVersionUID = -145699297543303768 6L;
          private Connection connection;
          private ActiveMQConnectionFactory connectionFactory;
          private Session session;
          private MessageConsumer consumer;
          public JmsClient() throws HeadlessException {
                    super();
          }
          @Override
          public void init() {
                    JSObject window = JSObject.getWindow(this);
                    window.eval("onJmsClientInitialized();");
          }
          @Override
          public void destroy() {
                    try {
                              this.consumer.close();
                              this.session.close();
                              this.connection.close();
                    } catch (Exception ex) {
```

```
                        errorNotification(ex);
                }
        }
        /**
         * Can be used by the javascript client to reset current connection Must be
         * followed by the initJMSConnection() method to start new connection
         */
        public void resetConnection() {
                try {
                        this.consumer.close();
                        this.session.close();
                        this.connection.close();
                } catch (Exception ex) {
                        errorNotification(ex);
                }
        }
        @Override
        public void start() {
                JSObject window = JSObject.getWindow(this);
                window.eval("onJmsClientStarting();");
                if (connection != null) {
                        resetConnection();
                }

                System.out.println("Loading JMS Server settings...");

                Double userId = (Double) window.eval("getJmsUserId()");
                String host = (String) window.eval("getJmsServerHost()");
                Double port = (Double) window.eval("getJmsServerPort()");

                System.out.println("Starting JMS Client...");

                initJMSConnection("JMS_CLIENT_APPLET_[" + new Double(userId).intValue()
                                + "]" + System.currentTimeMillis(),
"USER_NOTIFICATION_"
                                + new Double(userId).intValue(), host,
port.intValue());
                window.eval("onJmsClientStarted();");
        }
        /**
         * Creates a new JMS Connection
         *
         * @param clientName
         * @param queue
         */
        public void initJMSConnection(String clientName, String queue, String host,
                        Integer port) {
                JSObject window = JSObject.getWindow(this);
                String user = "admin";
                String password = "activemq";

                try {
                        System.out.println("GOT HOST FROM JS = "+host);
                        System.out.println("Connecting to:
"+InetAddress.getByName(host).getHostAddress());
```

```
                                connectionFactory = new ActiveMQConnectionFactory(user,
password,
                                                "tcp://" +
InetAddress.getByName(host).getHostAddress()
                                                                                + ":" + port);


                                window.eval("onJmsClientCreatingConnection();");

                                connection = connectionFactory
                                                .createQueueConnection(user, password);
                                this.connection.setClientID(clientName + "_"
                                                + System.currentTimeMillis());
                                connection.start();
                                window.eval("onJmsClientConnectionStarted();");
                                // Create a Session
                                session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
                                window.eval("onJmsClientSessionCreated();");
                                // Create the destination (Topic)
                                Destination destination = session.createQueue(queue);
                                if (destination == null) {
                                          throw new JMSException("failed to create JMS
destination for: "
                                                                + queue);
                                }
                                window.eval("onJmsClientDestinationCreated();");
                                // Create a MessageConsumer from the Session to the Topic or
Queue
                                consumer = session.createConsumer(destination);
                                if (consumer == null) {
                                          throw new JMSException(
                                                                "failed to create a consumer
subscription for topic: "
                                                                                + queue);
                                }
                                window.eval("onJmsClientConsumerCreated();");
                                // Wait for a message
                                consumer.setMessageListener(new AppletMessageListener(this));
                                window.eval("onJmsClientReady();");
                      } catch (Exception e) {
                                errorNotification(e);
                      }
            }
            /**
             * This method is fired when a JMS error occurs
             *
             * @param t
             */
            public void errorNotification(Throwable t) {
                      StringWriter sw = new StringWriter();
                      PrintWriter pw = new PrintWriter(sw);
                      t.printStackTrace(pw);
                      System.out.println(sw.toString());
                      JSObject window = JSObject.getWindow(this);
```

