HIGH AVAILABILITY QUEUING SYSTEM

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Firas Arabo

Spring 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Firas Arabo:

High Availability Queuing System

__________________________
Roger Whitney, Chair
Department of Computer Science

__________________________
John Donald
Department of Computer Science

__________________________
Yusuf Ozturk
Department of Electrical & Computer Engineering

12/8/2010
Approval Date
Copyright © 2011

by

Firas Arabo

All Rights Reserved
DEDICATION

I would like to dedicate this thesis project to the late Layla Toma, my mother who gave me endless love and care. I would also like to dedicate it to my wife, Luma Zakaria; the love and support she has been giving me is exceptional.
ABSTRACT OF THE THESIS

High Availability Queuing System
by
Firas Arabo
Master of Science in Computer Science
San Diego State University, 2011

Queuing systems have been gaining a rapid popularity in the software industry. Web applications and web services use queuing systems to hold messages produced by a producer component of a system. These messages later will be either picked up by a consumer component or pushed by the queuing system to registered consumers. The industry offers quite a few queuing systems, some of which are open source.

High availability of a queuing system is the main concern for users when using such systems. Queues and messages managed by the system must be available at all times which are hard to achieve in case of a system crash. Amazon offers a highly available queuing system called SQS. SQS assures high availability of queues and messages at all times. Amazon achieves the state of high availability by distributing the messages into multiple servers. Not all servers have the same messages. Crashing of one or more queue servers does not impact the availability of queues and messages. SQS samples a subset of servers based on random weighed distribution algorithm when retrieving a message. SQS has some limitations resulting from inconsistency of servers. For example, a message might be returned more than once, messages are not retrieved in the order they were received and if a queue has few messages, 1000 or less, a retrieve message request might return no message and users must make consecutive retrieve requests.

In this thesis project, a queuing system named “VSQS” is introduced for academic research only and it is not available for commercial use. This system is derived from Amazon SQS and uses the same mechanism of distributing a message into multiple servers to achieve high availability. However, VSQS takes a different approach in creating the subset; in that, a subset is preselected for each queue created and all messages of that queue go to the same subset. The advantage of pre-selecting a subset is to have better control over the subset and reduce the period of inconsistency between servers. This helps to overcome the limitations that appear when using SQS; the most important one is that the message will not be retrieved more than once. On the other hand, the disadvantage is that the chances of unavailability are increased when one or more servers of the pre-selected subset go down. This thesis studies the effects of pre-selecting a subset.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT...............................................................................................................................v</td>
</tr>
<tr>
<td>LIST OF TABLES ................................................................................................................ix</td>
</tr>
<tr>
<td>LIST OF FIGURES .................................................................................................................xi</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS.........................................................................................................xiv</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1 INTRODUCTION .............................................................................................................1</td>
</tr>
<tr>
<td>1.1 Introduction...............................................................................................................1</td>
</tr>
<tr>
<td>1.2 Queuing Systems ......................................................................................................1</td>
</tr>
<tr>
<td>1.3 Importance of the Thesis.........................................................................................3</td>
</tr>
<tr>
<td>1.4 Thesis Outline..........................................................................................................3</td>
</tr>
<tr>
<td>2 BACKGROUND AND REVIEW ....................................................................................5</td>
</tr>
<tr>
<td>2.1 Queue Term .............................................................................................................5</td>
</tr>
<tr>
<td>2.2 Queue as Data Structure.........................................................................................5</td>
</tr>
<tr>
<td>2.3 Queue as Repository...............................................................................................7</td>
</tr>
<tr>
<td>2.4 Queue as Messaging Service .................................................................................8</td>
</tr>
<tr>
<td>2.5 JMS ..........................................................................................................................9</td>
</tr>
<tr>
<td>2.5.1 JMS API Architecture.......................................................................................9</td>
</tr>
<tr>
<td>2.5.2 Messaging Domains.........................................................................................10</td>
</tr>
<tr>
<td>2.5.2.1 POINT-to-Point Messaging Domain .........................................................11</td>
</tr>
<tr>
<td>2.5.2.2 Publish/Subscribe Messaging Domain ......................................................12</td>
</tr>
<tr>
<td>2.5.3 Message Consumption......................................................................................13</td>
</tr>
<tr>
<td>2.6 JMS Providers.........................................................................................................13</td>
</tr>
<tr>
<td>2.6.1 JBOSS Messaging............................................................................................14</td>
</tr>
<tr>
<td>2.6.2 Apache Active MQ..........................................................................................14</td>
</tr>
<tr>
<td>2.7 High Availability Queuing System (HAQS) .........................................................15</td>
</tr>
<tr>
<td>2.8 Amazon SQS..........................................................................................................15</td>
</tr>
<tr>
<td>2.8.1 SQS Features.................................................................................................16</td>
</tr>
</tbody>
</table>
3.5.4.2 Get Message ................................................................. 61
3.5.4.3 Delete Message ............................................................ 62
3.5.4.4 Return to Queue ........................................................... 63

3.5.5 Queue Server Life Cycle .................................................... 63

3.5.5.1 Queue Server – Available State .................................... 64
3.5.5.2 Queue Server – Unavailable State ................................. 64
3.5.5.3 Queue server – Synchronizing State .............................. 65

4 SOFTWARE TEST AND RESULTS ...................................................... 67

4.1 Types of Software Testing ...................................................... 67
4.2 Testing VSQS ................................................................. 68

4.2.1 Unit Testing ................................................................. 68
4.2.2 Functional Testing ......................................................... 71

4.2.3 Performance and Load Testing .......................................... 72

4.2.3.1 JMeter ................................................................. 73
4.2.3.2 General System Performance ..................................... 76
4.2.3.3 System Availability .................................................. 80
4.2.3.4 System Latency ....................................................... 91
4.2.3.5 System Consistency ................................................ 97
4.2.3.6 Message Properties ............................................... 98

4.3 Testing Summary .............................................................. 99

5 CONCLUSION AND RECOMMENDATIONS ............................... 101

5.1 Amazon SQS ............................................................... 102
5.2 VSQS ................................................................. 104
5.3 VSQS Test Results .......................................................... 105
5.4 Conclusion ................................................................. 106
5.5 Recommendations ........................................................ 107

BIBLIOGRAPHY ................................................................................. 108
LIST OF TABLES

Table 3.1. Schema Description of Database Table Named “queue” ........................................34
Table 3.2. Schema Description of Database Table Named “message” ....................................36
Table 3.3. Schema Description of Database Table “subset” ..................................................43
Table 3.4. Schema Description of Database Table “subsetlocation” .......................................44
Table 3.5. Schema Description of Database Table “queues” ..................................................44
Table 3.6. Schema Description of Database Table “requestplaceholder” ...............................47
Table 3.7. Schema Description of Database Table “QSM” .....................................................54
Table 3.8. Schema Description of Database Table “susbsset” ..................................................55
Table 3.9. Schema Description of Database Table “qs” ..........................................................55
Table 4.1. Results of Few Tests of 10 Producers Making 1000 Put Message Requests With 0 to 1 Second Delay in Between Requests ..........................................................77
Table 4.2. Results of Few Tests of 10 Consumers Making 1000 Get Message Requests Followed by a Delete Request of the Retrieved Message ........................................80
Table 4.3. Availability and Unavailability Recordings of Subsets, 1 Minute of Down Time ..........................................................................................................................89
Table 4.4. Availability and Unavailability Recordings of Subsets, 2 Minutes of Down Time ..........................................................................................................................90
Table 4.5. Availability and Unavailability Recordings of Subsets, 4 Minutes of Down Time ..........................................................................................................................90
Table 4.6. Response Time Results in Milliseconds While Increasing the Request Size ..........92
Table 4.7. Minimum Response Time Recordings in Milliseconds When Cache Size Increases .......................................................................................................................93
Table 4.8. Maximum Response Time Recordings in Milliseconds While Increasing the Cache Size ....................................................................................................................93
Table 4.9. Average Response Time Recordings in Milliseconds While Increasing the Cache Size ....................................................................................................................94
Table 4.10. Results of 50% Threshold Line of the Distribution Graph of Response Time in Milliseconds While Increasing the Cache Size .........................................................95
Table 4.11. Results of 90% Threshold Line of the Distribution Graph of Response Time in Milliseconds While Increasing the Cache Size .........................................................95
Table 4.12. Page Fault Recordings of the Test in KB .............................................................96
Table 4.13. Results of Measuring Latency When Number of Threads Increase .................96
Table 4.14. Results of Measuring Latency When Shutting Down Queue Servers and Bringing Them Back Up.................................................................97
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Source</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Queue</td>
<td>Robert Lafore, Data Structures and Algorithms in Java, second edition, Sam publishing, Indianapolis, Indiana, 2002, pp. 143</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Deque</td>
<td>Robert Lafore, Data Structures and Algorithms in Java, second edition, Sam publishing, Indianapolis, Indiana, 2002, pp. 144</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>JMS API Architecture</td>
<td>Java Message Service Tutorial. Basic JMS API concepts, <a href="http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html">http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html</a>, accessed October 2010</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Point-to-Point Messaging Domain</td>
<td>Java Message Service Tutorial. Basic JMS API concepts, <a href="http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html">http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html</a>, accessed October 2010</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Publish/Subscribe Messaging Domain</td>
<td>Java Message Service Tutorial. Basic JMS API concepts, <a href="http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html">http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html</a>, accessed October 2010</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>SQS Architectural overview</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Put a message into a queue</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Retrieve a message from a queue</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Retrieving messages is done by sampling a subset of servers</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Message visibility timeout</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>Deleting the message “A” from the queue</td>
<td>Amazon web services. Amazon Simple Queue Service (Amazon SQS), <a href="http://aws.amazon.com/sqs/">http://aws.amazon.com/sqs/</a>, accessed October 2010</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.12</td>
<td>Single instance of queue server ensure consistency</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.13</td>
<td>Queue distributed among two servers</td>
<td></td>
<td>27</td>
</tr>
</tbody>
</table>
Figure 2.14. One queue distributed among many servers. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010. .................................................28

Figure 2.15. Retrieving a message by sampling a subset of queue servers in gray and return the messages from the sampled servers only. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010. .................................................30

Figure 3.1. Queue server instance holding three queues. ........................................................33

Figure 3.2. A subset consists of five queue servers. .................................................................33

Figure 3.3. Class hierarchy of queue server data access object management .........................38

Figure 3.4. “WebService.java” and its methods as appear in the IDE....................................39

Figure 3.5. QSM calls all queue servers. .................................................................................42

Figure 3.6. Class hierarchy of QSM data access object management. ..................................45

Figure 3.7. VSQS System infrastructure..................................................................................56

Figure 3.8. Create queue flow chart. ......................................................................................59

Figure 3.9. Delete queue flow chart. ......................................................................................60

Figure 3.10. Put message flow chart. ......................................................................................61

Figure 3.11. Get message flow chart. ......................................................................................62

Figure 3.12. Delete message flow chart. ..................................................................................63

Figure 3.13. QSM forward request received to available queue server. .................................64

Figure 3.14. Requests are put in place holder, QSM monitors the queue server..................65

Figure 3.15. Queue servers in synchronizing state. .................................................................66

Figure 4.1. General flow of processing requests is used in functional testing. .......................71

Figure 4.2. Jmeter summary report example. ....................................................................... 74

Figure 4.3. Distribution graph sample that shows 50% and 90% threshold lines. ................. 74

Figure 4.4. Spline Visualizer graph sample. ...........................................................................75

Figure 4.5. Distribution graph of 10 producers making 1000 put message requests with zero to one second delay in between requests. .........................................................76

Figure 4.6. Spline Visualizer graph of 10 producers making 1000 put message requests with zero to one second delay in between requests. The peak is at the beginning of the run only.............................................................77

Figure 4.7. Distribution graph of 10 consumers making 1000 get message requests followed by a delete request of the retrieved message. A delay of zero to one second is applied to the consumers.................................................78
Figure 4.8. Spline Visualizer graph showing 10 consumers making 1000 get message requests followed by a delete request of the retrieved message. A delay of zero to one second is applied to the consumers. .................................................................79

Figure 4.9. Spline Visualizer of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. .........................................................81

Figure 4.10. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. .........................................................81

Figure 4.11. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run. .................................................................82

Figure 4.12. Spline Visualizer of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run. .................................................................82

Figure 4.13. Distribution graph of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run. .................................................................82

Figure 4.14. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. Queue servers go down randomly with at least one queue server available at any time. .................................83

Figure 4.15. Spline graph of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. Queue servers go down randomly with at least one queue server available at any time. ........................................84

Figure 4.16. Distribution graph of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. Queue servers go down randomly with at least one queue server available at any time. ................85

Figure 4.17. Summary report of running 10 producers; each puts 2000 messages with a delay range between 0 to 1 second in between requests. Queue servers go down randomly. ...........................................................................................................86

Figure 4.18. Spline graph of running 10 producers; each puts 2000 messages with a delay range between 0 to 1 second in between requests. Queue servers go down randomly. ...........................................................................................................86

Figure 4.19. Distribution graph of running 10 producers; each puts 2000 messages with a delay range between 0 to 1 second in between requests. Queue servers go down randomly. ...........................................................................................................87

Figure 5.1. Direct communication between components “A” and “B.” .................................101

Figure 5.2. Queuing mechanism used to transfer messages between components. ...........102
ACKNOWLEDGEMENTS

First and foremost I offer my sincerest gratitude to my supervisor, Dr Roger Whitney, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I could not wish for a better or friendlier supervisor. I am heartily thankful to Dr. John Donald, throughout my thesis-writing period, he provided me with an exceptional teaching, sound advice, and lots of great ideas. I attribute the level of my Masters degree to their encouragement and effort and without them this thesis would not have been completed or written.
CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The purpose of this thesis project is to study and build a high availability queuing system. Today’s web applications and web services require that thousands of messages be passed from one component to another every second. Most of these messages are not trivial and must always be available for processing; therefore high availability is the main factor in building a queuing system since the queue holding messages must be available at any time, even in case of partial system crashing. The queuing system receives messages generated by producers and makes them available for retrieval only when consumers decide to process the message. Because producers and consumers usually are distinct components and most likely reside in different machines, providing a queuing system that decouples them is highly desirable. The queuing system is built as a service that requires neither setup nor maintenance by the end user; it provides a very simple web-service API to create and manage queues. Amazon has built a high availability queue called “SQS” based on duplicating a message into multiple servers to ensure its availability even if one of the systems that have a copy of the message crashes. This thesis project is not an attempt to study or evaluate Amazon SQS; rather it is an attempt to build a similar system that is derived from Amazon SQS to overcome some of its limitations.

1.2 QUEUING SYSTEMS

The rising need of web application and web services to a queuing mechanism resulted in many software available to help user’s application to queue messages and retrieve them at later time and ensure the high availability of their queues. The majority of these software’s are integrated into the user’s application where, in some cases is not trivial, a set up is required and some coding implementation also needed to use these software. Amazon SQS takes a different approach in building a queuing system that requires no set up and no extra implementation to use the queuing mechanism instead it
provide a simple Web Service API for the user to use to create queues and push or retrieve messages.

Amazon SQS achieves high availability by distributing queues and messages into multiple servers called a subset, the subset of servers is chosen based on random weighted distribution, a different subset can be chosen for messages even if they belong to the same queue. Messages, similarly a random subset is chosen when retrieving a message. When a server crashes, messages can be served by other available servers that have the same copy of messages, the only scenario of a message being unavailable is when all servers containing the message crashes, the more servers used to hold a copy of the same message the more unlikely for this scenario to happen.

Amazon SQS servers relax the consistency in favor of high availability and partition tolerance. The first factor affecting the inconsistency is the randomness of selecting a subset of servers to hold copy of messages even if they belong to the same queue. All servers are in one pool, a different subset is chosen for every message arrives to the system where the message is saved, and it is a possibility that a different subset is chosen for a second message even if it is belong to the same queue as the previous saved message. Similarly retrieving a message is done by randomly selecting a subset and returns the messages found in this subset only. The second factor is when a server comes back online after it was down, the server missed all the requests arrived to the system during the down time therefore it might hold retrieved or deleted messages. Relaxing the inconsistency resulted in some issues like:

- The messages might not be retrieved in the same order they arrived.
- User’s consumers might receive the message more than once.
- Retrieve request may not result in returning a message even though the queue has messages.

Amazon SQS is an eventually consistent system where all servers will eventually sync up but it is not clear what the time frame is, it could be few seconds in cases like deleting all copies of a message or few hours when a server comes back online after it was crashed.
1.3 | IMPORTANCE OF THE THESIS

This thesis tries to build a queuing system named “VSQS” that is similar to Amazon SQS in that it achieves high availability of queues and messages by replicating messages into multiple servers, VSQS also favors availability and partition tolerance over consistency. This thesis is neither an attempt to evaluate Amazon SQS nor building a system that compete with it; rather it is an attempt to overcome the issues found in Amazon SQS resulted from over relaxing the inconsistency. Out of order retrieving of messages, retrieving messages more than once or not returning messages contained in the queue changes the fundamental characteristics of queues. The goal of this thesis is keep these fundamentals as well as achieving the high availability.

VSQS takes a different approach when it comes to select a subset, instead of choosing a subset based on weighted random distribution, VSQS preselect the subset of servers where each queue and all it messages go to the same preselected subset. Preselecting a subset gives a better control over the queue, the inconsistency is reduced to the time required to distribute incoming requests to all servers within a subset, and also it is easier to sync up servers that came back online after crashing. Creating a subset from a definite number of servers might increases the likelihood of unavailability if all the servers in the subset go down at the same time. This thesis is to study the effectiveness of preselecting the subset of servers.

1.4 | THESIS OUTLINE

The thesis will be comprised of 5 chapters. Following the introductory chapter, which outlines the nature of the study, Chapter 2 will present the history of queuing in computer science, open source versus non open source software that offers queuing mechanism and then focus on Amazon SQS as a distinctive software available for users to manage their queues without any set up or integration and study the limitation of SQS. Chapter 3 will be devoted to define and build a queuing system that is similar to SQS that aims to reduce the limitation found in SQS, the system built called VSQS. The chapter explains all three main components of the system, System Manager (SM), Queue Server Manager (QSM) and Queue Server (QS) and how they communicate with each other. Chapter 4 will present the various testing mechanisms conducted on VSQS and the results
of these tests. It will start by showing how unit testing was done followed by explanation of functional testing ending with performance testing that shows how VSQS overcame the limitations of message properties appears in Amazon SQS while still keeping the high availability characteristic. Chapter 4 also shows various response time recording. Finally, Chapter 5 will conclude the study and provide recommendation to enhance VSQS based on the test results.
CHAPTER 2

BACKGROUND AND REVIEW

2.1 QUEUE TERM

Queue, in my own words, is a collection of items waiting in line. It is defined in Webster’s dictionary [1] as follows:

“1. a braid of hair worn hanging down behind. 2. a file or line, esp. of people waiting their turn. 3. a sequence of items waiting in order for electronic action in a computer system. 4. to form in a line while waiting. 5. to arrange or organize into a queue.”

Computer scientists have been using queues since the early days of computing. The main idea of queues in computer science is to hold items in order waiting to be retrieved. Software engineers then expanded the idea by adding different characteristics to meet various needs in solving problems in building software. We will discuss the usage of the queue in computer science in the coming sections.