```
            window.eval("onJmsError('" + t.getMessage() + "');");
    }
    /**
     * This method fires when a JMS Message is received by the client
     *
     * @param msg
     *             {@link String} JSON encoded message
     */
    public void clientNotification(String msg) {
            JSObject window = JSObject.getWindow(this);
            window.eval("onJmsMessage('" + msg + "');");
```

APPENDIX E

JMSProviderFactory Java Class Source Code

```
package com.csm.backend.messaging.jms;
import javax.jms.JMSException;
import com.csm.backend.Result;
import com.csm.backend.ResultCode;
import com.csm.backend.exception.NotificationException;
import com.csm.backend.messaging.jms.providers.ActiveMQJmsProvider;
/**
 *
 * @author David Khanaferov
 *
 */
public class JMSProviderFactory {
          private static ThreadLocal<JmsProvider> localJmsProvider = new
ThreadLocal<JmsProvider>();
          private static JmsProvider buildProvider() throws JMSException {
                    JmsProvider provider = new ActiveMQJmsProvider();
                    localJmsProvider.set(provider);
                    return provider;
          }
          private static JmsProvider buildProvider(JmsProviderBuilder builder)
                              throws JMSException, NotificationException {
                    JmsProvider provider = new ActiveMQJmsProvider(builder);
                    localJmsProvider.set(provider);
                    return provider;
          }
          public static JmsProvider getProvider(JmsProviderBuilder builder)
                              throws NotificationException, JMSException {
                    JmsProvider provider = localJmsProvider.get();
                    try {
                              if (provider == null) {
                                        provider = buildProvider(builder);
                              } else {
                                        throw new NotificationException(

ResultCode.OPERATION_NOT_ALLOWED,
                                                                  "Local JMS Provider already
configured, can not configure a local JMS provider more than once.");
                              }
                    } catch (Result ex) {
                              if (ex.getResult().equals(ResultCode.OPERATION_NOT_ALLOWED))
{
                                        // TODO: Add logging
                                        System.out
                                                                  .println("WARNING: "
                                                                                      +
ActiveMQJmsProvider.class
                                                                                      + " is
already configured, returning previously configured instance");
                                        provider = getProvider();
                              }
                    }
                    return provider;
```

```
        }
        public static JmsProvider getProvider() throws JMSException {
                JmsProvider provider = localJmsProvider.get();
                if (provider == null) {
                        provider = buildProvider();
                }
                if (!provider.isConnected()) {
                        provider = buildProvider();
                }
                return provider;
        }

        public static void shutdownProvider() throws JMSException {
                JmsProvider provider = localJmsProvider.get();
                if (provider != null) {
                        provider.closeConnection();
                        localJmsProvider.remove();
                }
        }
}
```

APPENDIX F

BusinessLogicExecutor Java Class Source Code

```
package com.smsuite.backend.logic;
import com.csm.backend.Result;
import com.csm.backend.ResultCode;
import com.csm.backend.logic.operation.AuditOperation;
import com.csm.backend.logic.operation.MaintenanceOperation;
import com.csm.backend.logic.operation.NotificationOperation;
import com.csm.backend.logic.operation.Operation;
import com.csm.backend.logic.operation.SmsOperation;
import com.csm.backend.messaging.ProducerMessagingService;
import com.csm.backend.messaging.jms.JMSProducerServiceImpl;
import com.smsuite.backend.logic.exception.BusinessOperationException;
/**
 * This class serves as an executor service for operations. Performs checks to
 * identify which type of operation being executed is and determines the actions
 * which need to be taken. This class is final and can not be instantiated. Must
 * only be used in as a static utility class to execute operations.
 *
 * <p>
 * Note:
 * <p>
 * This class shall be the entry point to business logic layer and shall be the
 * only method of executing business operations.
 *
 * @author David Khanaferov
 *
 */
public final class BusinessOperationExecutor {
          /**
           * This class shall never be instantiated.
           */
          private BusinessOperationExecutor() {
                    throw new AssertionError("operation not allowed");
          }
          /**
           * Executes all implemented and required actions for the Operation according
           * to the operation type
           *
           * @param op
           *            {@link Operation} instance which will be executed
           * @return {@link Result} instance containing result code, error messages of
           *         the executed operation
           */
          public static Result executeOperation(Operation op)
                              throws BusinessOperationException {
                    Result operationResult = null;
                    try {
                              if (op instanceof AuditOperation) {
                                        ((AuditOperation) op).logBegin();
                              }
                              /*
                               * If operation is a MaintenanceOperation and the system is
                               * currently in maintenance mode call the isInMaintenance()
```