2.2 QUEUE AS DATA STRUCTURE

A queue in computer science is a data structure collection designed for holding items that are waiting to be processed. The items are inserted to the rear of the queue and the removed from the front of the queue. This makes the queue a First In - First Out (FIFO) data structure Figure 2.1.

Figure 2.1. Queue. Source: Robert Lafore, Data Structures and Algorithms in Java, second edition, Sam publishing, Indianapolis, Indiana, 2002, pp. 143.
In a FIFO data structure [2], the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure. A queue is like a line in real life, such as students forming a line to pay their tuition. The first student who joins the line is the first to be serviced, and a new student joining the line goes to the rear of the line and must wait for all students in front to be served first. Many computer applications use queues: when you type using the keyboard the keystrokes go into a queue and the application takes these keystrokes one after another as they were entered.

Computer Science expands the concept of a queue and adds more data structures that are based on queues but behave differently. An application might have different requirements on how the queue should work. To support this need new types of queues were designed, all of which allow adding some items into the queue and retrieval of those items. All ensure adding the item only once and retrieving it only once, even if there are multiple threads reading off the queue. Also the item is not available for retrieval until it is inserted into the queue. A type of queue called deque (pronounced deck), which stands for Double Ended QUEue [2], allows adding and removing items from both the front and the rear of the queue Figure 2.2.

![Figure 2.2. Deque. Source: Robert Lafore, Data Structures and Algorithms in Java, second edition, Sam publishing, Indianapolis, Indiana, 2002, pp. 144.](image)
Some systems want to process queued tasks based on a priority criterion other than FIFO. For example, an item with higher priority will be processed before an already existing lower priority item in a priority queue [2]. A delay queue [2] can store elements that are not available until after a delay period is passed.

The original definition of a queue in computer science involves items forming a line waiting for something. The definition, however, changes if the item added or removed is dependent on how the items should be inserted in the queue or which items are to be processed first. For example, in a priority queue the first item inserted is no longer the first to be retrieved from the queue. Also, the first item might not be in the rear of the queue, as in deque.

2.3 QUEUE AS REPOSITORY

The queue as a data structure is usually used internally by a standalone application. Web based applications and web services need a different queuing mechanism that match the nature of individual web applications. Web applications usually consist of multiple components that reside on multiple machines and serve thousands of users or requests simultaneously. Queue based mechanisms can by be used to solve issues that web applications run into:

- The application might not be able to process all requests it receives at the same time. One approach is to queue these requests first before processing.
- The application needs to schedule tasks to run on certain dates and times. They can be deposited into a persistent queue and then processed on the due date, e.g. recurring charges for subscriptions or sending reminder messages to users.
- A producer component of an application produces items and dumps them into a repository; and it is up to another component, usually called consumer, to pick up these items and process them.

The queue needed for the above examples can be defined as a temporary repository of items that are waiting for processing. This temporary repository works as a place holder for all items produced by some component, usually called producer, and to be retrieved by another component, usually called consumer. The queue usually is an independent component designed to receive items from one or many producers and allow one or more consumers to retrieve those items. These three components are normally designed to be independent in web applications even if they reside in the same application. It is not
unusual to see the components on different applications or even on different machines. The queue component assists in decoupling producers from consumers.

The queue can be non-persistent: messages are kept in memory, and are available as long as the server containing the queue is running and there are enough resources to hold messages. The number of messages that can be kept in a non-persistent queue relies on the availability of memory. The queue will fill up fast if the consumer is not retrieving any messages while the producer is still putting messages into the queue. In case of a system crash, all messages saved into the queue prior to the crash will be lost for good.

This leads to the necessity of having a persistent queue where messages are saved either in a file system on disc or using some database management system. Keeping messages in a persistent device has the advantage of freeing resources and holding many more messages compared to a non-persistent queue. However, a persistent queue takes more CPU time to save and retrieve a message. To reduce the CPU time, we can make a persistent queue that has a number of messages available in memory for fast retrieval.

The queue may reside on the same box as the producer and/or the consumer. If either of the boxes containing the queue fails, all components including the queue will not be available. Having the queue on a separate box minimizes this possibility since the queue is decoupled from either component. The queue can be shared by multiple producers and consumers. In the above example we have only a single queue; if this queue is disabled for any reason and becomes unavailable, producers will not find any place to put the messages produced and consumers will not find any messages to process until that single queue comes back to life. Depending on how the producer and consumer are designed, we might lose some messages. Whatever the cause of the failure, we need to find a solution by designing a high availability queue that serves producers and consumers under any circumstance.

2.4 Queue as Messaging Service

Message queuing is a widely adopted technology for developing distributed processing systems where business logic extends to multiple and potentially remote systems. The asynchronous nature of message queuing enables applications on geographically separated systems to interact effectively, without the difficulty of
synchronizing activities on communication ends that is usually associated with the conversational client/server model. This makes message queuing a preferred technology for many large scale e-commerce applications.

Message queues normally provide an asynchronous communications mechanism where a sender can push messages into the queue without the need to know if the receiver is actually connecting at the same time to retrieve message. Messages placed onto the queue are stored until the recipient retrieves them. This will also helps in decoupling the producers from consumers, each component can run independently from the other and they most likely reside in two different physical boxes. Most message queues have set limits on the size of data that can be transmitted in a single message. Those that do not have such limits are known as mailboxes.

The focus of this thesis project is to study queue as messaging service and how to build a messaging service that meets current needs of many web applications and web services with many components communicating through messaging protocols. The queuing system built in this thesis project is based on Amazon SQS which takes a different approach in solving queuing needs compared to other open source software that are based on JMS, therefore the following sections will study JMS and few messaging system providers and then discuss Amazon SQS.

### 2.5 JMS

Java provides Message-oriented middleware (MOM) [3] [4] API called Java Messaging System API used for sending messages between two clients or more. JMS is a part of the Java Platform, Enterprise Edition [5], and is defined by a specification developed under the Java Community Process as JSR 914 [6]. It is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition J2EE to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous.

#### 2.5.1 JMS API Architecture

A JMS application is composed of the following parts [7]:

- **A JMS provider**: is the implementation of the JMS interface that result in a messaging system, it provides features to controls and administer the system.
- **JMS clients**: clients can be either a producer of a message or a consumer of a message; both are written in java programming language.

- **Messages**: represents the objects of information that are communicated between the clients.

- **Administered objects**: are the JMS destination and connection factories objects that are preconfigured by an administrator for the use of producers and consumers.

- **Native clients** are programs that use a messaging product's native client API instead of the JMS API. An application first created before the JMS API became available and subsequently modified is likely to include both JMS and native clients.

Figure 2.3 illustrates how the parts listed above interact with each other. Administrative tools allow the user to bind destinations and connection factories into a Java Naming and Directory Interface (JNDI) API namespace. A JMS client can then look up the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

![Figure 2.3. JMS API Architecture. Source: Java Message Service Tutorial. Basic JMS API concepts, http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html, accessed October 2010.](image)

### 2.5.2 Messaging Domains

There are two approaches for messaging to be transferred between components normally from producers to consumers; the two approaches are “Point To Point” and “Publish/Subscribe”. The JMS specification provides a separate domain for each approach and defines compliance for each domain. It is up to the developers of JMS providers to implement one or both of the domains however most of the implementation for JMS provider available currently in the industry support both domains, and some take
it the extra step of having the client to combine the user of both, such clients extends the power of JMS API and supplements the flexibility of messaging. In the other hand J2EE provider must implements both domains. The following two sections explain each messaging domain.

2.5.2.1 POINT-TO-POINT MESSAGING DOMAIN

Point-to-point (PTP) messaging domain is based on the idea of message queues that senders push messages to, those messages are waiting to be consumed by receivers. Figure 2.4 illustrates how PTP works, the queue is created with an address known to sender shown as “Client 1” in the figure and also known to receiver which is shown as “Client 2”. Senders push all messages to the specified queue, the messages will wait in the queue until a receiver request a message or a message expires, receivers normally acknowledge receiving and processing the message. PTP messaging domain has the following characteristics: Each message has only one consumer. A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message. The receiver acknowledges the successful processing of a message. PTP messaging domain is recommended to be used when every message pushed to the queue must be processed successfully by only one consumer.

Figure 2.4. Point-to-Point Messaging Domain. Source: Java Message Service Tutorial. Basic JMS API concepts, http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html, accessed October 2010.
2.5.2.2 Publish/Subscribe Messaging Domain

Publish/Subscribe (pub/sub) messaging domains is based a topic with an address used by the publisher clients to send messages to the topic and the messaging system takes care of distributing the messages to all subscribers registered on that topic. Figure 2.5 illustrates pub/sub messaging domain. Publishers and subscribers may change dynamically, the system will push the messages to all the subscribers registered on the topic at the time of sending the message, the already sent message will not be sent to future subscribers. Topics retain messages only as long as it takes to distribute them to the current subscribers. Topics can have multiple publishers and subscribers at a time.

Pub/sub messaging domain has the following characteristics: Topic is the center of the domain. Multiple publishers can publish messages to a topic at once. Published messages can be distributed to as many subscribers on the topic. Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order to consume messages.

![Figure 2.5. Publish/Subscribe Messaging Domain. Source: Java Message Service Tutorial. Basic JMS API concepts, http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html, accessed October 2010.](image-url)
The timing dependency may become an issue to many applications therefore JMS API relaxes this timing dependency to some extent by allowing clients to create what so called **durable subscriptions**. Durable subscriptions can receive messages sent while the subscribers are not active normally the topics keeps a copy of the messages until the subscriber becomes active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. Pub/sub messaging domain must be chosen carefully because a message can be lost if there are no subscribers at the time of publishing a message. This messaging domain is perfect choice when each message needs to be processed by zero, one, or many consumers.

### 2.5.3 Message Consumption

Message consumption is asynchronous by default for messaging products in that consumption of a message is not timely tied to the production of the message. However, the JMS Specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- **Synchronously.** A message can be consumed synchronously after it is produced when a subscriber or a receiver explicitly retrieves the message from the destination by calling the receive method. The receive method can block until a message arrives therefore a message will be handed from the producer to the consumer almost instantly. The receive method can also time out if a message does not arrive within a specified time limit.

- **Asynchronously.** A message is consumed asynchronously after it was produced in an asynchronous path e.g. a client can register a message listener, whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's **onMessage** method, which acts on the contents of the message.

### 2.6 JMS PROVIDERS

To use JMS, one must have a JMS provider that can manage the sessions and queues. Examples of commercial implementations are IBM’s Web Sphere MQ (formerly MQ Series) and Oracle Advanced Queuing (AQ). There are a number of open source choices of messaging middleware systems, including JBOSS Messaging, JORAM, ACTIVEMQ, Open Message Queue, Apache QPID and HTTPSQS. For the sake of this thesis we will briefly describe two examples of queue messaging systems: JBOSS and
**ACTIVEMQ**, these two systems are open source and used widely in the industry, describing them as examples can help in comparing JMS providers with Amazon SQS.

### 2.6.1 JBOSS Messaging

JBOSS is an application server that offers messaging queuing capability [8], the provider is implemented based on Java messaging service (JMS) and supports both messaging domains PTP and PUB/SUB. JBOSS queuing system characteristics are as follows:

- **Point to Point (P2P):** a sender delivers messages to a queue and a single receiver pulls the message off the queue. The receiver does not need to be listening to the queue at the time the message is sent.

- **The JMS publish/subscribe (Pub/Sub):** message model is a one-to-many model. A publisher sends a message to a topic and all active subscribers of the topic receive the message. Subscribers that are not actively listening to the topic will miss the published messages.

- **Durable JMS** supports a messaging model that is a cross between the P2P and pub/sub models. When a pub/sub client wants to receive all messages posted to the topic it subscribes to, even when it is not actively listening to the topic, the client may achieve this behavior using a durable topic.

- **Clustered destinations (queues and topics)** can be deployed at all or none of the nodes of the cluster. A JMS client uses HA JNDI to lookup the connection factory. When creating connections using that connection factory, a client side load balancing policy will automatically choose a node to connect to.

### 2.6.2 Apache Active MQ

Apache active MQ [9] is an open source offers message queuing solutions supported in many languages such as Java, C, C++, Ruby, PHP, Perl and Python. While fully supporting JMS 1.1 and J2EE 1.4. Apache Active MQ is released under the Apache 2.0 License. This messaging system provider has some distinct characteristics:

- **Support a Master/Slave Cluster** basically means that all messages are replicated across each broker in the master/slave cluster. If the Master goes down, the clients can automatically failover to a slave which will have all the messages already, so each message is highly available. The Slave(s) provide a hot standby broker which will always be in sync, ready to take over if the master goes away due to hardware failure, etc.

- **It allows you to configure in a way to load balance the coming request into multiple brokers.**
All JMS based providers including the two examples described above require some work to allow application to use them; this work can involve installation of the messaging system, configuration and implementation of interfaces. Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. The challenge in designing a distributed queuing algorithm is to minimize message traffic and delay.

2.7 High Availability Queuing System (HAQS)

Having an unavailable queue is a serious issue for many web services and applications. HAQS is a queuing system that offers a solution to many web applications and web services to quickly and reliably queue messages and retrieve them at a later time. JMS based queuing systems offer a master/slave mechanism. The approach Apache Active MQ takes which is basically to distribute the queue into many servers by replicating the message into multiple brokers. All JMS providers require some work and maintenance to configure them into web applications and services; it will become trickier to achieve some advanced goals like having our ActiveMQ, for example, support both load balancing and failover. Amazon has created a rather interesting system that offers a high availability queuing system with a simple API to interact with. There is no need to spend any effort in setting it up or maintaining it. The following section is to discuss Amazon SQS system to understand the approach it took in providing a high availability queuing system.

2.8 Amazon SQS

Amazon inc. has built what is called a Simple Queuing System (SQS) [10]. SQS freed developers from all the work needed to integrate with all other queuing systems providers by providing a simple API where users can call to create queue, push messages into the queue, retrieve messages and then delete the queue once the user is done with it. In this thesis project I will studying SQS, build a similar system, and discuss the challenges and limitations of this system.

Amazon SQS is a distributed queue system that enables web service applications to quickly and reliably queue messages that one component in the application generates for consumption by another component. Using Amazon SQS, decoupled components of an
application run independently, with Amazon SQS easing message management between components. Any component of a distributed application can store any type of data in a fail-safe queue. Any other component can then later receive the data programmatically using the SQS API.

The queue acts as a buffer between the component producing and saving data, and the component receiving the data for processing. This means the queue resolves issues that arise if the producer is producing work faster than the consumer can process it, or if the producer or consumer is only intermittently connected to the network.

SQS ensures delivery of each message at least once, and supports multiple readers and writers interacting with the same queue. A single queue can be used simultaneously by many distributed application components, with no need for those components to coordinate with each other to share the queue.

Amazon SQS is engineered to always be available and deliver messages. One of the resulting tradeoffs is that SQS does not guarantee first in, first out delivery of messages. For many distributed applications, each message stands its own, and as long as all messages are delivered, the order is not important. If your system requires that order be preserved, you can place sequencing information in each message, so that you can reorder the messages when the queue returns them.

Amazon SQS works by exposing Amazon’s web-scale messaging infrastructure as a web service. Any computer on the Internet can add or read messages without any installed software or special firewall configurations. Components of applications using Amazon SQS can run independently, and do not need to be on the same network, developed with the same technologies, or running at the same time.

2.8.1 SQS Features

Amazon SQS provides quite few major features [11]. Redundant infrastructure offers high availability system for sending and retrieving messages, it guarantees delivery of messages at least once, concurrent producers can push messages and concurrent consumers can pull message simultaneously. Multiple writers and readers SQS locks the message during processing preventing simultaneous processing attempts of the same message by other components of the system. Configurable settings per queue user’s
queues can be created with different characteristics, e.g. one queue can be optimized for messages that require longer processing time compared to messages of other queues. Variable message size SQS allows its users to push messages of size up to 8KB. Unlimited queues and messages the number of queues and messages in the Amazon SQS system is unlimited.

### 2.8.2 Architectural Overview

Amazon SQS consists of three main components in the overall system; Figure 2.6 is a diagram depicts SQS architecture. The components of SQS are:

- **The components of your distributed system**: the system uses SQS has several components that send messages to the queue and receive messages from the queue.

- **Queues**: each queue is distributed among many servers, each server has an instance of the queue and a message pushed to the queue can appear in more than one instance of the queue.

- **Messages**: messages will be duplicated in more than one instance of the queue.

![Figure 2.6. SQS Architectural overview. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010.](image-url)
2.8.3 How Amazon SQS Works

SQS users will have to create an account prior to using the system. Each account is identified by an account number, e.g. “123456789012” and each account has a single namespace. The user can create as many Amazon SQS queues as desired by using CreateQueue, but a unique name within the account namespace must be given to each queue. SQS assigns each queue created an identifier called a queue URL, which includes the queue name and other components that SQS determines. Whenever you want to perform an action on a queue, you must provide its queue URL. http://queue.amazonaws.com/123456789012/queue2 is the queue URL for a queue named "queue2" owned by a person with the AWS account number "123456789012". SQS users can delete a queue at any time whether it is empty or not. The queue can be empty if either no messages were sent to it or all messages were processed and deleted.

After the queue is created, users can start putting message in their queues and retrieve them at a later time. Amazon SQS allows user’s application components to send a message using SendMessage request, SQS system will assign an identifier to each submitted message and it can be used to identify a message by the user, e.g. to delete it. SQS will duplicate the message to a subset of queue instances as shown in Figure 2.7.

![Figure 2.7. Put a message into a queue. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010.](image)

A different user system component that processes messages needs more messages to process, so it calls ReceiveMessage request, SQS will return the message from the queue as shown in Figure 2.8.
When a message is retrieved from the queue, SQS samples a subset of the servers (based on a weighted random distribution) and returns messages from just those servers. This means that a particular receive request might not return all your messages. Or, if you have a small number of messages in your queue (less than 1000); it means a particular request might not return any of your messages, whereas a subsequent request will. If you keep retrieving from your queues, SQS will sample all of the servers, and you will receive all of your messages. Figure 2.9 shows messages being returned (Message A, C, D, and B). Message E is not returned to this particular request, but it would be returned to a subsequent request.
Immediately after the component receives the message, the message is still in the queue. However, you don't want other components in the system receiving and processing the message again. Therefore, Amazon SQS blocks them with a visibility timeout, which is a period of time during which Amazon SQS prevents other consuming components from receiving and processing that message. Figure 2.10 shows a visibility chart.

![Visibility Timeout Chart](http://example.com/visibility.png)

**Figure 2.10.** Message visibility timeout. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010.

When user component processes messages successfully it then calls **DeleteMessage** request, which removes the message from the queue so no one else will ever process it as shown in Figure 2.11. If this system fails to process the message, than it will be read by another **ReceiveMessage** request as soon as the visibility timeout passes. SQS will not automatically delete the message and wait for user component to request deletion and that due to the distributed nature of components, there is no guarantee that the message is processed.