```
method
                                * of this operation which will allow the operation's logic
to decide
                                * how to handle the current system state
                                */
                               if (op instanceof MaintenanceOperation) {
                                         if(isInMaintenance()){
                                                   ((MaintenanceOperation)
op).isInMaintenance();
                                         }
                                         if(isUpcommmingMaintenance()){
                                                   ((MaintenanceOperation)
op).isUpcomingMaintenance();
                                         }
                               }
                               /*
                                * If operation is an SMS operation it will require a lock
on the
                                * SMS object so a lock must be acquired before proceeding
                                */
                               if (op instanceof SmsOperation) {
                                         if (!((SmsOperation) op).getLock()) {
                                                   return new Result(ResultCode.FAILED,
                                                             "could not obtain
lock on sms");
                                         }
                               }
                               /*
                                * Only proceed to execute the operation if the lock was
successfully acquired
                                */
                               try {
                                         operationResult = op.execute();

                                         if(operationResult.isSuccess()){
                                                   if (op instanceof NotificationOperation)
{
                                                             ProducerMessagingService
jmsService = new JMSProducerServiceImpl();

jmsService.sendNotification((NotificationOperation) op);

jmsService.shutdownMessagingService();
                                                   }
                                         }
                               } catch (Exception ex) {
                                         if (op instanceof AuditOperation) {
                                                   ((AuditOperation) op).logFailed();
                                         }
                                         if (operationResult == null) {
                                                   operationResult = new
Result(ResultCode.FAILED,
                                                             ex.getMessage(),
ex);
                                         }
                               } finally {
                                         if (operationResult == null) {
```

```
                                                operationResult = new
Result(ResultCode.FAILED, "unknown");
                                        }
                                        /*
                                         * If operation is an SMS operation it will
require a lock to be released
                                         */
                                        if (op instanceof SmsOperation) {
                                                if (!((SmsOperation) op).releaseLock())
{
                                                        return new
Result(ResultCode.FAILED,
                                                                        "could not
release lock on sms");
                                                }
                                        }
                                }

                                if (op instanceof AuditOperation) {
                                        ((AuditOperation) op).logSuccess();
                                }
                        } catch (Exception ex) {
                                if (op instanceof AuditOperation) {
                                        ((AuditOperation) op).logFailed();
                                }

                                throw new BusinessOperationException(
                                        "Operation Executor failed to execute
{ " + op.getName()
                                                        + "}: " +
ex.getMessage(), ex);
                        }
                        return operationResult;
                }

                //TODO: implement
                private static boolean isInMaintenance(){
                        return false;
                }

                //TODO: implement
                private static boolean isUpcommmingMaintenance(){
                        return false;
                }
}
```

APPENDIX G

ProducerMessagingService Java Interface Source Code

```
package com.csm.backend.messaging;
import javax.jms.JMSException;
import com.csm.backend.Result;
import com.csm.backend.exception.NotificationException;
import com.csm.backend.logic.operation.NotificationOperation;
public interface ProducerMessagingService {

          public Result sendNotification(NotificationOperation operation)
                              throws NotificationException;

          public boolean shutdownMessagingService() throws JMSException;
}
```

JMSProducerServiceImpl Java Class Source Code

```
package com.csm.backend.messaging.jms;
import javax.jms.JMSException;
import com.csm.backend.Result;
import com.csm.backend.ResultCode;
import com.csm.backend.exception.NotificationException;
import com.csm.backend.logic.operation.NotificationOperation;
import com.csm.backend.messaging.MessagingParserService;
import com.csm.backend.messaging.ProducerMessagingService;
import com.csm.backend.services.ServiceContext;
/**
 *
 * @author David Khanaferov
 *
 */
public class JMSProducerServiceImpl implements ProducerMessagingService{

          static {
                    JmsProviderBuilder builder = new JmsProviderBuilder()//
                                        .connectionType(JMS_MESSAGING_TYPE.QUEUE)//