![Deleting Message](http://example.com/deleting.png)

**Figure 2.11.** Deleting the message “A” from the queue. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010.
SQS makes a best effort to preserve order in messages, but due to the distributed nature of the queue, we cannot guarantee you will receive messages in the exact order you sent them. If your system requires that order be preserved, you can place sequencing information in each message, so that you can reorder the messages when the queue returns them.

SQS stores copies of your messages on multiple servers for redundancy and high availability. On rare occasions, one of the servers storing a copy of a message might be unavailable when you receive or delete the message. If that occurs, the copy of the message will not be deleted on that unavailable server, and you might get that message copy again when you receive messages. Because of this, you must design your application to be idempotent (i.e., it must not be adversely affected if it processes the same message more than once).

2.8.4 Request and Responses

Amazon SQS sets specific rules of how users can make requests and the responses they should expect [12]. This section explains the various requests types can be made to SQS and the responses returned respectively.

2.8.4.1 QUERY REQUESTS

Amazon SQS supports Query requests for calling service actions. Query requests are simple HTTP or HTTPS requests, using the GET or POST method. Query requests must contain an Action parameter to indicate the action to be performed. The response is an XML document that conforms to a schema. You might use Query requests when a SOAP toolkit is not available for your platform, or when Query requests are easier to make than a heavier SOAP equivalent. Amazon SQS GET requests as URLs, which can be used directly in a browser. The URL consists of Endpoint, Action and Parameters. The following is an example GET request to send a message to an SQS queue:

```
http://queue.amazonaws.com/queue1
?Action=SendMessage
&MessageBody=Your%20Message%20Text
&AWSAccessKeyId=0GS7553JW74RRM612K02EXAMPLE
&Version=2008-01-01
&Expires=2008-02-10T12:00:00Z
```
In SQS, all parameters except `MessageBody` always have values that have no spaces. The value you provide for `MessageBody` in `SendMessage` displayed with the spaces encoded (as %20). For clarity, the rest of the URL is not displayed in a URL encoded format. The first line represents the `endpoint` of the request. This is the resource the request acts on. The preceding example acts on a queue, so the request's endpoint is the queue's identifier, known as the `queue URL`. The “?” mark after the endpoint separates the endpoint from the parameters. Each parameter is separated by an ampersand (&). The `Action` parameter indicates the action to perform. To create a POST request follow a few steps the first one is to assemble the query parameter names and values into a form. This means you put the parameters and values together like you would for a GET request (with an ampersand separating each name-value pair). The following example shows a `SendMessage` request with the line breaks used in this guide to make the information easier to read.

```
Action=SendMessage
&MessageBody=Your Message Text
&AWSAccessKeyId=0GS7553JW74RRM612K02EXAMPLE
&Version=2008-01-01
&Expires=2008-02-10T12:00:00Z
&Signature=lBP67vCvGlDMBQ1dofZxg8E8SUEXAMPLE
```

The second step is to form-URL-encode the form according to the `Form Submission` section of the HTML specification.

```
Action=SendMessage
&MessageBody=Your+Message+Text
&AWSAccessKeyId=0GS7553JW74RRM612K02EXAMPLE
&Version=2008-01-01
&Expires=2008-02-10T12%3A00%3A00Z
&Signature=lBP67vCvGlDMBQ1dofZxg8E8SUEXAMPLE
```

The third step is to provide the encoded form as the body of the POST request. Include the Content-Type HTTP header with the value set to `application/x-www-form-`
url-encoded. The following example shows the final POST request. The entire form has been form URL encoded.

```
POST /queue1 HTTP/1.1
Host: queue.amazonaws.com
Content-Type: application/x-www-form-urlencoded
Action=SendMessage
&MessageBody=Your+Message+Text
&AWSAccessKeyId=0GS7553JW74RRM612K02EXAMPLE
&Version=2008-01-01
&Expires=2008-02-10T12%3A00%3A00Z
&Signature=lBP67vCvGIDMBQ1dofZxg8E8SUEXAMPLE
```

2.8.4.2 SOAP REQUESTS

Amazon SQS supports the SOAP message protocol for calling service actions over an HTTP or HTTPS connection. The easiest way to use the SOAP interface with your application is to use a SOAP toolkit appropriate for your programming platform. SOAP toolkits are available for most popular programming languages and platforms.

The service's Web Services Description Language (WSDL) file describes the actions along with the format and data types of the actions' requests and responses. Your SOAP toolkit interprets the WSDL file to provide your application access to the actions. For most toolkits, your application calls a service action using routines and classes provided or generated by the toolkit.

A SOAP request is an XML data structure that your SOAP toolkit generates and sends to the service. SQS recognizes that the request is a SOAP request by the presence of the optional SOAPAction header. If no SOAPAction header appears in the request, then the content type of the first (or only) message part must be one of text/xml, text-xml-SOAP, application/soap+xml.

The following example shows the XML for a SOAP message that calls the CreateQueue action. Although you probably won't build the SOAP message for a service request manually, it is useful to see what your SOAP toolkit tries to produce when provided with the appropriate values. The CreateQueue element contains the operation-specific QueueName parameter.
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    <aws:AWSAccessKeyId>1D9FVRAYCP1VJS767E02EXAMPLE</aws:AWSAccessKeyId>
    <aws:Timestamp>2008-02-10T23:59:59Z</aws:Timestamp>
    <aws:Signature>SZf1CHmQ/nrZbsrC13hCZS061ywsEXAMPLE</aws:Signature>
  </soapenv:Header>
  <soapenv:Body>
    <CreateQueue xmlns="http://queue.amazonaws.com/doc/2008-01-01/">
      <QueueName>MyQueue</QueueName>
    </CreateQueue>
  </soapenv:Body>
</soapenv:Envelope>

2.8.4.3 RESPONSES

In response to an action request, SQS returns an XML data structure that contains the results of the request. This data conforms to the SQS schema. Other than the use of a message envelope in the case of SOAP, the schema for the results is the same for Query and SOAP responses.

If the request succeeded, the main response element is named after the action, but with "Response" appended. For example, CreateQueueResponse is the response element returned for a successful CreateQueue request. The XML schema describes the XML response message for each SQS action. The following is an example of a successful response.

<CreateQueueResponse>
  <CreateQueueResult>
    <QueueUrl>
      http://queue.amazonaws.com/queue2
    </QueueUrl>
  </CreateQueueResult>
  <ResponseMetadata>
    <RequestId>cb919c0a-9bce-4afe-9b48-9bdf2412bb67</RequestId>
  </ResponseMetadata>
</CreateQueueResponse>

If a request is unsuccessful, the main response element is called *ErrorResponse* regardless of the action that was called. This element contains an Error element and a *RequestId* element. Following is an example of error response:

```
<ErrorResponse>
  <Error>
    <Type>Sender</Type>
    <Code>InvalidParameterValue</Code>
    <Message>
      Value (quename_nonalpha) for parameter QueueName is invalid. Must be an alphanumeric String of 1 to 80 in length
    </Message>
  </Error>
  <RequestID>42d59b56-7407-4c4a-be0f-4c88daeea257</RequestID>
</ErrorResponse>
```

### 2.9 Challenges and Issues

Systems, applications, and web applications use queues in various ways to store messages produced by a component waiting to be processed by another component. The queue can have one or more producers and one or more consumers,. The queue can be local to the system or remote. The main advantage of the queue is that it decouples producers and consumers so that the message can sit in the queue based on the system requirement. The question remains: what happens to the messages if the queue crashes because of system failure? They are simply going to be lost. Also if the system holding the queue crashes or simply becomes unavailable, how wills that impact the producer and/or the consumer?
Distributing the queue is a solution implemented to solve both issues of availability of the queue and, more importantly, possible loss of messages when the system holding the queue crashes, in turn crashing the queue. With distributing queuing a message will be duplicated and saved in more than one queue, each of which is physically separated. If one physical location crashes, the other locations can still save and retrieve the messages. Distributing the queue is not a straightforward easy task and has many challenges and limitations, which this chapter will discuss.

2.10 CAP THEOREM

The CAP theorem [13], also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all of these three items: consistency, availability, and partition tolerance. The theorem began as a conjecture made by University of California, Berkeley computer scientist Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). In 2002, Seth Gilbert and Nancy Lynch of MIT [14] published a formal proof of Brewer's conjecture, establishing it as a theorem.

2.10.1 Consistency

A fully consistent queue is one that can guarantee that when a message is in a stored state it will report the same state for all consecutive operations until a request changes that state. When a request arrives to store a message in the queue, the message is stored in a state that must remain the same for any future request. Once a retrieve request arrives, the state of the message changes from stored to deliver [15]. Example 1 is a single instance queue ensures consistency since there is only one physical place to manage messages; once a message is stored, it will retain the same state until a retrieve request arrives. See Figure 2.12.

Figure 2.12. Single instance of queue server ensure consistency.
In **Example 2** a queue is distributed among two or more nodes and each node must have the same message as the others, then when a message is stored in the first node, the message is not in a consistent state until it is stored in all other nodes. See Figure 2.13.

![Figure 2.13. Queue distributed among two servers.](image)

**2.10.2 Availability**

The queue is available at any time that a producer tries to put a message in or a consumer tries to retrieve a message. The single queue in Example 1 above is not highly available. If the server that is holding the queue goes down, both producer and consumer will be unable to do the work. There is also the possibility of losing all of the messages already in the queue. The duplicate queue in Example 2 has better availability because if one node goes down the other nodes can still serve. The more nodes created, the higher the availability of the queue. High availability of a queue distributed among nodes does not imply 100% availability no matter how many nodes used where some edge case scenarios are still possible which makes the queue unavailable however increasing the number of nodes will increase the probability of an available queue.

**2.10.3 Partition Tolerance**

In Example 1 the queue exists in only one node, the queue works as one atomic process, assuming it is scalable and code perfect. It will either work or not depending on the state of the queue. In Example 2 the queue is distributed into two or more nodes, the system keeps operating for both saving and retrieving messages even if one node loses connectivity or crashes. In this case, the system or queue is partition tolerant.
2.10.4 CAP Theorem and SQS

CAP theorem tells us that it is possible to have two of consistency, availability and partition tolerance. It is impossible to achieve all three simultaneously. The distributed queue is based on having the queue in several nodes where the message is duplicated in all or some of those nodes. The network can go down on some nodes, or some nodes can crash. This implies that the queuing system must be partition tolerant, which leaves us with availability and consistency to choose from. Where availability of the queue is the main reason for having the queue distributed, we must give up consistency, which means that not all nodes have the same status for a message. Some nodes might have deleted the message while other nodes still think they need to deliver the message.

Amazon SQS gives up consistency in favor of high availability and partition tolerance, and introduces the concept of an eventually consistent queue. As we saw in this chapter, SQS creates a queue and distributes it among many SQS servers. When a message arrives to be inserted into the queue, SQS will sample a subset of servers based on weighted random distribution and saves the message into the servers selected only. When a subsequent retrieval request arrives, SQS samples a subset of servers again based on weighted random distribution and returns messages found in the chosen subset implying that the message retrieved might be in other server which were not chosen as part of the subset, in this case the servers are not consistent and there is a possibility of returning the same message again in subsequent requests. Figure 2.14 shows how queue servers are inconsistent where a message is not across all queue servers.

![Figure 2.14. One queue distributed among many servers.](http://aws.amazon.com/sqs/)

2.11 SQS Properties

Amazon SQS has few properties some of which show the advantage of using SQS while others shows the limitation, these properties are as follows:

- **SQS is highly available.** We can consider SQS highly available since losing one or more SQS servers will not impact the availability of the queue; other SQS servers will still be serving producer and consumer components. SQS remains available unless all SQS servers are lost at once.

- **SQS is partition tolerant.** The queue created using SQS servers is partition tolerant, since if connectivity is lost on one or more, but not all, SQS servers, the queuing system continues operating.

- **SQS is not consistent.** The distributed nature of the queue causes the messages saved to be inconsistent. A message is not saved on all SQS servers at the same time a message is saved in some SQS servers and the message is not consistent on all SQS servers.

- **SQS does not preserve order.** SQS makes a best effort to preserve order in messages, but due to the distributed nature of the queue, SQS cannot guarantee retrieval of messages in the exact order they were sent. A cure for this issue is for the user to place sequencing information in each message before pushing it to the queue.

- **SQS might deliver a message more than once.** SQS stores copies of messages on multiple servers for redundancy and high availability. On some occasions, one of the servers storing a copy of a message may be unavailable when a message is received or deleted. If that occurs, the copy of the message will not be deleted from the unavailable server and the undeleted copy of the message will be delivered again.

- **Retrieve message request might not return a message.** When you retrieve messages from the queue, SQS samples a subset of the servers (based on a weighted random distribution) and returns messages from just those servers. Since a message is not consistent on all SQS servers, a particular receive request might not return all requested messages. Or if you have a small number of messages in your queue (less than 1000), a particular request might not return any of your messages, whereas a subsequent request will. If you keep retrieving from the queues, SQS will sample all of the servers, and all of the messages will be returned. Figure 2.15 shows messages being returned after one of the system components makes a receive request. SQS samples several of the servers (in gray) and returns the messages from those servers (Message A, C, D, and B). Message E is not returned to this particular request, but it would be returned by a subsequent request.
Figure 2.15. Retrieving a message by sampling a subset of queue servers in gray and return the messages from the sampled servers only. Source: Amazon web services. Amazon Simple Queue Service (Amazon SQS), http://aws.amazon.com/sqs/, accessed October 2010.
CHAPTER 3

BUILDING QUEUING SYSTEM

In Chapter two we talked about how queues and queuing are used in computer applications and how they play a big role in today’s web applications and web services. This thesis project focuses on queuing systems characterized by a distributed queue where availability is the most important factor in building the system. Amazon SQS has built a system that creates queues that allow users to send and retrieve messages via a simple API. This thesis project is not an evaluation of SQS.

In this thesis project I build a queuing system that is derived from SQS and is based on CAP theorem, choosing two out of three from availability, consistency and partition tolerance; the main goal of building such a system is to study the challenges involved in building systems like Amazon SQS, as well as to study their limitations and ways to reduce these limitations. Amazon SQS is not an open source project so we don’t know how it was developed or implemented. Therefore, I will design and build my system, choosing the programming language, technologies and algorithms. The system I built is not suitable for commercial use, but it is adequate for academic research. Thus I am calling it Very Simple Queuing System (VSQS). Throughout this chapter I will point out the differences between VSQS and Amazon SQS.

3.1 VSQS AND CAP THEOREM

VSQS is a queuing system that ensures high availability to push and retrieve messages at any given time. VSQS follows the Amazon SQS example to achieve high availability by distributing the queue among two or more servers where messages are duplicated on each of the servers. Losing one of those servers will not affect the availability of the queue but it increase the risk of reaching to unavailability state especially when losing more servers. As a result, the lost server might not be consistent with other available servers once it comes back on line if messages are received, retrieved or deleted while it was down.
As discussed in Chapter Two, CAP theorem states that a distributed system cannot have all three characteristics of consistency, availability and partition tolerance. At most a distributed system can have two.

VSQS, like SQS, chooses to have availability and partition tolerance while sacrificing consistency. The design of VSQS takes into consideration an eventually consistent queue where the lost server, when it comes back on line, will be consistent with other on line servers. Later in this chapter we discuss how to achieve an eventually consistent distributed queue.

### 3.2 VSQS COMPONENTS AND DESIGN

VSQS is a multiple component system, each of which is identified based on responsibility and level of communication with other components within VSQS, as well as communication with users. Dividing the system into dependent components reduces coupling and allows installing different components in different physical boxes. It also gives the flexibility to change each component’s design without the necessity of changing the others. The three components are listed below. Figure

- Queue Server
- Queue Server Manager
- Queue System Manager

#### 3.2.1 Queue Server (QS)

The queue server (QS) is the component responsible for managing queues and messages. It can create queues and keep messages safe and available for retrieval. The queue server knows how to create the queue as a single entity and not as a distributed entity. In this component we can decide how to create queues and where the messages of the queue reside, i.e. is it up to the queue server to decide whether to use DB or in-memory DB to accomplish creating the queue and saving the messages. The rest of VSQS system does not know how the queue server is actually creating the queues and how messages are saved. The queue server can freely change its way of managing the queue without having any impact on the rest of VSQS. It provides a protocol for communication used by the queue server manager to direct queuing requests such as “create queue” or “put message”. The queue server is designed to be accessed only by other VSQS
component, this component called “Queue System Manager QSM” as we are going to see later in this chapter. The queue server is hidden from the users of the system, users will not be given a direct access to queue server and that because queues created by users are actually distributed among multiple queue servers and a message will be duplicated into all queue server instances. A queue server will not have the privilege of making requests to another instance of queue server to limit managing of the queue servers and all queues they hold by QSM. Queue servers can manage multiple queues that might belong to different users. Figure 3.1 shows one instance of a queue server that manages three different queues with names E, C and D.

![Figure 3.1. Queue server instance holding three queues.](image)

Each queue server is part of the distributed entity that makes up the pre-selected subset of servers used to create the distributed queue, each of which holds its own version of the queues and messages. Each instance is unaware of other instances within the same subset. Actions received on a queue server will be performed on the queue instance belonging to it only. It is up to the component that manages the subset to communicate the request to the rest of the instances. Figure 3.2 shows a subset of queue servers where each one contains queue A.

![Figure 3.2. A subset consists of five queue servers.](image)
3.2.1.1 DATA MANAGEMENT

The queue server uses data storage to manage queues and store messages. It is designed so that it can connect to any RDBM or in-memory DB and is configured to turn message caching on or off. The question remains: what is the best choice to store messages? Amazon SQS uses its own database called “SimpleDB” for which we have very limited information on what this DB, is or how it works.