.setClientId(ServiceContext.instance().getServiceName())//
                                        .setTraced(true)//
                                        .isDurable(true);
                    try {
                              JMSProviderFactory.getProvider(builder);
                    } catch (Exception e) {
                              // TODO: add logging
                              e.printStackTrace();
                              System.out
                                                  .println("Failed to build JMSProvider: "
+ e.getMessage());
                    }
          }

          public JMSProducerServiceImpl() throws JMSException, NotificationException{


                    if(     !JMSProviderFactory.getProvider().isConnected()){
```

```
                        JMSProviderFactory.getProvider().openConnection();
                }
        }

        @Override
        public Result sendNotification(NotificationOperation operation)
                        throws NotificationException {
                if (operation == null) {
                        throw new NullPointerException("operation");
                }

                try {
                        boolean sentStatus = JMSProviderFactory.getProvider()
                                        .sendMessageToQueue(

MessagingParserService.getProducerTopic(operation).getTopic(),

MessagingParserService.getDeliveryMode(operation),

MessagingParserService.getMessageContent(operation),

MessagingParserService.getEncodingType(operation));
                        if (sentStatus) {
                                return new Result(ResultCode.SUCCESS,
                                                "sendNotification() SUCCESS");
                        } else {
                                return new Result(ResultCode.FAILED,
                                                "sendNotification() Failed:
JMS provider failed to send message: "
                                                                                +
MessagingParserService.getMessageContent(operation));
                        }
                } catch (Exception ex) {
                        return new NotificationException("Failed to send
notification: "
                                        + ex.getMessage(), ex);
                }
        }
        @Override
        public boolean shutdownMessagingService() throws JMSException {
                JMSProviderFactory.getProvider().closeConnection();
                JMSProviderFactory.shutdownProvider();
                return true;
        }
}
```

APPENDIX H

StressTestRequest Java Servlet Class Source Code

```
package com.smsuite.sms.api;

import javax.servlet.http.HttpServletRequest;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

import com.csm.backend.Result;
import com.csm.backend.ResultCode;
import com.csm.backend.exception.EncoderException;
import com.csm.backend.logic.operation.Operation;
import com.csm.backend.messaging.jms.JMSProviderFactory;
import com.csm.sms.services.result.WebServiceResult;
import com.smsuite.backend.logic.BusinessOperationExecutor;
import com.smsuite.logic.stress.SimpleStressTestOperation;

@Path("/test/stress-test")
public class StressTestRequest {

	/*
	 * Message id is static to allow message id to be synchronized across all
	 * threads of the executing Servlet
	 */
	private static Long messageId = 1L;

	@Context
	UriInfo uriInfo;
	@Context
	HttpServletRequest request;

	// Application integration
	@GET
	@Produces(MediaType.APPLICATION_JSON)
	public Response login()
			throws EncoderException {
		try {
			/*
			 * thread safe increment the unique static message id
			 */
			Long id = incrementMessageId();

			/*
			 * Simple operation decorated with NotificationOperation interface
			 * will publish a message to JMS broker
			 */
			Operation op = new SimpleStressTestOperation(id);

			/*
			 * Execute the operation which will trigger logger and JMS publisher
			 */
			Result result = BusinessOperationExecutor.executeOperation(op);
			JMSProviderFactory.shutdownProvider();
			return Response.ok(new WebServiceResult(result)).build();
		} catch (Exception e) {

			e.printStackTrace();
			if (e instanceof Result) {
				return Response.ok(new WebServiceResult((Result) e)).build();
```

```
                    }

                    return Response.ok(new WebServiceResult(ResultCode.FAILED,
e.getMessage(), e)).build();
                }
        }

        private synchronized Long incrementMessageId(){
                messageId = new Long(messageId.longValue()+1);
                return new Long(messageId);
        }
}
```

SimpleStressTestOperation Java Test Class Source Code

```
package com.smsuite.logic.stress;

import javax.jms.TextMessage;

import org.apache.log4j.Logger;

import com.csm.backend.Result;
import com.csm.backend.ResultCode;
import com.csm.backend.logic.operation.MaintenanceOperation;
import com.csm.backend.logic.operation.NotificationOperation;
import com.csm.backend.messaging.Encoder;
import com.csm.backend.messaging.ProducerTopics;
import com.csm.backend.messaging.annotations.Message;
import com.csm.backend.messaging.annotations.Notification;

/**
 * Simple operation which logs the time-stamp of when the operation was executed.
 * along with a unique message id
 * <p>
 * The operation is annotated with Notification annotation
 * which will automatically send a JMS message to a STRESS_TEST queue
 *
 * Unique message id allows for the JMS message to be tracked on the
 * receiving end and time delay can be calculated from the time stamp
 * @author David Khanaferov
 *
 */
@Notification(deliveryMode = 2, encoding = TextMessage.class, topic =
ProducerTopics.STRESS_TEST)
public class SimpleStressTestOperation implements MaintenanceOperation,
NotificationOperation{

private static final String name = "STRESS-TEST Operation";

        //get a logger instance
        static Logger  logger = Logger.getLogger(SimpleStressTestOperation.class);

        @Message(encoder=Encoder.JSON)
        StressTestMessage message;

        public SimpleStressTestOperation(Long id){
                message = new StressTestMessage();
                message.setMessageId(id);
                message.setTimeStamp(System.currentTimeMillis());
        }

        @Override
        public void logBegin() {
                // TODO Auto-generated method stub

        }

        @Override
```

```
        public void logSuccess() {
                // TODO Auto-generated method stub

        }

        @Override
        public void logFailed() {
                // TODO Auto-generated method stub

        }

        @Override
        public Result execute() throws Result {
                logger.debug("MSG_ID="+message.getMessageId()+"|
TIME_STAMP="+message.getTimeStamp());
                return new Result(ResultCode.SUCCESS);
        }

        @Override
        public String getName() {
                // TODO Auto-generated method stub
                return name;
        }

        @Override
        public void isInMaintenance() {
                // TODO Auto-generated method stub

        }

        @Override
        public void isUpcomingMaintenance() {
                // TODO Auto-generated method stub

        }

}
```

APPENDIX I

Analyzer Java Stress Test Class Source Code

```
package com.smsuite.testing;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.lang.instrument.IllegalClassFormatException;
import java.nio.channels.IllegalBlockingModeException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Stress test data analysis algorithms. Contains a set of algrothims which
 * process a log file generated by the stress test JMS Java Applet client. This
 * class calculates data loss ratio, latency, data gaps. Presents results in a
 * human readable format.
 *
 * @author David Khanaferov
 *
 */
public class Analyzer {

	private Map<Long, Long> msgList;

	public Analyzer() {
		msgList = new HashMap<Long, Long>();
	}

	public void calculateTotals(String testName, int totalNumberRequests,
			int totalSuccessful, int totalNetworkErrors, int totalTimeouts,
			InputStream stream) {
		try {
			msgList = parseDataFile(stream);

			// output
			System.out.println();
			System.out.println(generateLine('-', 50));
			System.out.println("TEST [" + testName + "]:");
			System.out.println(generateLine('-', 50));

			System.out.println("Total requests sent: " + totalNumberRequests);
			System.out.println("Total successful requests: " + totalSuccessful);
			System.out.println("Network errors: " + totalNetworkErrors);
			System.out.println("Timeout errors: " + totalTimeouts);
			System.out.println("Error ratio: "
					+ (1 - totalSuccessful / totalNumberRequests) * 100 +
"%");

			System.out.println(generateLine('-', 50));
			System.out.println("Data loss: "
					+ calculateDataLoss(totalSuccessful));
			System.out.println("Data latency: " + calculateAverageLatency());