To survive system failure the messages must be persistent. In-Memory DBs are not sufficient to achieve this goal, so we must have our messages saved using an RDBM like MySQL. Using a persistent RDBM has the drawback of extra latency expense of reaching out to the database for every single request, especially for retrieval requests where messages need to be retrieved and returned to users as soon as possible. RDBMs are safe but not efficient enough for our system. If we need to retrieve messages fast, then we must have them in memory. Most of the popular RDMBs, including MySQL, offer an in-memory capability where a portion of the database is actually cached into memory and the rest stays on disk. This might work nicely if we had one table for each queue, which is not our solution. If we have one table containing messages for all queues for all users, then we are not sure which messages of which queue(s) are or are not in memory. Some messages will be retrieved faster than others. The first decision on database management is to create one table called “queue” to hold information of all queues. The schema is shown in Table 3.1.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>queueId</td>
<td>String PK</td>
<td>Unique queue identification, the value is ($userName_$queueName).</td>
</tr>
<tr>
<td>subsetId</td>
<td>String</td>
<td>The subsetId which the queue belong to.</td>
</tr>
<tr>
<td>queueName</td>
<td>String</td>
<td>The nameId which the queue the user chose.</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>The user name owning this queue.</td>
</tr>
<tr>
<td>createDate</td>
<td>Date time</td>
<td>The date this queue was created.</td>
</tr>
<tr>
<td>invisibleInSecond</td>
<td>Int</td>
<td>The number of seconds the message stays invisible after it is retrieved.</td>
</tr>
</tbody>
</table>
The table “queue” has a java representation model that looks like:

```java
public class Queue implements Serializable {
    private static final long serialVersionUID = -7804143486229412043L;
    private String queueId;
    private String queueName;
    private String userName;
    private Date createDate;
    private int invisibleInSeconds;
}
```

Also, “QueueDao” interface is provided to define all necessary data manipulation methods which will need to be implemented to match the data management mechanism chosen. Currently Hibernate [16] is used to assist in managing queries of the table. Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions. We can add more methods to this interface as needed, we then also need to add implementation of the new added methods to the class implementing this interface. The interface looks like:

```java
public interface QueueDao {
    public void save(Queue queue);
    public void saveOrUpdate(Queue queue);
    public void delete(Queue queue);
    public void save(String queueName, String userName, Date createDate);
    public void delete(String queueName, String userName);
    public Queue get(String queueId);
    public List<Queue> getAll();
    public List<Queue> getAllQueuesForUser(String userName);
    public void saveAll(List queues);
    public void saveOrUpdateAll(List queue);
    public void deleteAll(List queue);
}
```

The next decision is to create only one table called “message” to hold all messages of all queues. The schema of the table is shown in Table 3.2.
Table 3.2. Schema Description of Database Table Named “message”

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageId</td>
<td>String PK</td>
<td>The unique message identifier created by QSM and passed to queue server.</td>
</tr>
<tr>
<td>message</td>
<td>String</td>
<td>Limited to 255 char long.</td>
</tr>
<tr>
<td>Sequence</td>
<td>String</td>
<td>String representation of message sequence.</td>
</tr>
<tr>
<td>Status</td>
<td>String</td>
<td>It can be Available, In-Memory or Invisible.</td>
</tr>
<tr>
<td>InvisibleUntil</td>
<td>dateTime</td>
<td>Datetime the message invisible state will expire.</td>
</tr>
<tr>
<td>createDate</td>
<td>Date time</td>
<td>The date the message was added to the queue.</td>
</tr>
<tr>
<td>queueId</td>
<td>String FK</td>
<td>Foreign key links to queues.queueId primary key</td>
</tr>
</tbody>
</table>

The table “message” has a java representation model that looks like:

```java
public class Message implements Serializable{
    private static final long serialVersionUID = -7872407785812459641L;
    private String messageId;
    private String message;
    private String sequence;
    private String queueId;
    private Date createDate;
    private Date invisibleUntil;
    private String status; //Available, InMemory, Invisible
}
```

Similarly, message model has a DAO interface called “MessageDao” that looks like:

```java
public interface MessageQueueDao {
    public void save(Message message);
    public void update(Message message);
    public void delete(Message message);
    public void deleteById(String messageId);
    public Message getById(String id);
    public void deleteAllByQueueId(String queueId);
    public Message lockAndGetById(String id);
    public Message lockAndGetByDate();
    public boolean lockMessage(String messageId, Date invisibleUntil);
    public void makeInvisible(String id, Date invisibleUntil);
    public void makeAvailable(String messageId);
    public List<Message> get(String queueId, int count);
}
```
Programmatically create one cache per queue and store a limited number of messages in memory. Retrieve requests will first look into the cache before it attempts querying the database. This methodology will assist in fast retrieval of messages and reduces the number of hits on the database to get messages. Loading messages into memory is done by calling “\texttt{loadIntoMemory()}” method of class “\texttt{com.firas.thesis.queueserver.QueueManager}”. This method can be called within the interval of a scheduled job that monitors the number of messages available in memory or it can be called when the message count reaches a particular threshold. Below is the code of this method.

\begin{verbatim}
protected void loadIntoMemory()
{
    //ensure only one call to this method at anytime
    if(this.loadIntoMemoryCounter.tryAcquire(1)) {
        try {
            //the cache has a limit of the number of messages it can hold
            int capacity = this.messageIdsQueue.remainingCapacity();

            //get a limited number of message from db
            List<Message> messages =
                    this.getMessageQueueDaoManager().get(this.getQueueId(), capacity);

            //load the message if any into memory
            Iterator<Message> iter = messages.iterator();

            while(iter.hasNext()) {
                Message message = iter.next();
                messageIdsQueue.add(message.getId());
                messageMap.put(message.getId(), message);
            }
        } catch(Exception e) {
            log.info("unexpected error", e);
        } finally {
            this.loadIntoMemoryCounter.release(1);
        }
    }
}
\end{verbatim}

To use hibernate, each java model has a corresponding “*.hbm.xml” hibernate data mapping xml file that describe the table, hibernate will use these two files to map java models to database tables. Also, two data object managers are provided
“QueueDaoManager” and “MessageQueueDaoManager” to manage “QueueDao” and “MessageDao” respectively. Figure 3.3 shows DAO code hierarchy.

```java
com.fras.thesis.queueserver.dao
  \- J MessageQueueDao.java
  \- J MessageQueueDaoHibernate.java
  \- J MessageQueueDaoJDBC.java
  \- J QueueDao.java
  \- J QueueDaoHibernate.java
com.fras.thesis.queueserver.dao.model
  \- J Message.java
  \- J Queue.java
com.fras.thesis.queueserver.daomanagers
  \- J MessageQueueDaoManager.java
  \- J QueueDaoManager.java
com.fras.thesis.queueserver.tasks
service/src/main/resources
com.fras.thesis.queueserver.dao.model
  X Message.hbm.xml
  X Queue.hbm.xml
```

Figure 3.3. Class hierarchy of queue server data access object management.

### 3.2.1.2 Communication Protocol – Web Service API

Queue server provides a web service API as a mean of communication available mainly to queue server manager (QSM) to send requests. To send a request, the sender needs to package a web service method and a queue server URL along with other information needed to process each API request, the general web service parameters packaging looks like:

```java
WebServiceClient.Params params = new WebServiceClient.Params();
params.setAppId("subsetmanager");

// set the method and url you want to call
params.setMethod($WS API METHOD);
params.setURL($QUEUE SERVER URL);

// add all parameters needed
params.addParameter("param1", value);
params.addParameter("param2", value);
...```
//create the client and make the call
WebServiceClient client = new WebServiceClient();
String response = client.getResponseBodyAsString(params);

Queue server provides a separate web service call for each function it supports.
Figure 3.4 shows class “WebService.java” that defines all these different calls followed
by a description of all these calls, how to package the parameters and the xml responses
the caller should expect.

```
package com.firas.thesis.queuingsystem.ws;

class WebService {
    ...
    ...
    public void createQueueServlet(HttpServletRequest request, HttpServletResponse response) {
        // implementation...
    }
    ...

    ...}
```

Create Queue Request is called to create a queue for the given user name with
given queue name.

```
params.setAppId("subsetmanager");
params.setMethod(com.firas.thesis.queuingsystem.ws.createQueue);  
params.setUrl("$QUEUE SERVER URL");
params.setParameter("userName", value);
params.setParameter("queueName", value);
params.setParameter("invisibleInSeconds", value);
```

Delete Queue Request is called to delete the given queue name for the given user
name.

```
params.setAppId("subsetmanager");
params.setMethod(com.firas.thesis.queuingsystem.ws.deleteQueue);
params.setUrl("$QUEUE SERVER URL");
params.setParameter("userName", value);
```
params.setParameter("queueName", value);

Put Message Request is called to add a message to the user’s queue. The parameters look like:

params.setAppId("subsetmanager");
params.setMethod com.firas.thesis.queuingsystem.ws.putMessage);
params.setURL($QUEUE SERVER URL);
params.setParameter("userName", value);
params.setParameter("queueName", value);
params.setParameter("message", value);
params.setParameter("messageId", value);

Get Message Request is called to retrieve a message from a queue. The parameters look like:

params.setAppId("subsetmanager");
params.setMethod com.firas.thesis.queuingsystem.ws.getMessage);
params.setURL($QUEUE SERVER URL);
params.setParameter("userName", value);
params.setParameter("queueName", value);

Lock Message Request is called to put the message into an invisible mode. The parameters look like:

params.setAppId("subsetmanager");
params.setMethod com.firas.thesis.queuingsystem.ws.lockMessage);
params.setURL($QUEUE SERVER URL);
params.setParameter("userName", value);
params.setParameter("queueName", value);
params.setParameter("messageId", value);

Unlock Message Request is called to put the message into available mode. The parameters look like:

params.setAppId("subsetmanager");
params.setMethod com.firas.thesis.queuingsystem.ws.unlockMessage);
params.setURL($QUEUE SERVER URL);
params.setParameter("userName", value);
params.setParameter("queueName", value);
params.setParameter("messageId", value);
Delete Message Request is called to delete the message from the queue. The parameters look like:

```java
params.setAppId("subsetmanager");
params.setMethod com.firas.thesis.queuingsystem.ws.deleteMessage);
params.setURL($QUEUE SERVER URL);
params.addParameter("userName", value);
params.addParameter("queueName", value);
params.addParameter("messageId", value);
```

The responses for all requests are going to be an xml document. Following are failure response for all requests, successful response for all requests and successful response for `get_message` requests.

```xml
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<response status="error">
  <message>${message}</message>
</response>

<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<response status="ok">
</response>

<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<response status="ok">
  <#if message?exists>
    <messageId>${message.id}</messageId>
    <message>${message.message}</message>
    <messageSequence>${sequence}</messageSequence>
    <createDate>${message.createDate}</createDate>
  </#if>
</response>
```

### 3.2.2 Queue Server Manager (QSM)

The **queue server manager (QSM)** works as the engine of VSQS that manages user requests and routes them to the designated queue servers, as well as managing the subsets. It is the component where most of the decisions are made, and has full control of all pieces of the system, as we see in this section. All future enhancements and additions might all be added to this component.
3.2.2.1 MANAGING SUBSETS OF QUEUE SERVERS

Queue servers, as we have seen in this chapter, work as individual units. Even though VSQS consists of many queue servers, each queue server does not know about the other queue servers in the system. A subset consists of a few queue servers, each of which within the same subset still does not know of the others. When a queue server receives a request, it sends the response back based on the information it has. It will not look into any of the other queue servers, i.e. it doesn’t understand what the subset is and does not even know it is a part of a subset. QSM is the component that knows about all queue servers within a subset and about all subsets within the system.

QSM, to perform the requests, needs to know how to communicate with each queue server in the subset via its protocol and that by using Web Service client it will package all parameters needed for each request as described in the Queue Server Web Service API section. More importantly, QSM needs to know the URL for each queue server within the subset so that it can distribute the requests to all of them. Figure 3.5 depicts how QSM communicates with all queue servers in the subset.

Figure 3.5. QSM calls all queue servers.
Queue manager server (QSM), is the server responsible for performing the following tasks:

- Managing multiple subsets.
- Choosing only one subset when creating a queue. All consecutive requests for that queue go to the subset chosen at creation time.
- Creating or deleting user queue requests needing to be sent to all queue servers in the subset.
- Pushing a message to the queue, meaning sending it to all queue servers in the subset.
- Deleting a message from the queue, meaning deleting it from all queue servers in the subset.
- Retrieving a message from the queue, resulting in putting the message in an invisible state on all queue servers.
- Managing simultaneous requests to retrieve a message from a queue, no message is returned to multiple requests.

QSM knows about all queue servers, how they are divided into subsets, the location of each queue server within a subset, and how many queues each subset holds at any given time. This information is retained in central database tables used by QSM to determine the subset and location of its queue servers, as well as the number of queues. The first table is called **subset**. This table holds information about all subsets QSM manages. Table 3.3 shows the schema description of this table.

**Table 3.3. Schema Description of Database Table “subset”**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubsetId</td>
<td>String (PK)</td>
<td>Uniquely identifies the subset.</td>
</tr>
<tr>
<td>serversInSubset</td>
<td>Integer</td>
<td>Number of queue server in the subset.</td>
</tr>
<tr>
<td>numberOfQueues</td>
<td>Integer</td>
<td>Number of queues registered in this subset.</td>
</tr>
<tr>
<td>invisibleInSeconds</td>
<td>Long</td>
<td>Defines the invisible period of a retrieved message.</td>
</tr>
<tr>
<td>createDate</td>
<td>Date time</td>
<td>Date the subset was created.</td>
</tr>
</tbody>
</table>

The second table is called **subsetLocation**. This table contains the locations of all queue servers. Many rows may belong to one **subsetId**. Table 3.4 shows schema description of this table. This table has a one-to-one relationship with subset table and linked using “**subsetId**” field.
Table 3.4. Schema Description of Database Table “subsetlocation”

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>String pk</td>
<td>url points to a queue server.</td>
</tr>
<tr>
<td>subsetId</td>
<td>String FK</td>
<td>Links to SUBSET.SUBSETID.</td>
</tr>
</tbody>
</table>

The third table is **called queues**. This table contains the definition of all queues created using this QSM; the queue will reside in one subset managed by this QSM. Table 3.5 shows schema description of this table. This table has one-to-one relationship with table subset linked by “subsetId” field.

Table 3.5. Schema Description of Database Table “queues”

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>queueId</td>
<td>String pk</td>
<td>Unique queue identification, the value is ($userName_$queueName).</td>
</tr>
<tr>
<td>susbetId</td>
<td>String FK</td>
<td>Links to SUBSET.SUBSETID.</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>The user name owns this queue.</td>
</tr>
<tr>
<td>queueName</td>
<td>String</td>
<td>The name given to this queue by the user.</td>
</tr>
<tr>
<td>messageSequence</td>
<td>Long</td>
<td>Keeps track of message sequence of this queue.</td>
</tr>
<tr>
<td>invisibleInSecond</td>
<td>Int</td>
<td>Number of seconds the message be invisible.</td>
</tr>
<tr>
<td>createDate</td>
<td>DateTime</td>
<td>The date the queue was created.</td>
</tr>
</tbody>
</table>

QSM depends on the subset data stored in the above tables to determine the number of subsets it manages, the number of queue servers included in each subset and their location as well as the queues created in each of the subset. QSM uses these data to create a separate “SubsetManager” object for each subset which makes it easy for QSM to manage multiple subsets without conflicts.

Similarly, these tables are managed by Hibernate; each table has a java model representation, Data Access Object (DAO) interface and DAO manager as well as “hbm.xml” file. Figure 3.6 shows the class definition hierarchy of database access object management for queue server manager.
The "SubsetManager" contains all information it needs about the subset like the number of queue servers in the subset, their locations and the queues created by users the subset has; it knows how to process requests and how to forward them to all queue servers. "SubsetManagerFactory" object will assist in creating subset managers as needed by providing the below method:

```java
public SubsetManager createSubsetManager(String subsetId){
    return new SubsetManager(subsetId,
        this.getQueuesDaoManager(),
        this.getSubsetLocationDaoManager(),
        this.getQueueServerLocationFactory);
}
```

Subset manager object keeps track of all queues and the users who create those queues by putting the configuration data found in "queues" table into a "HashMap", subset manager will add to it when a new queue gets created and remove any queue when receiving a delete request, this hash map is kept in sync with "queues" table at all times. Keeping this info in memory will reduce the number of hits to the database trying to lookup the existence of a queue.
public void prepareAvailableQueues()
{
    if(queues == null)
    {
        queues = new HashMap<String, Queues>();
        List<Queues> allqueues =
            this.getQueuesDaoManager().getAllQueuesForSubset(
                this.getSubsetId());
        for(Queues queue : allqueues){
            this.queues.put(queue.getUserName() +
                queue.getQueueName(), queue);
        }
    }
}

Subset manager keeps track of the locations of all queue servers by putting the configuration data found in "subsetLocation" table into a list, by keeping this info in memory will make it easy to forward request to all queue servers without hitting the database for every request.

public void prepareQueueSeverLocation() {
    if(qsLocations == null)
    {
        qsLocations = new LinkedList<QueueServerLocation>();
        List<SubsetLocation> locations =
            this.getSubsetLocationDaoManager().getAllLocationsForSubset(this.getSubsetId());
        for(SubsetLocation location : locations){
            qsLocations.add(this.getQueueServerLocationFactory().
                getQSLocation(location.getUrl()));
        }
    }
}

QSM monitors all queue servers it manages and it is responsible of knowing when a queue server becomes unavailable and stop sending requests to it, more importantly it should know when the queue server comes back on line and decide when it become part of the available servers. when a queue server goes down it loses all the requests QSM receives, when queue server comes back on line it will be out of sync with the rest of queue servers in the subset, QSM is responsible on sending all those requests it missed, QSM will mark it as available when it is in sync with the rest of queue servers within the subset.
When the system is deployed QSM will check on all queue serves it manages and ensure all of them are available, mark those that are unavailable. When a queue server marked as available fails to handle any of the requests it will be marked as unavailable. Each queue server has three states:

- **Available**: the queue server is online and is available to receive request.
- **Unavailable**: the queue server is not online, it is not available to receive requests
- **Synchronizing**: the queue severer is online, it is available only to receive old requests, and all new requests will stay in the queue until older requests are handled. Once the queue server received all missed requests it then becomes available to receive new requests.

For each unavailable queue server a monitoring job is created that runs periodically checking if the queue server came back on line. All missed new requests will be saved into a place holder which is a database table named “requestplaceholder”; Table 3.6 shows the schema of this database table, the table hold all the information needed for re-processing.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String PK</td>
<td>Uniquely identified a message</td>
</tr>
<tr>
<td>url</td>
<td>String</td>
<td>Queue server url.</td>
</tr>
<tr>
<td>queuetId</td>
<td>String</td>
<td>Queue id this request belong to.</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>User name this request belong to.</td>
</tr>
<tr>
<td>queueName</td>
<td>String</td>
<td>Queue name this request belong to.</td>
</tr>
<tr>
<td>Command</td>
<td>String</td>
<td>The request command</td>
</tr>
<tr>
<td>messageId</td>
<td>String</td>
<td>Id assigned to the message</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>The message received in put request</td>
</tr>
<tr>
<td>messageSequence</td>
<td>Long</td>
<td>Keeps track of message sequence of this queue</td>
</tr>
<tr>
<td>invisibleUntil</td>
<td>long</td>
<td>Milliseconds until the message becomes invisible</td>
</tr>
<tr>
<td>invisibleInSeconds</td>
<td>Int</td>
<td>Number of seconds the message be invisible</td>
</tr>
<tr>
<td>createDate</td>
<td>DateTime</td>
<td>Datetime, this record is created</td>
</tr>
</tbody>
</table>
Once the monitoring jobs sees that the queue server is back on line, it will then sends a signal to start synchronizing by calling “startRsyncing” method on class “QueueServerLocation”, the method will retrieve all requests saved in the place holder table and send it to the queue server.

3.2.2.2 QSM AND USER REQUESTS

The other side of QSM is to manage user requests. Registered users, after creating queues, will be given a URL to send in all commands available to users to manage their queues. The URL is actually points to the QSM responsible for managing the subset of queue servers chosen to hold the newly created queue for that particular user. The user will be provided with a client API, which can be used by the user application for both producer and consumer components to facilitate calling VSQS. In this case, it is actually calling the QSM component of VSQS. In this section we see commands available for users and note which is for producers and which are for consumers.

QSM should be capable of receiving simultaneous calls from all users that have queues in any of the subsets managed by QSM. QSM can differentiate the calls and knows which subset this call is targeting. When making calls, users send in the “subsetId” parameter as we are going to see in the Web Service API section. QSM will depend on this information to process the request.

As we saw in the previous section QSM internally creates a “SubsetManager” object for each subset it manages, the “SubsetManager” object responsible on managing the subset and all queue servers belong to it. QSM might manage multiple subsets therefore it might have multiple subset managers. “SubsetFinder” object makes it easy to track all the different subset managers and create a new one when needed; “SubsetFinder” provides “findSubsetManager” method that returns an instance of subset manager, below is the code snippet of this method.

```java
public SubsetManager findSubsetManager(String subsetId){
    //if already created then return it
    if(subsetManagers.containsKey(subsetId))
        return subsetManagers.get(subsetId);
    //look up the subset table
    Subset subset = this.getSubsetDaoManager().getById(subsetId);
    //don’t create a new subset manager if we can’t find the info in db
    if(subset == null)
```
return null;
//create an instance of subset manager, add to subsetManagers map
SubsetManager subsetManager =
this.getSubsetManagerFactory().createSubsetManager(subsetId);
subsetManagers.put(subsetId, subsetManager);
return subsetManager;
}

QSM should be capable of handling all the different commands for each user and be capable of handling simultaneous calls of the same command for a user. Users provide “command” parameters for each request. QSM will rely on the value of the command to hand the request off to the right processor.

QSM uses command patterns to accomplish handling the commands. All different command objects must extend a “Command” abstract class and implement a “process” abstract method:

```java
public abstract class Command {
    private SubsetManager subsetManager;

    public Command(SubsetManager subsetManager) {
        this.setSubsetManager(subsetManager);
    }

    public abstract void process(UserRequest userRequest);

    public void setSubsetManager(SubsetManager subsetManager) {
        this.subsetManager = subsetManager;
    }

    public SubsetManager getSubsetManager() {
        return subsetManager;
    }
}
```

When the `SubsetManager` object is initialized, it will create all command objects available and puts them into hash map. When QSM receives an add message command, for example, it will retrieve the `PutMessageCommand` object from the commands registry and call `process` method on it. The code snippet below shows the creation of commands and putting parameters:

```java
this.commands = new HashMap<String, Command> ();
this.commands.put(UserCommands.CREATE_QUEUE.name(), new CreateQueueCommand(this));
this.commands.put(UserCommands.DELETE_QUEUE.name(), new
```
DeleteQueueCommand(this));
this.commands.put(UserCommands.ADD_MESSAGE.name(), new
AddMessageCommand(this));
this.commands.put(UserCommands.DELETE_MESSAGE.name(), new
DeleteMessageCommand(this));
this.commands.put(UserCommands.GET_MESSAGE.name(), new
GetMessageCommand(this));

User’s requests will be forwarded to all queue servers in the subset that contains
the queue the requests are targeting. Subset manager object provide
“broadcastMessageViaWS” method for all commands to use, this method when it’s
called will distribute the request to all queue servers found in “qslocation” list by creating
a “SendRequestToQSRunnable” for each queue server location and pass it to an
executor:

```java
/**
 * @param userRequest
 * this method will create a runnable for each queue server
 * Location and pass it to the executor
 */
protected void broadcastMessageViaWS(UserRequest userRequest){
    for(QueueServerLocation location : qsLocations){
        SendRequestToQSRunnable runnable =
            new SendRequestToQSRunnable(location, userRequest);
        this.executor.execute(runnable);
    }
}
```

QSM should be capable of handling load of requests coming from both producer
and consumer components of the user application. When receiving a request, it will be
handed off to an Executor object which has a few worker threads that are waiting for new
requests. If there are more requests coming in than available threads, then the request will
wait until a worker is free. QSM should provide a Web Service API to users to support all
four commands available to users to manage their queues: PutMessage, GetMessage,
DeleteMessage and Delete queue.
3.2.2.3 QSM Web Service API

The QSM Web Service API is available to users who have already created a queue for the purpose of managing their queues. Users will receive a URL when creating a queue. The user must use this URL to manage each queue, as other queues for the same user might have different URLs. Using http, client users must send the required parameters for each of the four available requests; the first request is called **PutMessage Request** which is used to add a message to the queue. The following is an example of how to add all parameters required submitting put message request:

```java
params.setURL(url provided back to users when creating the queue);
params.addParameter("username", value);
params.addParameter("queuename", value);
params.addParameter("subsetId", value);
params.addParameter("message", value);
params.addParameter("command", "PUT_MESSAGE");
```

After making the request, the user gets a response; the response can be either a failure or successful response. The successful response should looks like:

```xml
<response>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>PUT_MESSAGE</command>
  <messageId>$messageId</messageId>
  <message></message>
  <messageSequence></messageSequence>
  <invisibleUntil></invisibleUntil>
  <status>success</status>
  <statusMessage>message added</statusMessage>
</response>
```

The second request is called **GetMessage Request** which is used to retrieve a message from the queue. Following are the parameters required which are similar to those parameters used in PutMessage request however the command value changes and there is no need to parameter “message”, VSQS ignores “message” parameter if it was sent by the user, GetMessage parameters look like:

```java
params.setURL(url provided back to users when creating the queue);
params.addParameter("username", value);
params.addParameter("queuename", value);
params.addParameter("subsetId", value);
```
params.addParameter("command", "GET_MESSAGE");

The response of a successful request a user gets has a message and a date that the messages stays invisible until it is deleted. The response looks like:

```xml
<request>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>GET_MESSAGE</command>
  <messageId>$messageId</messageId>
  <message>$message</message>
  <messageSequence>$Sequence number</messageSequence>
  <invisibleUntil>$date</invisibleUntil>
  <status>success</status>
  <statusMessage>message retrieved</statusMessage>
</request>
```

The third request is called **DeleteMessage Request** which is used to delete a message from the queue.

```java
params.setURL(url provided back to users when creating the queue);
params.addParameter("username", value);
params.addParameter("queueName", value);
params.addParameter("subsetId", value);
params.addParameter("messageId", message Id provided when calling get message);
params.addParameter("command", "DELETE_MESSAGE");
```

The response of a successful request a user gets has a messageId of the deleted message. The response looks like:

```xml
<request>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>DELETE_MESSAGE</command>
  <messageId>$messageId</messageId>
  <message></message>
  <messageSequence></messageSequence>
  <invisibleUntil></invisibleUntil>
  <status>success</status>
  <statusMessage>message deleted</statusMessage>
</request>
```
The fourth request is called **DeleteQueue Request** which is used to delete the queue. The queue is deleted even if it has messages.

```java
params.setURL( url provided back to users when creating the queue);
params.addParameter("username", value);
params.addParameter("queuename", value);
params.addParameter("subsetId", value);
params.addParameter("command", "DELETE_QUEUE");
The user gets a response for a successful request like:

```xml
<request>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>DELETE_QUEUE</command>
  <messageId> </messageId>
  <message></message>
  <messageSequence></messageSequence>
  <invisibleUntil></invisibleUntil>
  <status>success</status>
  <statusMessage>queue deleted</statusMessage>
</request>
```

NOTE: If an error occurs in any of the above requests, the response will look like:

```xml
<request>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>$Command</command>
  <messageId> </messageId>
  <message></message>
  <messageSequence></messageSequence>
  <invisibleUntil></invisibleUntil>
  <status>error</status>
  <statusMessage>$erromessage</statusMessage>
</request>
```

Some web service APIs can be used by other components of the VSQS. In this case, the system manager will make a web service call to the queue server manager to create queue. The parameters look like:

```java
params.setURL( QSM url);
params.addParameter("username", value);
params.addParameter("queuename", value);
params.addParameter("command", "CREAE_QUEUE");
```
The successful response looks like:

```xml
<request>
  <userName>$userName</userName>
  <queueName>$queueName</queueName>
  <command>CREAE_QUEUE</command>
  <messageId></messageId>
  <message></message>
  <messageSequence></messageSequence>
  <invisibleUntil></invisibleUntil>
  <status>success</status>
  <statusMessage>queue created</statusMessage>
</request>
```

### 3.2.3 System Manager (SM)

The last component of VSQS is called system manager. SM manages users and user relations like billing, which is beyond the scope of this thesis project and is not included in our discussion or testing. No attempt to build this part of the system is made; instead we assume that users are already registered and paying their dues. The system manager is the part of the system where registered users can request creating a queue, be provided with a URL to be used to manage the queue they just created. This URL basically points to the QSM. To perform this task, SM must have enough information about QSM(s) and the subset each of them manages, which what I am going to discuss next.

You can think of VSQS as a cloud of queue servers divided into subsets, where each subset is managed by only one QSM, each of which can manage multiple subsets. This cloud is managed by a system manager.

System manager manages VSQS and keeps adequate information about all queue servers, subsets and queue server managers in a central database. This information is summarized in three tables; the first table is called QSM. This table contains information about every single Queue Server Manager available in VSQS at any given time. Table 3.7 shows the schema description of “QSM” table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsmId</td>
<td>Integer PK</td>
<td></td>
</tr>
<tr>
<td>url</td>
<td>String</td>
<td>url points to the QSM server</td>
</tr>
</tbody>
</table>
The second table is called **SUBSET**. This table contains information about every subset of queue servers available in VSQS. Each entry links to an entry on the QSM table. Table 3.8 shows the schema description of “**subset**” table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubsetId</td>
<td>String (PK)</td>
<td>Identifies the subset</td>
</tr>
<tr>
<td>QsmId</td>
<td>Integer (FK)</td>
<td>Links to QSM.QSMID</td>
</tr>
<tr>
<td>serversInSubset</td>
<td>Integer</td>
<td>Number of queue server in the subset</td>
</tr>
<tr>
<td>numberOfQueues</td>
<td>Integer</td>
<td>Number of queues registered in this subset</td>
</tr>
<tr>
<td>invisibleInSeconds</td>
<td>Long</td>
<td>Defines the invisible period of a retrieved message</td>
</tr>
</tbody>
</table>

The third table is called **QS**. This table contains an entry for every queue server available in VSQS. Each entry may link to an entry in the subset table. Table 3.9 shows schema description of “**qs**” table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Integer</td>
<td>PK</td>
</tr>
<tr>
<td>subsetId</td>
<td>String</td>
<td>FK, Links to subset.subsetId</td>
</tr>
<tr>
<td>url</td>
<td>String</td>
<td>url points to the queue server</td>
</tr>
</tbody>
</table>

### 3.3 PROGRAMMING LANGUAGE AND TECHNOLOGIES

VSQS is built using the Java programming language. All web services API for all its components are based on **spring framework**, packaged as a war file and deployed into a **JBoss** application server. **MYSQL** is the **DBMS** used to create the database schema, using **hibernate** technology to access the database. Java Concurrent package is used extensively for thread management. **JUnit** is used for testing all components code while **Jmeter** is used for functional testing.
3.4 VSQS Installation

VSQS can be installed in Windows or UNIX machines, these machines must be equipped with Java run time environment version 1.5 or newer, Jboss version 4.0.3SP1 or newer and MYSQL version 5.1 or newer. It is recommended to set up JAVA_HOME environment variable to point to the installation of java, also recommended to set up JBOSS_HOME environment variable to point to jboss installation. Each queue server needs to be installed in a separate machine, also recommended to have queue server manager to be installed in a separate machine. Figure 3.7 shows the structure of VSQS system and general communication flow between user’s applications and the various component of VSQS system.

![Figure 3.7. VSQS System infrastructure.](image-url)
As a distributed system each queue server is installed in a separate box, the installation is to put the war file generated of building queue server project into the deploy folder of your JBOSS server \%JBOSS_HOME\%server\queuingsystem-deploy\qsws.war. Running queue server requires a properties file that has database info plus the in-memory cache size

```
qs.db.host=localhost:3306
qs.db.dbname=qs
qs.db.user=root
qs.db.password=root
memory.queue.size=2000
```

QSM can be installed preferably in a separate box, you can also install it in any of the other boxes you have your queue server installed. Similarly, you need to put the war file generated of building queue server manager project into the deploy folder of JBOSS server directory. Running queue server manager requires a properties file that has database info.

```
qsm.db.host=localhost:3306
qsm.db.dbname=qsm
qsm.db.user=root
qsm.db.password=root
```

### 3.5 VSQS Architectural View

This section explains the architecture of VSQS system and how queues are distributed into multiple servers. Also talks about the life cycle of queues and messages.

#### 3.5.1 Queue as Distributed Entity

Amazon SQS distributes their queues among many servers. When the system receives a request to push or retrieve a message, SQS takes a subset of all available servers based on weighted random distribution and performs the request. SQS documentation says nothing about choosing subsets at queue creation time, but it appears that the subset is chosen only at the message request handling level. Two messages belonging to same queue can be in different subsets. Also, we are uncertain how many servers are used by SQS, but it could be a cloud of servers.

VSQS takes a different approach: the subset of queue servers is preselected for each queue created; all messages for that queue go to the same subset of queue servers;
and when retrieving messages, VSQS will look into only this preselected subset. The subset will be identified at queue creation time, the subset can hold queues for different users, and queues for one user can be located in different queues.

One advantage of a pre-selection mechanism is that we know which subset is going to hold messages for a given queue, which makes it easier to have eventually consistent servers. When a message arrives, it will take some time to be saved into all instances in a subset, but since its location is known we can monitor consistency. When the server goes down, it may become inconsistent with other available servers. When it comes back on line, the system will have the capability of making it consistent with other servers within the subset. We will talk about this in more detail after we study the detail design of the system. Another advantage of this approach is to ensure having a message sent to the consumer component only once.

One disadvantage of a pre-selection approach is that we limit ourselves with pre-defined resources. We have a problem when a subset can hold a defined number of messages and the system has no idea how many messages a user is going to send. One possible solution for this dilemma is ‘fail-over’ where a queue can go to another subset of servers if the current one is out of resources.

Having a pre-selected subset is challenging: how many servers should each subset include? Having only a few servers in a subset increases the chance of subset unavailability, while having many servers can be an obstacle due to the complexity of having more network communications necessary to duplicate or remove a message from the queue. Only testing and monitoring can give us a good sense of the most accurate number.

### 3.5.2 Data Storage

Queues hold messages in data storage, which can be either persistent or non-persistent, either storing the messages in a DBMS or in memory. The clear advantage of storing messages in memory is fast lookup for messages when processing requests. However, memory usually is very expensive and limited compared to disk storage. Another disadvantage is that messages can be permanently lost if servers go down, which contradicts the main goal of high availability in a distributed queuing system.
Using persistent storage solves the issue of losing messages in case of system crash. On the other hand, it is slow. Fortunately many database systems offer in-memory capability where part of the messages are stored in memory and can be fetched faster when needed. Using in-memory capability can be tricky. Depending on how the database schema is designed, we do not need to have messages that are located in memory belong to only one queue while messages for other queues are all on disk. In our case, the queue server can hold multiple queues belonging to one or more users, and uses only one table to hold all messages. Therefore using in-memory capability offered by MYSQL is not sufficient to have an approximately equal number of messages for each queue in memory.

Amazon SQS most probably uses its own database system. VSQS uses its own hybrid data storage, which is a combination of MYSQL as RDBM and in-memory capability so that we have one single table per server to hold all messages assigned to that server and one in-memory storage per queue based on memory available.

### 3.5.3 Queue Life Cycle

VSQS users who are registered users can create as many queues as they are eligible for. Users need to make a request using the system manager website. They receive a URL and a **subsetId**; both need to be used in future requests to manage the newly created queue. Figure 3.8 shows the flow chart of creating a queue request followed by the steps to make the request.

![Figure 3.8. Create queue flow chart.](image)
1. Using the SM site, the registered user makes a request to create a queue, provides a name for the new queue, and must provide a unique name for each queue.

2. System Manager will look up the central database to choose the QSM with the least number of queues in it. This will assist in distributing the load.

3. The system manager then makes a request to create the queue to the chosen QSM using the “createqueue” web service API.

4. QSM calls all queue servers in the subset it manages to request creating a queue in each of them.

5. QSM returns back to SM.

6. SM then returns back to the user with two important pieces of information, the URL and SubsetId which must be used by the user in all future requests for the created queue.

Users can at any time delete the queue from VSQS systems. The queue will be deleted even if it is still has messages. Figure 3.9 shows the flow chart of deleting the queue request. The steps are:

1. User makes a delete queue request using the URL received when creating the queue.

2. QSM calls all queue servers to delete the queue.

3. QSM calls SM to delete the database entry of that queue from the central database.

4. QSM replies back to user.

![Delete queue flow chart](image-url)
3.5.4 Message Life Cycle

The messages life cycle starts when a user makes a put message request and ends when the message is deleted. This section describes each state of the message life cycle.

3.5.4.1 Put Message

The producer component of the user system makes a request to put a message into a queue using the “PutMessage” web service API. QSM creates a unique “messageId” for it and calls all queue servers in the subset to duplicate the message into all servers. The message is available for retrieval once it is saved into at least one server. The producer component that pushed the message will receive a status response of successful once it is saved or failed otherwise. The producer needs not to know about “messageId”. Figure 3.10 shows the put message flow chart.

![Put Message Flow Chart](image)

Figure 3.10. Put message flow chart.

3.5.4.2 Get Message

The consumer component of the user system makes a request to retrieve a message. First QSM forwards the request to only one randomly chosen queue server in the subset to retrieve a message and lock it by putting the message in the invisible state, QSM return the message to the user then makes a call to the rest of the queue servers in the subset to lock this message and put it in an invisible state. QSM returns the message along
with message Id to the caller. The message Id can be used later as a parameter to delete the message. Figure 3.11 below shows the get message flow chart.

![Get message flow chart](image)

**Figure 3.11. Get message flow chart.**

VSQS tries to retrieve messages in FIFO order. Due to the distributed nature of the queue and invisibility middle stage of the life of a message can cause cases where this is not true. For example, if a retrieve request is made and VSQS sends the message back but the consumer component never receives it, this message will be served again after the invisibility period which is the number of seconds that the messages stays invisible after a retrieval expires, during the invisibility of a message many other older messages can be retrieved hence all are actually out of order.

VSQS tries its best to send messages only once. But it is possible for the consumer to receive the message more than once. For example, if a consumer did not delete an already retrieved message and this message passed its invisibility period, the message then is served again. To prevent processing the message twice, consumers can use “messageId” as a defense mechanism to help identify messages previously received.

### 3.5.4.3 DELETE MESSAGE

The consumer component of the user system can use message Id returned when retrieving the message to request deletion after finishing processing the message. QSM,
after receiving the request, will forward the request to all queue servers in the subset to ensure deleting the message in all instances. Figure 3.12 shows the delete message flow chart.

3.5.4.4 **RETURN TO QUEUE**

If the consumer does not delete an already retrieved message, after the end of invisibility period VSQS returns the message back to the queue and makes it available for retrieval.

3.5.5 **Queue Server Life Cycle**

Queue servers go through three possible cycles based on its availability, the queue server can be up and running and available to receive requests, it is possible that the server goes down and becomes unavailable for some time, when the server comes back online and it spends some time to sync its queues with other queue servers in the same subset before it becomes available for requests. Queue server manager is the component that manages the subsets and all queue servers in a subset. QSM knows when the queue server goes down, monitoring it and update it when it comes back online. The main question is what is going to happen to the requests the system receives while one or more queue servers in one subset is down. Following are the three stages of queue server life cycle and the behavior of QSM in each stage.
3.5.5.1 QUEUE SERVER – AVAILABLE STATE

The normal stage of queue server life cycle is available and ready to receive requests. QSM has the knowledge of the availability of the queue server thus it forward all requests it receives from users down to the queue server. “GET_MESSAGE” is a special request in that QSM will choose randomly one queue server to get and lock the message and then it call the rest of queue servers to lock the message based on message identifier, the queue server at “available” stage can be chosen by QSM for this purpose. Figure 3.13 shows the path of request to an available queue server.