		} catch (Exception ex) {
			System.err.println(ex.getClass().getSimpleName() + ": "
					+ ex.getMessage());
		}
	}

	private String generateLine(char c, int length) {
```

```
                StringBuilder builder = new StringBuilder();
                for (int i = 0; i < length; i++) {
                        builder.append(c);
                }

                return builder.toString();
        }

        private Map<Long, Long> parseDataFile(InputStream data) throws IOException {
                if (data == null) {
                        throw new NullPointerException("data");
                }

                Map<Long, Long> parsedList = new HashMap<Long, Long>();
                BufferedReader reader = new BufferedReader(new InputStreamReader(data));
                String line = null;

                while ((line = reader.readLine()) != null) {

                        try {
                                line = line.trim().toLowerCase();
                                if (line == null || line.isEmpty()) {
                                        continue;
                                }

                                String[] pairs = line.split(",");
                                if (pairs.length != 3) {
                                        throw new IllegalBlockingModeException();
                                }
                                for (int j = 0; j < pairs.length; j++) {
                                        String[] vals = pairs[j].split(":");
                                        if (vals.length != 2) {
                                                throw new IllegalClassFormatException("corrupt
message");
                                        }
                                        if (vals[0].trim().equalsIgnoreCase("messageId")) {
                                                parsedList.put(
                                                                Long.parseLong(vals[1]),
                                                                Long.parseLong(pairs[j +
2].split(":")[1])
                                                                                -
Long.parseLong(pairs[j + 1]
                                                                                                .
split(":")[1]));
                                                j++;
                                        }
                                }

                        } catch (Exception e) {
                                System.out.println("got bad message: " + e.getMessage());
                        }
                }

                reader.close();

                return parsedList;
        }

        private String calculateDataLoss(int totalSent) {

                List<Long> gaps = new ArrayList<Long>();
                Long currGap = null;
                long prev = 0;

                List<Long> keys = new ArrayList<Long>(msgList.keySet());

                Collections.sort(keys);

                for (Long msgId : keys) {
                        if (prev != 0) {
```

```
                                if (msgId - prev == 1) {
                                        // no gap
                                } else {
                                        currGap = msgId - prev;

                                        /*
                                         * We subtract 1 from the gap value to get the number
of
                                         * missing messages. For example if two adjacent id's
are 3
                                         * and 5 the gap is 2 however only 1 id is missing
from the
                                         * sequence
                                         */

                                        gaps.add(currGap - 1);
                                }
                        }
                        prev = msgId;
                }

                StringBuilder builder = new StringBuilder();
                long totalLost = 0;
                long[] gapVals = new long[gaps.size()];
                int i = 0;
                for (Long gap : gaps) {
                        totalLost += gap;
                        gapVals[i] = gap;
                        i++;
                }

                builder.append("\n\t\tTotal lost data: " + totalLost + "\n");
                builder.append("\t\tData loss ratio: " + ((double) totalLost)
                                / (msgList.size() + totalLost) * 100 + "%\n");
                builder.append("\t\tData gap mode: " + getMode(gapVals) + "\n");
                if (gaps.size() > 0) {
                        builder.append("\t\tData gap average: " + ((double) totalLost)
                                        / gaps.size() + "\n");
                } else {
                        builder.append("\t\tData gap average: " + 0 + "\n");
                }
                builder.append(generateLine('-', 50));
                builder.append("\nTest accuracy: "
                                + ((double) (msgList.size() + totalLost)) / totalSent * 100.0
                                + "%");
                return builder.toString();
        }

        private String calculateAverageLatency() {

                double total = 0.0;

                for (Long msgId : msgList.keySet()) {
                        total += msgList.get(msgId);
                }
                return total / msgList.size() + " ms";
        }

        public static long getMode(long[] values) {
                HashMap<Long, Long> freqs = new HashMap<Long, Long>();

                for (long val : values) {
                        Long freq = freqs.get(val);
                        freqs.put(val, (freq == null ? 1 : freq + 1));
                }

                long mode = 0;
                long maxFreq = 0;

                for (Map.Entry<Long, Long> entry : freqs.entrySet()) {
```

```
                        long freq = entry.getValue();
                        if (freq > maxFreq) {
                                maxFreq = freq;
                                mode = entry.getKey();
                        }
                }

                return mode;
        }

}
```