![Figure 3.13. QSM forward request received to available queue server.](image)

3.5.5.2 QUEUE SERVER – UNAVAILABLE STATE

Queue servers can go down for various uncontrolled reasons, it is not the intention of the system to bring queue servers down but it might go down because of hardware failure or network failure. When the queue server is down it will not receive any requests from the time it went down until it comes back up. QSM, the manager of queue server knows about unavailability of a queue server. QSM will mark the queue server as “unavailable” when it fails to send a request, this request will be saved into a place holder. QSM will retain a copy of all future requests for that unavailable queue server into a place
holder until the queue server comes back online. QSM will still forward user’s requests to all available queue servers. QSM will set a periodic monitoring mechanism on the unavailable queue server to check when it is going to be back online. Figure 3.14 shows the requests flow of a server just went down.

3.5.5.3 QUEUE SERVER—SYNCHRONIZING STATE

Queue server manager monitors unavailable queue servers waiting for it to come back online. Once the queue server is up again, QSM will mark it as “synchronizing” meaning that the queue server is back up but it is not fully available, QSM starts sending all request it finds in the place folder for this queue server in the order they were received, the ordering is essential so that QSM does not send delete message request before put message request as example.

The system might receive new requests while the queue server is synchronizing, the synchronizing queue server acts differently based on type of the request received. The synchronizing queue server acts as available for “PUT_MESSAGE” requests which are forwarded from QSM to the synchronizing queue server directly, there is no need to add the put message requests at the end of place holder since it causes no conflicts with older requests. Similarly “CREATE_QUEUE” request.
The synchronizing queue server acts as unavailable for “GET_MESSAGE” so that the queue server is not chosen by QSM as a first queue server to get and lock the message from, getting messages happen on only available servers, this mechanism prevents a possible duplicate retrieval of the same message, the synchronizing server might still hold messages that are either locked or deleted where the lock or delete requests are still in the place holder waiting to be processed.

Processing “LOCK_MESSAGE” or “DELETE_MESSAGE” requests on a synchronizing queue server can either be forwarded directly to the queue server to speed up the synchronization process or be put in the place holder to prevent conflicts in processing requests that belong to the same message. If the request belongs to a message that has an entry in the place holder, then the entry will be updated, this will prevent sending delete or lock request before add message request. If the request doesn’t have an entry in the place holder then there is no harm of forwarding the request directly to the queue server. “DELET_QUEUE” is added to the place holder if it was received in the time of synchronizing the queue server.

After QSM finishes sending all requests found in place holder it then marks the queue server as available. Figure 3.15 shows the flow of old and new requests on a queue server that just came back online.

![Figure 3.15. Queue servers in synchronizing state.](image)
CHAPTER 4

SOFTWARE TEST AND RESULTS

Software testing cycle is an integral part of the software development cycle. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

A primary purpose for testing is to detect software failures so that defects may be discovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for members of the testing team. Information derived from software testing may be used to correct the process by which software is developed.

4.1 TYPES OF SOFTWARE TESTING

The development process involves various types of testing. Each test type addresses a specific testing requirement. The most common types of testing involved in the development process are:

- **Acceptance testing:** Testing to verify a product meets customer specified requirements. A customer usually does this type of testing on a product that is developed externally.
- **Black box testing:** Testing without knowledge of the internal workings of the item being tested. Tests are usually functional.
- **Functional testing:** Validating that an application or Web site conforms to its specifications and correctly performs all of its required functions. This entails a
series of tests which perform a feature by feature validation of behavior, using a wide range of normal and erroneous input data. This can involve testing of the product's user interface, APIs, database management, security, installation, networking, etc. Testing can be performed on an automated or manual basis using black box or white box methodologies.

- **Load testing**: Load testing is a generic term covering Performance Testing and Stress Testing.

- **Performance testing**: Performance testing can be applied to understand the application or WWW site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.

- **Unit testing**: Functional and reliability testing in an Engineering environment. Producing tests for the behavior of components of a product to ensure their correct behavior prior to system integration.

### 4.2 TESTING VSQS

The queuing system built and presented in this thesis project promises users of high availability of queues they create as well as the availability of messages at any given time. Various types of testing are required to prove this promise. VSQS is built for academic research purposes only and not for commercial use. The tests conducted against the system are to ensure the workability and performance of the system to assist in carefully analyzing the system, the properties and its performance. Unit testing, functional testing and performance and load testing are thought to be the essential test methodologies to prove that the system works as intended and to study the performance of the system. Following are the types of testing conducted against VSQS.

#### 4.2.1 Unit Testing

Unit testing is meant to test individual units of the code separately to ensure that each unit is doing what their implementations claim. The java programming language [17] units are actually the methods created inside each class. To unit test VSQS means testing the methods in each class included in the project. Java developers use JUnit [18] framework as a tool to unit test the software. VSQS is built using spring framework where some of the objects are created as beans by spring and are injected into other objects that use them; therefore, Spring Junit [19] is used to benefit from the abstract injection class to
assist in creating the beans and injecting them into the test classes. Below is the base class created that should be extended by every test class:

```java
public class BaseSpringTestCase extends AbstractDependencyInjectionSpringContextTests{

    protected static FileSystemXmlApplicationContext factory;
    protected static Random generator = new Random();

    private static Log log = LogFactory.getLog(BaseSpringTestCase.class);

    static {
        final String UNIT_TEST_CONFIG_FILE="unit_test.properties";
        // Read properties file.
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream(UNIT_TEST_CONFIG_FILE));
        } catch (IOException e) {
            log.error("BaseSpringTestCase", e);
        }
        //merge custom properties and system properties
        System.getProperties().putAll(properties);
    }

    protected String[][] getConfigLocations() {
        return new String[][] {System.getProperty("applicationContext")};
    }

    public BaseSpringTestCase() {
        super.setAutowireMode(AbstractDependencyInjectionSpringContextTests.AUTOWIRE_BY_NAME);
    }
}
```

Sample property file “unit_test.properties” used for testing is:

```
queuesystem.db.host=localhost:3306
queuesystem.db.dbname=queuing_srv
queuesystem.db.user=root
queuesystem.db.password=root
default.ip.address=127.0.0.1
applicationContext=applicationContext.xml
```

There is no necessity to list all unit test classes in this section; however, a sample test class is provided to highlight the necessary details to create new test classes as needed
if an improvement is done to the system and more components and classes are added. The
sample shown below is a test class written to test “QueueManger.java” object and it can
be used as a template to write new Junit test classes. Before showing a code snippet, the
following are a few points that need to be considered when writing a unit test:

- The test class is named “QueueMangerTest.java”; note that the name of the test
class has the word “Test” appended to the original class name.
- The test class must extend “BaseSpringTestCase.”
- Auto wiring capability of spring is used to inject any needed bean; in this example
the object “MessageQueueDaoManager” is injected by spring via using
“setMessageQueueDaoManager” method.
- Use method “onSetUp” to create object or initialize variables needed to perform
the test; this method is called automatically by Junit before each test method is
called.
- Use method “onTearDown” to clean up after the test is done; this method is called
automatically by Junit after each test method is done.
- A test method implementation can be written for each method in the original class;
the test method needs to be prefixed with the word “test”, e.g., use
“testDeleteMessage” name to create testing method for “deleteMessage” method
in the original class.

```java
public class QueueManagerTest extends BaseSpringTestCase{

    private MessageQueueDaoManager messageQueueDaoManager;
    private QueueManager queueManager;

    public void setMessageQueueDaoManager(
            MessageQueueDaoManager messageQueueDaoManager) {
        this.messageQueueDaoManager = messageQueueDaoManager;
    }
    public void onSetUp(){}
    public void onTearDown(){}
    public void testQueueManager(){assertNotNull(this.queueManager);

    assertTrue(this.queueManager.getQueueId().equalsIgnoreCase(
            "test_queue1"));
    assertTrue(this.queueManager.getQueueName().equalsIgnoreCase(
            "queue1"));

    assertTrue(this.queueManager.getInvisibleInSeconds() == 50);
    assertNotNull(this.queueManager.getMessageQueueDaoManager());
}
```
4.2.2 Functional Testing

The next level of testing conducted is functional testing. Functional testing refers to tests made to verify a specific action or function of the system. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question "can the user do this" or "does this particular feature or function work." Performing an action supported by the software usually goes through multiple units of the code and for a distributed system it might span multiple servers or system components; one component of the system might receive a request while another system actually performs the action. For example, a request comes to a Web Service component to create a queue for a user; however, the request is actually performed by another system, the RDBM that creates the persistent queue. Both queue server and queue server manager are tested separately to ensure that all functions supported by each component are working as expected. Function testing does not include load testing.

Various functional tests were conducted against queue server and queue server manager, each functional test corresponded to a function like add, retrieve or delete a message. Tests were conducted using a direct call using the web service API provided by each component. Figure 4.1 shows the general flow of receiving and processing the requests until the response is returned. All functional tests have passed successfully and the response returned was as expected. The database was also checked and verified.

![Figure 4.1. General flow of processing requests is used in functional testing.](image-url)
4.2.3 Performance and Load Testing

Performance and load testing are very important to ensure that the system is working as expected under all conditions. They also help in spotting the limitation of the system and finding edge cases where the system might not perform as expected. These tests are not related to a specific function but are an indicator of the system performance and effectiveness. Some other types of testing, answers questions like “how many requests the system can take per second” or “how many seconds it takes the system to respond to the requester under load.” These types of questions can be answered through load testing.

The main goal of conducting performance testing on VSQS is to show that adding a mechanism to preselect a subset of queue servers where all messages of one queue are distributed to the same preselected subset didn’t impact the high availability of the system. The test should also show that the system was able to overcome the message properties issues that appear when using Amazon SQS; for example, users might retrieve the same message more than once, the messages are not retrieved in the same order as they were received, and a retrieved message request might not result in returning a message even though the queue contains a message. The test represented in this section also assists in shedding light on the system performance by measuring the average response time, throughput and latency. The system prepared for testing consists of one queue server manager and three queue servers that make up one subset.

Ideally, each of the queue servers needs to be installed on a separate physical machine, deployed into an application server like JBOSS and connected to its own database. It is normal nowadays to find servers running with 16 to 32 GB of memory and 1TB of disk space, yet more resources are used to improve the performance of a system. A new load and performance testing need to be conducted on VSQS every time it is installed on a new machine.

For the sake of this thesis project the test is simulated by using one machine, the queue server manager, and all queue servers are deployed into one JBOSS as separate “war” files. One MYSQL server is used to hold all databases. The JBOSS used is set with minimum memory allocation of 128MB and maximum of 512MB. JBOSS is installed in WIN XP on an IBM machine with a single core processor of 1.8GHz and 1.5GB of RAM.
After they are deployed, each of the components has a distinctive URL. To simulate the crashing of queue servers, “shutdown” web service API is used, and then “start” web service API is used to bring it up again.

Performing these types of tests requires more than placing a parameter in a web form, clicking on a submit button and watching the result. The flexibility of running a simulation of multiple users making lots of requests is needed; also required is a mechanism to collect result data and analyze it. There are quite a few open source systems that can assist in such testing, such as Jmeter [20], SoapUI [21] or Grinder [22]. JMeter was chosen in this thesis project to assist in testing, and it is used as a replacement for user’s producers and consumers’ components. Jmeter can be used to simulate as many producers and consumers as needed.

4.2.3.1 JMETER

Apache JMeter is a pure Java desktop application designed to test client/server software, such as a web application or web services. It may be used to test the performance of both static and dynamic resources, such as static files, Java Servlets, CGI scripts, Java objects, databases, FTP servers, and more. JMeter can be used to simulate a heavy load on a server, network or object in order to test its strength or to analyze overall performance under different load types. Additionally, JMeter can help to perform regression tests of the application by creating test scripts with assertions to validate that the application is returning the expected results. For maximum flexibility, JMeter lets its users to create these assertions using regular expressions. Stefano Mazzocchi of the Apache Software Foundation was the original developer of JMeter. He wrote it primarily to test the performance of Apache JServ. It was then enhanced to add functional testing capabilities.

All test plans discussed in this section were created as ten simultaneous threads; each thread represents either a producer of a message or a consumer of a message depending on the test. Each thread sleeps randomly between 0 and 1000 milliseconds before it repeats the tests. The results of each test are saved into a “csv” file for analyzing purposes, and the results analysis will be shown in three different formats; the first format
is summary report. Summary report is a raw output of information that resulted in running the test as shown in Figure 4.2.

![Summary Report Table]

Figure 4.2. Jmeter summary report example.

The second format is called Distribution Graph. The distribution graph will display a bar for every unique response time. The graph will draw two threshold lines: 50% and 90%. What this means is 50% of the response times finished between 0 and the line. The same is true of 90% line. Figure 4.3 shows a sample distribution graph.

![Distribution Graph]

Figure 4.3. Distribution graph sample that shows 50% and 90% threshold lines.
The third format is called **Spline Visualizer**. The spline visualizer presents a smooth curve that represents the data of the entire test run. There are 10 points on the curve; each point represents an average of some subset of the sample run. For instance, if the test run has been running for some time, and there are 1000 samples, the first "point" on the curve will represent an average of the first 100 samples, the next will be for the second 100 samples, etc. These points are connected using **spline** curves, and allow users to see the entire history of the test and how the responsiveness of their application has changed over time. Figure 4.4 shows a sample of spline visualizer graph.

![Spline Visualizer graph sample.](image)

The testing in this section represents users’ requests made by producers and consumers to manage their queues. All requests go directly to QSM web service API which will then forward them to all queue servers in the subset; therefore, all tests seen in this section were made to QSM Web Service API. Collecting results, however, might
involve queue servers. For example, making sure that put message request was completed in putting the messages in all queue servers.

4.2.3.2 GENERAL SYSTEM PERFORMANCE

The first set of tests is conducted to show the general system performance and response time of requests made to the system as well as the throughput. Each test is repeated several times to ensure the consistency of the system performance. Studying the results of the general system performance helps in studying and comparing the results of later test cases.

Users can have multiple producers, each of which puts messages into the queue by calling “PUT_MESSAGE” request. The test simulates ten producers trying to put messages into the queue; each producer makes 1000 requests with a random delay of zero to one second between requests for each producer. The sample distribution graph shown in Figure 4.5 shows that 50% of the requests took 18 ms or less while 90% of the requests took 50 ms or less, the remaining 10% took longer than 50ms.

![Distribution Graph (alpha)](image)

**Figure 4.5.** Distribution graph of 10 producers making 1000 put message requests with zero to one second delay in between requests.
The graph in Figure 4.6 shows a peak at the beginning of the test which might be related to JMeter preparing the context that it is running on including creating the threads. The graph shows a more steady and low response time in later requests until the end of the test.

![Spline Visualizer graph](image)

**Figure 4.6.** Spline Visualizer graph of 10 producers making 1000 put message requests with zero to one second delay in between requests. The peak is at the beginning of the run only.

The same test was repeated four times. The repetition is important to prove that the system behaves approximately the same every time. All test samples were run under the same machine condition. Table 4.1 shows the results of all sample tests; these results are taken from the summary report provided by Jmeter.

**Table 4.1. Results of Few Tests of 10 Producers Making 1000 Put Message Requests With 0 to 1 Second Delay in Between Requests**

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Throughput</th>
<th>50%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>2</td>
<td>680</td>
<td>26</td>
<td>18.6</td>
<td>19</td>
<td>49</td>
</tr>
<tr>
<td>Test 2</td>
<td>2</td>
<td>360</td>
<td>26</td>
<td>18.4</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>Test 3</td>
<td>2</td>
<td>234</td>
<td>25</td>
<td>18.1</td>
<td>19</td>
<td>48</td>
</tr>
<tr>
<td>Test 4</td>
<td>2</td>
<td>339</td>
<td>26</td>
<td>18.2</td>
<td>19</td>
<td>51</td>
</tr>
</tbody>
</table>
The table shows that 90% of sample requests took around 50ms or less, leaving only 10% of the requests that took over 50ms. It is noted that some requests took a maximum of around 300ms, and one or more requests in the first test took up to 680ms; however, according to the Spline graph that shows a peak at the beginning of the test which can be related to Jmeter initialization. The conclusion is that only a few requests took close to the max recorded since the average response time stays relatively low at 26ms of the entire test. As seen in the table, there is only a slight difference in response time or throughput throughout all test samples which indicates that the system is consistent in terms of response time of processing put message requests.

Users can have multiple consumers, each of which attempts to retrieve a message off the queue by calling “GET_MESSAGE” request; users normally delete the message after processing it by calling “DELETE_MESSAGE” request. The test simulates ten consumers, each of which makes 1000 get message requests followed by a delete message request; each consumer includes a random delay of zero to one second between requests. The sample distribution graph is shown in Figure 4.7.

![Figure 4.7. Distribution graph of 10 consumers making 1000 get message requests followed by a delete request of the retrieved message. A delay of zero to one second is applied to the consumers.](image-url)
It shows that 50% of the requests took 44ms or less which is more than the recordings shown for testing put message requests and more than is expected because getting a message requires a call to queue server to reserve and lock a message before returning the response to the user. On the other hand, 90% of the requests took 149ms or less which means 40% of the requests added an average of 100ms to the response time. This indicates a high latency added that might be related to one of these factors:

- The attempt to prevent a message from being retrieved more than once and improve the effort of retrieving the messages in the order they were received has a cost of extra latency.
- The implementation of In-Memory cache where messages are kept for fast retrieval is not efficient enough; there is a possible leak when loading messages from the database into the cache.
- The limited resources the machine is running on, JBOSS has only a maximum of 512MB of Heap size that might cause more memory paging as the size of request increases.

The spline visualizer showed in Figure 4.8 shows few peaks in the response time throughout the test; these peaks are related to the increase of virtual memory that the system acquires during the run.

![Spline Visualizer](image)

**Figure 4.8.** Spline Visualizer graph showing 10 consumers making 1000 get message requests followed by a delete request of the retrieved message. A delay of zero to one second is applied to the consumers.
The same test was repeated for four times on the same machine and under the same conditions to see if the system has a consistency in recording the response time and throughput. Table 4.2 shows the results of these tests. The times shown in the table combine the time consumed for both GET_MESSAGE and DELETE_MESSAGE requests. The time consumed to get a message is usually longer than put or delete a message and that is due to the time QSM takes to first get a message from one queue server and lock it before it returns it to the user. All four tests show a very close throughput which is around 17 messages per second. All tests show that 50% of the requests took around 45ms and 90% took around 149ms. The timings show a maximum of around 2.5 seconds for few messages; this is due to heap allocation limitation of JBOSS. During the run, JBOSS halted a couple of times and where the virtual memory allocation increases, the cycles used to allocate the extra needed virtual memory caused few peaks on processing request. Later in this chapter, latency is closely studied which shows similar behavior.

Table 4.2. Results of Few Tests of 10 Consumers Making 1000 Get Message Requests Followed by a Delete Request of the Retrieved Message

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Throughput</th>
<th>50%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>2</td>
<td>2287</td>
<td>68</td>
<td>17.1</td>
<td>44</td>
<td>149</td>
</tr>
<tr>
<td>Test 2</td>
<td>2</td>
<td>2424</td>
<td>67</td>
<td>17.3</td>
<td>46</td>
<td>149</td>
</tr>
<tr>
<td>Test 3</td>
<td>2</td>
<td>2795</td>
<td>69</td>
<td>17</td>
<td>44</td>
<td>149</td>
</tr>
<tr>
<td>Test 4</td>
<td>2</td>
<td>2668</td>
<td>68</td>
<td>17.1</td>
<td>44</td>
<td>150</td>
</tr>
</tbody>
</table>

4.2.3.3 SYSTEM AVAILABILITY

The main goal of this system is high availability; user’s requests must go through at any given time and under any condition. The tests represented in this section are an attempt to spot the cases where the system might not be available. There are some clear cases where the system is going to be unavailable; these tests will not be considered in our test and they are:

- The master and “failover” QSM servers are all down.
- QSM gets more hits per milliseconds than it can handle; this is dependent on the resources available on the machine QSM is installed on.
• All queue servers are down.

The first test simulates 10 producers trying to put messages into the same queue. Each producer issues a **PUT_MESSAGE** command a total of 2000 times; at the end of the test a total of 20000 messages should be present in the queue. The producers sleep randomly between zero and 1 second between requests. All queue servers are available in this test. The assertion of a successful call is done by validating the xml response to match the **xpath** `{/request/status/text()="success"}`. Figure 4.9 shows a peak at the beginning of the test which, as described before, might be related to Jmeter.

![Figure 4.9. Spline Visualizer of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests.](image)

The summary report is shown in Figure 4.10. It shows that the test sent 20000 sample requests to QSM with 0.00% error, meaning that all requests were received and processed by QSM system; As a second check, the database of all queue servers have exactly 20000 messages. 50% of the requests in this test sample took 17ms or less while 90% of the requests took 46ms or less. The test took approximately 18 minutes to finish with a throughput of 18.7 messages per second.

![Figure 4.10. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests.](image)
The second test is similar to the first test except that one queue server goes down randomly in the middle of the run and stays down until the end of the test. This test helps to see if the system stays available and able to receive requests. Figure 4.11 shows the summary report of this test that clearly proves the availability of the system where all 20000 messages were received by QSM and forwarded to all available queue servers; the database of all queue servers contains all 20000 messages. The average response time is 22 ms, 50% of the requests took 16ms or less while 90% of the requests took 43ms or less. The conclusion is that bringing one queue server down didn’t affect the availability of the system nor affect the throughput.

![Figure 4.11. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run.](image)

The graph shown in Figure 4.12 shows a similar behavior to the test conducted where all queue servers are available. A spike in response time is noticed at the beginning of the run only.

![Figure 4.12. Spline Visualizer of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run.](image)
The graph in Figure 4.13 shows 50% of the requests took 16ms or less while 90% of the requests took 43ms or less. These recordings are similar to the recordings seen where all servers are available. It may be concluded that bringing one queue server down did not have any impact on the distribution of the response time.

Figure 4.13. Distribution graph of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests while one queue server goes down in the middle of the run.

The third test is to shut down queue servers and bring them up again randomly, ensuring the availability of at least one queue server at any given time. The summary report shown in Figure 4.14 indicates all 20000 messages were processed successfully and forwarded to the available queue servers at the time they were received.

Figure 4.14. Summary report of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests. Queue servers go down randomly with at least one queue server available at any time.
The summary report shows an average time of 39ms which is higher than the average time recorded when all servers were available or one queue server was unavailable. When a queue server goes down, all missed requests go to a place holder; when the queue server comes back online, QSM starts sending all missed requests. This process has an impact on processing new requests due to the system’s limited resources.

The graph shown in Figure 4.15 shows some peaks at the beginning of the test, during the test and even at the end of the test. The same behavior holds on repeated tests under the same machine conditions. Having queue servers go down and then back up during the run shows a clear negative impact on the average response time; the reason for this is that when one or more queue servers go down, they miss all the requests the system receives during that period, and when the queue server comes back online, QSM starts sending all missed requests to the queue server as one batch. Doing this uses machine resources and therefore impacts the response time of the new requests. Depending on the number of requests missed, this process might take a considerably long time.

Figure 4.15. Spline graph of running 10 producers; each put 2000 messages with delay ranges between 0 to 1 second in between requests Queue servers goes down randomly with at least one queue server available at any time.
Figure 4.16 shows the distribution graph of the test: 50% of the requests took 26ms or less, 90% of the requests took 84ms which is higher than the recordings where all servers are available or only one queue server is down.

The fourth test is to shut down queue servers in a series pattern. The queue servers go down in the middle of the run and stay down for a considerable length of time considerably long time. The decision to bring a queue server down is measured by the number of requests; for example, the first queue server goes down after the first 2000 requests and stays down for 4000 requests, and the other two queue servers follow the same pattern. This pattern ensures the overlapping of the down time of queue servers and therefore there might be a time where no queue server is available and so some requests will be rejected. As discussed in chapter 3, a queue server may not be available as soon as it comes back on line; it first has to sync up with all the requests that it missed while it was down. The sync up might take a considerably long time depending on the number of
missed requests and the number of new requests the system is receiving while the queue server is synchronizing. The summary report shown in Figure 4.17 indicates that almost 30% of the requests were rejected, reflecting the unavailability of the system.

**Table 4.17. Summary report of running 10 producers; each puts 2000 messages with a delay range between 0 to 1 second in between requests. Queue servers go down randomly.**

<table>
<thead>
<tr>
<th>Label</th>
<th># Samples</th>
<th>Average</th>
<th>Min</th>
<th>Max</th>
<th>Std Dev</th>
<th>Error %</th>
<th>Throughput</th>
<th>KB/sec</th>
<th>Avg. Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_message</td>
<td>20000</td>
<td>24</td>
<td>2</td>
<td>931</td>
<td>33.19</td>
<td>29.95%</td>
<td>18.4/sec</td>
<td>8.13</td>
<td>341.2</td>
</tr>
<tr>
<td>TOTAL</td>
<td>20000</td>
<td>24</td>
<td>2</td>
<td>931</td>
<td>33.19</td>
<td>29.95%</td>
<td>18.4/sec</td>
<td>8.13</td>
<td>341.2</td>
</tr>
</tbody>
</table>

The *spline* graph shown in Figure 4.18 is interesting in that it shows few peaks throughout the run. These peaks occur when a queue server comes back online and is currently in a synchronization state which has an impact on response time. In two other spots in the graph, the curve goes toward the bottom of the graph; this occurs when all queue servers are down and response to requests in this case is at lower rate.

**Figure 4.18. Spline graph of running 10 producers; each puts 2000 messages with a delay range between 0 to 1 second in between requests. Queue servers go down randomly.**
Figure 4.19 represents the distribution graph. It shows that 50% of the requests took 13ms or less. This is very low compared to a previous test where the system was available. The low response time is due to the 33% rejected; the rejected request is sent back to the user faster than the accepted request since there will be no processing on any sort.

The forth test shows that VSQS reached an unavailability state, even though the scenario of having all queue servers going down harshly for a considerably long time might not be realistic, but it is a possibility. The issue of the system being unavailable defeats the purpose of building a high availability queuing system. For a subset to be available, it must have at least one available queue server. If all queue servers in a subset are either crashed or in a synchronization state, then the subset is considered to be unavailable. There are a few factors that play a role in deciding the unavailability of the system:

- **Number of queue servers in the subset**: VSQS is built based on replicating messages into multiple queue servers. The number of queue servers included in the subset has an impact on the availability; the probability of (N +1) servers to be unavailable at the same time is less than the probability of (N) servers.
• **Queue server down time:** When a queue server goes down, this reduces the number of available servers in the subset by one which increases the probability of the system being unavailable.

• **Queue server synchronization time:** When a queue server comes back online, it goes through a synchronization process; all messages received during down time need to be sent to the queue server before marking it as available. The synchronization time depends on the number of requests the system received while the server was down; it also depends on the number of requests the system receives while the server is in a synchronization state. The higher number of requests the longer it takes the queue server to be available. The subset is considered unavailable if all of its queue servers are in a synchronization state.

• **The average number of requests:** The rate of requests received by the system impacts the unavailability of a queue server and in turn the availability of the system. While the server is down, all received messages go a place holder waiting for the server to come back online to be sent again. Receiving a high volume of requests while the queue server is in a synchronization state lengthens the unavailable time.

• **Total Unavailable Time:** The total unavailable time of a queue server can be calculated as:

\[
\text{total unavailable time} = \text{down time} + \text{synchronization time}
\]

The following test tries to emphasize these factors and assists in studying the behavior of system unavailability. The test consists of a group of threads that send "PUT_MESSAGE" requests, while another group of threads send "GET_MESSAGE" requests followed by "DELETE_MESSAGE" requests, and these two groups send an average of 1500 requests per minute. The other component of the test is to bring queue servers down in intervals. Each queue server goes down for a fixed number of minutes; that is, each queue server goes down for \(M\) minutes, and queue servers go down in intervals of \(N\) minutes.

The test is repeated three times; each test changes the number of minutes of queue server down time. The longer the down time, the more requests missed by a queue server which has an impact on the length of time that the queue server takes to sync up with the rest of the queue servers.

The queue servers go down in intervals of \(N\) minutes. Every test starts with zero minute intervals, meaning that the next server to go down waits zero minutes once the previous down server comes back up. The interval is increased until we have an available system even if using two queue servers. When a queue server comes back online, it needs
some time to sync up and that depends on the number of requests it missed as well as the number and type of requests received by the system during synchronization as discussed in chapter 3, section “3.5.5.3.” Both lock and delete requests might go into the placeholder, waiting for older requests to be processed first.

The test will use up to five queue servers to better illustrate the effect of factors on the system. The results recorded for the subset consist of 2, 3, 4 and 5 servers in order to see the effect of increasing the number of queue servers in the subset. The test will not consider bringing down more than one server at a time, and the reason is that bringing two servers down out of five queue servers in a subset is similar to bringing one queue server down out of a subset of four servers.

This test set up might not occur during the life of a queuing system; however, it helps measure the time needed for servers to sync up without affecting the availability of the system. A real life example is how to provide maintenance to queue servers without affecting the availability of the system.

Table 4.3 shows the result of bringing servers down for one minute, “U” is for unavailable and “A” is for available The results show that if queue servers are brought down in a sequence interval of 0 minutes between servers then the subset needs to have five queue servers for the system to be available. Putting a delay of 1 minute interval makes a four server subset enough to have an available queuing system. A two server subset requires approximately four minutes interval between the two servers to declare an available system.

<p>| Table 4.3. Availability and Unavailability Recordings of Subsets, 1 Minute of Down Time |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|</p>
<table>
<thead>
<tr>
<th>Subset-Size</th>
<th>2 servers</th>
<th>3 servers</th>
<th>4 servers</th>
<th>5 servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval 0</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
Table 4.4 shows the results of queue servers going down for 2 minutes. A zero interval was again enough for a subset of 5 servers to stay available. It was noted that a two minute interval is not enough for a subset of 3 servers to be available as seen in Table 4.3 shown in the previous page.

Table 4.4. Availability and Unavailability Recordings of Subsets, 2 Minutes of Down Time

<table>
<thead>
<tr>
<th>Subset-Size</th>
<th>interval</th>
<th>2 servers</th>
<th>3 servers</th>
<th>4 servers</th>
<th>5 servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5 shows the results when increasing the down time of queue servers to 4 minutes. The expected results are a queue server requires more time for synchronization due to more requests missed. A subset of 5 queue servers is no longer enough to ensure the availability of the system. A subset consists of two queue servers that require 10 minute intervals to be highly available.

Table 4.5. Availability and Unavailability Recordings of Subsets, 4 Minutes of Down Time

<table>
<thead>
<tr>
<th>Subset-Size</th>
<th>interval</th>
<th>2 servers</th>
<th>3 servers</th>
<th>4 servers</th>
<th>5 servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>U</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>U</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
The above test shows that increasing the number of queue servers in a subset reduces the possibility of the system being unavailable; however, it is not realistic to keep increasing the number of queue servers. The test also shows that the longer a queue server stays down, the more missed requests which lengthen the synchronization time and make the server unavailable. The system is considered unavailable if all queue servers are either down or synchronizing. What happens when a queue server goes down for one full day? More requests are missed and a longer period is needed for synchronization. Most likely, all the messages the server contained no longer exist in other queue servers within the subset, meaning that more time is wasted doing unnecessary synchronization.

The question remains, what is the likelihood that all queue servers in the subset will become unavailable or in a synchronization state. Another test design was set to capture the possibility of a system becoming unavailable; this test uses two groups for threads that continuously send all kinds of requests. The subset consists of five queue servers; each queue server goes down once every hour for 6 minutes, that is 10% down time for each server. Queue servers go down in a random fashion; the test uses JMeter randomness capability. The test was repeated for several times, and the system didn’t suffer from unavailability. Most likely, the test ensured a perfect randomness; however, it didn’t help to find the rate of unavailability of VSQS. VSQS still has a greater chance of being unavailable compared to Amazon SQS due to the fixed number of queue servers included in the subset, besides synchronization time.

4.2.3.4 SYSTEM LATENCY

Latency, a synonym for delay, expresses how much time it takes for a response to be sent back to the requester as a result of processing the request. Many factors might play a role in system latency. The producer and consumers of the user’s application make different types of requests to VSQS, mainly to QSM, while producers send in **PUT_MESSAGE** command, and consumers send **GET_MESSAGE** and **DELETE_MESSAGE**. Each of these requests has a different implementation that results in a different effect on latency. The increase of requests per second might have an impact on latency. Other reasons which might have an impact on latency are networking,
hardware and software limitations and database connectivity. This section goes over a couple of scenarios and tries to measure the amount of latency each one produces.

### 4.2.3.4.1 Increase Number of Requests

The first test scenario is to increase the number of requests on a fixed number of threads. For example, we have a fixed number of producers each of which calls the server; the number of calls each producer will make increases. The test starts by causing the 10 producers to make 1000 requests and then doubling the number of calls until 32000 is reached. The thread will sleep randomly between 0 and 1000 milliseconds. Table 4.6 shows summary reports of all runs made, starting with 1000 and ending with 32000. The result of this test shows that increasing the number of requests on a fixed number of threads doesn’t add any latency to the response time; the average response time stays within a reasonable range.

#### Table 4.6. Response Time Results in Milliseconds While Increasing the Request Size

<table>
<thead>
<tr>
<th>Request-Size Measure</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
<th>32000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>365</td>
<td>230</td>
<td>338</td>
<td>432</td>
<td>385</td>
<td>393</td>
</tr>
<tr>
<td>Average</td>
<td>27</td>
<td>35</td>
<td>38</td>
<td>33</td>
<td>33</td>
<td>29</td>
</tr>
<tr>
<td>Throughput</td>
<td>17.9</td>
<td>17.8</td>
<td>17.9</td>
<td>18.3</td>
<td>18</td>
<td>18.3</td>
</tr>
</tbody>
</table>

### 4.2.3.4.2 Increase Memory Cache Size

Each queue server loads a fixed number of messages into memory cache and makes them available for fast retrieval. When the cache gets emptied the queue server starts to load more messages into the cache. The test helps to measure the latency added to the response to `GET_MESSAGE` requests. The test is accomplished by changing two factors: the number of requests and the size of the cache. The test consists of multiple runs by doubling the request size, starting with 1000 requests and ending with 16000 requests. The runs will be repeated on different cache sizes of 1000, 2000, 4000 and 8000. Each thread makes a `GET_MESSAGE` request followed by a `DELETE_MESSAGE` request.
All recordings shown in this section belong only to get message requests. JMeter individual summary reports may not help us study the numbers closely; therefore, the results of all tests are gathered in separate tables, one table for the average min and one for the average max response time and throughput.

Minimum response time of requests stayed very close as shown in Table 4.7. Increasing the cache size didn’t have an impact on the minimum response time recorded on any request size.

Table 4.7. Minimum Response Time Recordings in Milliseconds When Cache Size Increases

<table>
<thead>
<tr>
<th>Cache-size</th>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>18</td>
<td>14</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>17</td>
<td>17</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>18</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

Maximum response time recordings in milliseconds are shown in Table 4.8. There is a possibility that a GET MESSAGE request takes longer time than average when there are no messages left in the cache and QS has to pull messages from the data base table and puts them into memory.

Table 4.8. Maximum Response Time Recordings in Milliseconds While Increasing the Cache Size

<table>
<thead>
<tr>
<th>Cache-size</th>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>945</td>
<td>1186</td>
<td>2040</td>
<td>2826</td>
<td>3269</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>779</td>
<td>853</td>
<td>2184</td>
<td>2808</td>
<td>3521</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>816</td>
<td>1087</td>
<td>2257</td>
<td>2845</td>
<td>3280</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>804</td>
<td>1534</td>
<td>2138</td>
<td>2491</td>
<td>3146</td>
<td></td>
</tr>
</tbody>
</table>
This possibility is minimal since the implementation of the cache has monitoring that checks the number of messages in the cache and starts to load more messages once the number of messages loaded into the cache reaches less or equal to half the cache size. The results show that there is no clear pattern of how increasing the cache size impacts the maximum response time. However, it is noticeable that the maximum response time increases when the request size increases. This phenomenon is due to the limitation of memory allocation for JBOSS. During the run, JBOSS halts for some time while more virtual memory is allocated; the bigger the request size, the more halts are noticed. The average response time recordings are shown in Table 4.9.

### Table 4.9. Average Response Time Recordings in Milliseconds While Increasing the Cache Size

<table>
<thead>
<tr>
<th>Cache-size</th>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>100</td>
<td>100</td>
<td>91</td>
<td>86</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>115</td>
<td>103</td>
<td>93</td>
<td>89</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>121</td>
<td>104</td>
<td>89</td>
<td>86</td>
<td>82</td>
<td></td>
</tr>
</tbody>
</table>

The results show that increasing the cache size for each request size separately did not have a great impact on the average response time. The bigger request size is, the smaller the difference in average response time when doubling the cache size. For instance, for a request size of 16000, we see an average of 84ms when setting the cache size to 1000, but the average response time goes down to 82 when setting the cache size to either 2000 or 4000. The difference is always a couple of milliseconds, but increasing the cache size does not always add improvement to the average request time; actually it seems to be random. Later in this section we will focus on this in discussing the distribution graph.

The small request size of 1000 is an exception in that it shows a longer average response time when doubling the cache size. This can be due to the fact that a bigger cache size requires more time to be filled with messages which adds some delay to some messages; this delay will show more when the request size is small, and it will vanish.
when more requests arrive to the system. It is also noticeable that the bigger the request size, the smaller the average response time.

The distribution graph of all these tests is shown in two tables. Table 4.10 shows the results of a 50% threshold line. While Table 4.11 shows the results of a 90% threshold.

Table 4.10. Results of 50% Threshold Line of the Distribution Graph of Response Time in Milliseconds While Increasing the Cache Size

<table>
<thead>
<tr>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>78</td>
<td>77</td>
<td>71</td>
<td>65</td>
<td>67</td>
</tr>
<tr>
<td>2000</td>
<td>88</td>
<td>81</td>
<td>72</td>
<td>70</td>
<td>65</td>
</tr>
<tr>
<td>4000</td>
<td>92</td>
<td>80</td>
<td>70</td>
<td>67</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 4.11. Results of 90% Threshold Line of the Distribution Graph of Response Time in Milliseconds While Increasing the Cache Size

<table>
<thead>
<tr>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>191</td>
<td>185</td>
<td>169</td>
<td>154</td>
<td>155</td>
</tr>
<tr>
<td>2000</td>
<td>NA</td>
<td>NA</td>
<td>171</td>
<td>162</td>
<td>150</td>
</tr>
<tr>
<td>4000</td>
<td>NA</td>
<td>NA</td>
<td>164</td>
<td>155</td>
<td>149</td>
</tr>
</tbody>
</table>

The results shown in these two tables have the same behavior as we saw in the average response time in Table 4.6 on page 91. When the request size increases, the 50% and 90% threshold decreases and becomes more uniform across all different cache sizes. Also, it is noticeable that the small size of 1000 is an exception. However, looking at these numbers doesn’t give a clear pattern. For example, the request size of 8000 has a 50% threshold of 65ms when the cache size is set to 1000; increasing this cache size to 2000 makes the 50% threshold to increase to 70ms and then decrease to 67ms when setting the cache size to 4000. This behavior is due to the number of page faults the system makes to swap memory pages with the virtual memory and that is because of limited memory on this machine and limited heap allocated to JBOSS.
Table 4.12 shows the page fault recordings of each test. It is noticed that increasing the request size results in increase in page fault. Increasing the cache size doesn’t have a clear impact on the page fault.

Table 4.12. Page Fault Recordings of the Test in KB

<table>
<thead>
<tr>
<th>Cache-size</th>
<th>Request-size</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>94</td>
<td>184</td>
<td>241</td>
<td>478</td>
<td>1352</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>104</td>
<td>159</td>
<td>277</td>
<td>748</td>
<td>1305</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>109</td>
<td>162</td>
<td>255</td>
<td>662</td>
<td>1170</td>
<td></td>
</tr>
</tbody>
</table>

4.2.3.4.3 Increase Number of Threads on a Fix Request Size

QSM manages many queues for many different users. Each user might set up multiple producers and consumers for each of their queues. This will result in more hits on QSM per second. This test set-up helps to measure the system latency when increasing the number of components that hits the server at once. The test can be accomplished in JMeter by increasing the number of threads and fixing the size of requests to 1000 that each thread is making; the threads will have no delays to increase the possibilities of simultaneous hits on the server. Table 4.13 shows the results of this test. All measurements show more latency as the number of threads (i.e. components) increase which sets up a signal to load balancing by using multiple QSM components to receive requests simultaneously.

Table 4.13. Results of Measuring Latency When Number of Threads Increase

<table>
<thead>
<tr>
<th>Threads Measure</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Max</td>
<td>131</td>
<td>201</td>
<td>457</td>
<td>1770</td>
<td>2948</td>
<td>4970</td>
</tr>
<tr>
<td>Average</td>
<td>34</td>
<td>65</td>
<td>133</td>
<td>305</td>
<td>586</td>
<td>1088</td>
</tr>
<tr>
<td>Throughput</td>
<td>25.6</td>
<td>28.4</td>
<td>28.8</td>
<td>25.3</td>
<td>25.7</td>
<td>27.8</td>
</tr>
</tbody>
</table>
4.2.3.4.4 Bring Servers Down While Sending Requests

QSM forwards requests to all queue servers in the subset. If one or more queue servers are down, QSM will keep the request in a placeholder until the queue server comes back on line, then QSM will start to push all missed requests before marking the queue server as available. Queue servers going off line and then coming back up might have an impact on the latency of the responses sent back to the user’s request. This test will try to randomly shut down queue servers and bring them back up while the system is continuously receiving requests; the test ensures the availability of at least one queue server at any given time. The test has 10 simultaneous threads; the test will start with 1000 requests doubling the number of requests until reaching 32000 requests. Threads will sleep randomly between 0 and 1000 milliseconds between requests. The results of this test, as expected, show no increase in latency when one or more of queue servers are unavailable as shown in Table 4.14.

Table 4.14. Results of Measuring Latency When Shutting Down Queue Servers and Bringing Them Back Up

<table>
<thead>
<tr>
<th>Request-size</th>
<th>Measure</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
<th>32000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td></td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>276</td>
<td>271</td>
<td>345</td>
<td>315</td>
<td>363</td>
<td>367</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>29</td>
<td>32</td>
<td>35</td>
<td>37</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
<td>17.2</td>
<td>17.7</td>
<td>17.9</td>
<td>17.9</td>
<td>17.8</td>
<td>17.7</td>
</tr>
</tbody>
</table>

4.2.3.5 System Consistency

The distribution nature of VSQS results of an inconsistent system. Messages received by the system are duplicated into multiple servers. When QSM receives a message, it calls each queue server in the subset to save a copy of the message; similarly when a message is locked or deleted, all queue servers process the same request. A queue server might have different timing in processing a request. Therefore, a message might appear in one queue server before others, also when locking or deleting the message, even though the time difference is too small according to our implementation, but it does exist.
If we take a timely snap shot while processing a request, we can see that all queue servers are not consistent in every time slice.

When a queue server goes down, it loses all future requests making the system inconsistent. When the queue server comes back on line, it is out of sync with the rest of the queue servers, and it may have messages that are deleted and it may be missing new added messages into the queue. Depending on how long the queue server was down, it can be partially or completely inconsistent with other queue servers. The test cases we have done in section “4.2.3.2.4” proved the inconsistency in our databases of those queue servers that went down by checking the “message” table.

VSQS is a consistent system. When the queue server came back online, it received all requests that had been missed, in the same order they were received to ensure consistency. After checking the “message” table we found that the number and status of all messages were identical in all queue servers.

The process of synchronizing the queue server expands the unavailability time of the queue server which in turn depends on how many messages the queue server missed while it was down and how much time was needed to process them.

4.2.3.6 MESSAGE Properties

VSQS was implemented to retain some properties when messages are retrieved. VSQS should ensure a message is retrieved only once. The queue is created as a real FIFO where the messages are retrieved in the same order that they are received, and when a “GET_MESSAGE” request returns no message, it means that the queue has no messages.

VSQS puts the messages in the queue in the order they are received. When retrieving the message, VSQS tries to keep the same order. To test this property, the test uses only one thread and works as a consumer; the test will record the “sequence” of the retrieved message and checks it against a fixed counter. The following test scenarios were conducted, and all show that VSQS retains the order of the messages:

- All queue servers are available; the consumer will delete messages after retrieval.
- One or more queue servers randomly go down; however, ensuring the availability of the system, the consumer deletes the messages after retrieval.
- One consumer thread is getting messages while a producer thread is putting new messages; the consumer deletes the messages after retrieval.
Each “GET_MESSAGE” request must result in retrieving a message as long as there is a message in the queue; the response returns no message only when the queue is empty. The test prepared to test this property uses only one thread that works as a consumer. The thread runs infinitely to retrieve messages. The test stops when any request returns no message in the response; once the test stops, the queue server “message” table is checked for any left messages. Our test shows that the response of each request returned a message; the test stopped after a first response was received with no message included. The “message” table is checked and it was empty of all messages.

A message saved in the queue is returned only once. The test uses multiple threads trying to retrieve messages at the same time; the messages retrieved were recorded and examined. The test shows that a message is retrieved only once.

The properties of message ordering and retrieving a message only once are dependent on deleting the message by the consumer after a successful retrieval. When the message is retrieved, VSQS doesn’t remove the message from the queue. Instead, VSQS locks the message and signs it as invisible for some period of time. If the consumer fails to send “DELETE_MESSAGE” requests, the message will be returned to the queue assuming that the consumer did not get the message. The message becomes available for retrieval and the message might be retrieved at a later time. The message retrieved can be out of order, or the consumer may receive the message more than once only when a message is retrieved and processed but not deleted. QSM puts none deleted messages back into the queue after invisibility time expires..

### 4.3 Testing Summary

VSQS is built based on Amazon SQS; however, it tries to eliminate SQS message property issues by using a preselected subset instead of randomly choosing a subset. This technique enables VSQS to control the subset more and allowed it to always send a message back to the consumer as long as there was at least one message in the queue. VSQS was also able to ensure delivery of a message only once, and messages were retrieved in the order they were received as long as the consumer was deleting the message after retrieval. Eliminating these issues was achieved because VSQS has more
strict consistency rules. A queue server is not involved when processing a request unless it is consistent with the rest of the queue servers. Mainly, the queue server that came back online will not be marked available until it is synchronized with the rest of the queue servers within the subset.

The main goal of VSQS is high availability, and duplicating messages into multiple queue servers gave VSQS a high availability. Few of the tests ran against VSQS show that the system is highly available and survives partial system crashes. When the subset includes more queue servers, it will help in making the system more available. In some other cases, VSQS became unavailable when all queue servers in the subset were either down or in a synchronization state. The queue server goes into a synchronization state after it comes back online. The period spent in synchronizing depends on the down time of the queue server and the rate of requests the system receives. The longer the down time, the higher the rate of messages expand the synchronization period, and in turn extend the unavailability time of a queue server.
CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

Today’s software systems consist of multiple components that can reside in a single machine; however, most are built to have the flexibility to be installed in different machines. The different components of a system usually interact with each other to process a faction where one or more components collaborate to process a request. A simple way of communication between components is illustrated in Figure 5.1. To process a request, component “A” finishes its part and then makes a direct call to Component “B” to perform its part of processing the request. A straight forward example of this communication is when a class of component A calls another class of component B. This type might work for small systems where all its components are in the same machine and where each component can easily locate the other component. There is a great deal of coupling in this scenario.

![Figure 5.1. Direct communication between components “A” and “B.”](image)

The direct communication between components might be sufficient for small systems where all components are deployed within one application. However, this may not be the case for almost all today’s systems. The queuing mechanism is used to help decouple the components of a system; a queue is created as a place holder for messages going from component “A” to component “B” as illustrated in Figure 5.2.
The software industry offers many queuing systems to assist users to send messages between the components of their system. Examples of those queuing systems are Web Sphere MQ (formerly MQ Series) and Oracle Advanced Queuing (AQ); some other systems are open source like JBoss Messaging and ActiveMQ. All these systems support both domains of sending and receiving messages which are point to point and publisher-subscriber. Users of any of these systems usually need to perform some configuration setup and implementation of interfaces to integrate with the queuing system they are using. Users need to do more work if they want to ensure the availability of their queues. Some queuing systems like ActiveMQ offer some assistance to their users, but still require some setup to be done by the users.

5.1 Amazon SQS

Amazon SQS takes a different approach in providing a queuing system that requires neither pre-configuration setup nor implementation of interfaces; users can create their queues and manage them using a web service API provided by Amazon SQS system. Users create their queues and start push and retrieve messages on that queue immediately. The main goal of SQS is to provide high available queues to its users. SQS achieves this
goal by distributing messages into a subset of servers. The subsets are chosen randomly based on “random weighted distribution.” Every message can go to a different subset even if they belong to the same queue for the same user. Duplicating a message increases its availability; in that, if one of the servers in the chosen subset goes down the other servers still have a copy of the message and it can be returned to the users when they try to retrieve a message.

CAP theorem states that a distributed system can have only two: availability, consistency and partition tolerance. Amazon SQS sacrifices consistency in favor of availability and partition tolerance. The distribution nature of Amazon SQS and inconsistency of the queues it holds introduce a set of issues that users need to be aware of since SQS might not fit the needs of their system. These issues are:

- A message can be retrieved more than once.
- SQS doesn’t guarantee the messages retrieved in the order they were received.
- A retrieved message request that returns no message does not necessarily mean that the queue is empty, especially when the queue has 1000 or less messages.

The advantages of using Amazon SQS are the high availability of queues and the messages they contain. Users of SQS don’t need to worry about maintaining the queue, and they don’t need to perform any configuration. However, SQS might not be a good choice for many users. Below is a discussion of SQS limitations.

SQS documentation talks about the high availability of a queue but never mentions the high availability of messages in the queue. There are some cases where a queue holds messages; however, they are not returned to the consumer, and this can happen according to SQS documentation if the queue has less than 1000 messages. The queue is available and it can process requests like adding a message to the queue; however, the queue is not available to retrieve messages. The other case is where a message is received and distributed into multiple servers. If all these servers go down, the messages in those servers become unavailable. However, the user still sees that the queue is available, and they can put new messages in and possibly retrieve them.

SQS doesn’t guarantee the order of retrieved messages to be the same order they were received. Users need to put some sequencing information into the message they put
into their queues. Adding sequencing information into messages is easy if the user system has only one producer and one consumer; however, it adds a lot of complexity if the user system has multiple producers. A sophisticated sequencing generator is necessary to provide unique sequencing information for every single message pushed by every producer; also, having multiple consumers requires having some type of internal communication to track the order of messages retrieved by individual consumers.

SQS states that a message can be retrieved multiple times. This property may not raise a red flag if the consumer component of the user’s system is implemented in a certain way so that processing the same message more than once has no negative impact.

5.2 VSQS

This thesis project introduces a queuing system that is derived from Amazon SQS. The system is called VSQS. The main difference between VSQS and SQS is that VSQS creates subsets of queue servers. Each queue created will be assigned to only one subset, and all messages that arrive for that queue are saved in that specific subset. A subset can contain multiple queues for possible different users. The reason for making this decision is to have better control over where messages are actually saved. This might help in reducing the issues found in using Amazon SQS, such as retrieving a message multiple times, out of order retrieving of a messages or sending no message as a response to a retrieve request when there is actually a message in the queue.

The disadvantage of having a pre-selected subset for each queue is losing the flexibility to distribute the load of messages into all available servers. Amazon SQS uses a weighted random distribution algorithm to distribute the load of messages into all servers available to hold messages by dividing them randomly into subsets for each message that arrives to the system. Consequently, two messages for the same queue might reside in a totally different subset of servers. VSQS, on the other hand, saves messages of the same queue into the same exact subset of servers and so it is very possible to have one subset filled with messages while other subsets are not. A subset might reach a point of not being able to hold more messages, if producing messages is faster than consuming them. The other disadvantage of having a preselected subset is that if all the queue servers go down or became unavailable then the system will start rejecting user’s requests, waiting for at
least one queue server to be on line or finished syncing with all missed requests while it was down. The phenomenon of having all queue servers unavailable within a subset causes the queue to be unavailable which defeats the purpose of building such a system.

Managing a queue in one subset was achieved by having a queue system manager (QSM) component for the user to communicate with. QSM then forwards the request in a certain way depending on the request type to the queue servers to process the request. QSM controls all queue servers it manages and it knows when a queue server goes down and when it comes back on line which helps in bringing the queue server in sync with the rest of the queue servers in its subset.

5.3 VSQS TEST RESULTS

Unit and functional testing shows that VSQS performs all functions that are supported properly. Under normal cases the average response time of PUT_MESSAGE is around 26ms, GET_MESSAGE is around 149ms while Delete_MESSAGE is around 35ms. The system didn’t record any missing operation; all requests were processed successfully.

System availability testing shows that the system stayed available when making push and retrieve message requests as long as at least one queue server was up and running. When all queue servers became unavailable VSQS started rejecting user’s requests. This is the first issue that arose from having a pre-selected subset of a finite number of queue servers. Even though having all queue servers go down at the same time is unlikely to occur, it is a possible scenario. When a queue server comes back online it spends some time to sync up with the other queue servers in the subset which increases the unavailable period. In particular, this period might take a longer time when the system keeps getting more requests. In some cases the server might not reach the completion of synchronization which results in having an unavailable queue server. The recommendation section will shed light on finding a cure for this issue.

System latency testing shows that bringing some of the queue servers down didn’t add noticeable latency to the response time. Increasing the number of consumers that hits the system simultaneously adds a great deal of latency to the response time. Doubling the number of consumers has a linear impact on latency. To keep the average response time
within an acceptable range we need to load balance requests among multiple queue server managers (QSMs). Changing the cache size didn’t help in measuring its effect on GET_MESSAGE response time. Even though the average response time stayed within an acceptable range, the higher max response time recorded as well as missing the 90% threshold of most get message tests is an indication that the decision of implementing the in-memory cache didn’t help in saving time on message retrieval.

VSQS is designed to sacrifice inconsistency in favor of availability and partition tolerance. Our tests, however, show that VSQS is an eventually consistent system. When one or more queue servers goes down, it becomes inconsistent with other queue servers; it misses all requests that the system has received while it was down. When the queue server comes back on line, QSM starts sending all the missing requests, and the queue server eventually becomes consistent with the rest of the queue system after it receives and processes all missed requests.

VSQS retrieves a message and sends it back to users only once, as long as the user confirms receiving and processing a message by deleting it. Also a get message request returns a message every time as long as there is a message in the queue; the response contains no message only when there is no message in the queue. Our tests also show that VSQS returns the messages in the order they were received.

5.4 CONCLUSION

In conclusion, VSQS is a highly available queuing system given that at least one queue server of a subset is up and running at any given time. The decision made of pre-selecting the subset of queue servers helped to retain messaging properties and so a message will be returned to the user only once. Messages are returned in the order they were received, and if a response to get message request returns no message, that means the queue has no more messages for that queue and the user doesn’t need to make consecutive requests. However, pre-selected subsets increased the chances of unavailability of the system that can occur when all queue servers in the subset are either down or in a synchronization state.

Having in-memory caching system to hold messages for fast retrieval made the average response time reasonable but inconsistent. Fifty percent of the responses to get
message requests stayed within the average; however, the other 50% are unpredictable and some might reach 2 to 2.5 seconds. Changing the size of the cache didn’t help normalizing the response time.

5.5 Recommendations

VSQS is built for academic research only. The tests conducted as shown in chapter 4 indicated some necessary changes to be done to VSQS components to improve its performance and make the system more consistent and stable. Below are the recommended changes to VSQS:

- Currently the subset consists of a fixed number of queue servers; the queue servers are identified and remain the same. When one or more of the queue servers go down, the subset will consist of a smaller number of queue servers than what it started with; the shortage in the number of queue server stays until the unavailable queue servers comes back on line. This design increases the chances of unavailability of the system. It is recommended to replace the unavailable queue server with others immediately to keep the number of queue servers almost fixed.

- A decision was made to have an in-memory cache system to hold a certain number of messages and keep them in memory for fast retrieval when a `get_message` request arrives. Having such a caching system might help in reducing the response time for 50% of the requests to below 100 milliseconds, it but made the remaining 50% unpredictable and the response time of some requests might go as high as 3 seconds. It is recommended to either re-work the implementation of the caching system or replace it with In-Memory database. **MySql** database used in this project offers memory capability.

- Increasing the number of hits per second on the system increases the latency of response time. It is recommended to load balance the requests among multiple queue server managers.

- System Manager SM is not implemented. It is recommended to implement this component to have a complete system. This component needs to be able to manage users and receives requests to create queues and return the proper information back to the users.
BIBLIOGRAPHY


