5-2013

Developing a Mobile-Commerce Financial Transaction Processing Model

Edward Nathaniel Thomas Charles Williams Jr.
Columbus State University

Recommended Citation

Williams, Edward Nathaniel Thomas Charles Jr., "Developing a Mobile-Commerce Financial Transaction Processing Model" (2013). *Theses and Dissertations*. 36.
https://csuepress.columbusstate.edu/theses_dissertations/36

DEVELOPING A MOBILE-COMMERCE FINANCIAL TRANSACTION PROCESSING MODEL

Edward Nathaniel Thomas Charles Williams, Jr.

Columbus State University

TSYS School of Computer Science

The Graduate Program in Applied Computer Science

**Developing a Mobile-Commerce
Financial Transaction Processing Model**

A Thesis in

Applied Computer Science

by

Edward Nathaniel Thomas Charles Williams, Jr.

Submitted in Partial Fulfillment

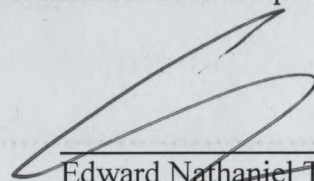of the Requirements

for the Degree of
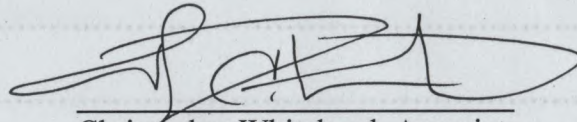
Master of Science

May 2013

I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science.

_5/2/2013_
_____
Date
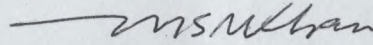
Edward Nathaniel Thomas Charles Williams, Jr.

We approve the thesis of Edward Nathaniel Thomas Charles Williams, Jr. as presented here.
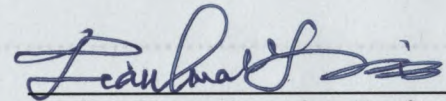
_4/29/13_
_____
Date

Christopher Whitehead, Associate Professor of Computer Science, Thesis Advisor
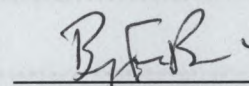
_5/1/13_
_____
Date

Shamim Khan, Professor of Computer Science

_05/01/2013_
_____
Date

Jianhua Yang, Associate Professor of Computer Science

_5/1/13_
_____
Date

Benjamin Blair, Butler Chair in Business and Finance

_5/2/2013_
_____
Date
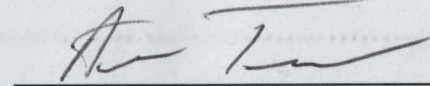
Alan Tidwell, Assistant Professor of Accounting and Finance

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgements

The author wished to thank several people. I would like to thank my mom, Dr. Bonita Williams for her love and support while completing this thesis. I also wish to thank Dr. Christopher Whitehead for his guidance and assistance while competing this thesis. Last, but not least, I would like to thank Ramon Perez from First Data for his assistance in accessing a payment platform for my project.

Abstract

The topic for this Master's Thesis is selected in compliance with the guidelines to complete a Master of Science in Applied Computer Science at Columbus State University. The problem to be addressed by this thesis is to produce an open standard for an m-commerce financial transaction processing system based on current e-commerce standards and mobile technology. This solution was to be specifically designed to build upon the strengths of a mobile platform using current smartphone and tablet technology.

An open source software stack in combination with a cloud computing solution was used to create a working example of the specification. Load testing and Denial of Service attack testing were completed to test the stability and capacity of the implementation. It was found that the initial implementation of the specification was able to accommodate a moderate level of concurrent transactions and connected users. It was also found that the system was brought down with a slow header denial of service attack, but was able to withstand a slowloris denial of service attack. An Android native application was built as a sample implementation of a mobile client for the system.

Chapter 1: Introduction

Mobile technology has rapidly evolved over the last 20 years. As a platform, mobile devices have evolved their functionality, beginning with basic voice communication and later adding text-based communication. As mobile devices gained additional capabilities, data and world wide web (WWW) functionality was added. "Not so long ago, low-speed, small screen mobile devices on cellular networks accessed the Internet with minimal functionality via the Wireless Application Protocol (WAP). Now, wireless devices are much more powerful, in some cases as powerful as the PCs of just a few years ago" (Vaughan-Nichols, 2008). With the addition of these capabilities, electronic commerce became a user-friendly function available on mobile devices.

1.1 Electronic Commerce (E-Commerce)

1.1.1 Definition

Electronic commerce (E-Commerce) is defined as "any type of business or commercial transaction that involves the transfer of information across the Internet" (Maamar, 2003). These transactions can take place between a business and person, a business and another business, or any combination of people and businesses.

1.1.2 Background

E-commerce has evolved through several major changes. First, businesses started with the digitization of their data to make it available online. An example of this would be a text-based representation of what was previously a paper-based catalog utilizing a service such as Compuserve or Prodigy. Later, businesses decided to reengineer this process to stay competitive. This change included increasing the ease of access to data and the consistency of the data presented. An example of this change would be the

movement of data to webpages, where standards such as HTML were used. The third

change that occurred was the offering of online forms to capture users' needs efficiently

and accurately. This involved inviting financial partners to join the shopper-vendor

relationship. At this point, this allowed a user not to just view information, but to

actually make a purchase from within the website. Ensuring security for the payment

process and the exchange of private information became a concern, as now sensitive

financial information was being transmitted and stored.

The next stage in e-commerce was the personalization of services. This involved the

introduction of user profiles, which included preferences and interests. The e-commerce

site would then adapt itself to meet the customer's needs based on the user profile. Joint

business ventures and social contexts have also been introduced into e-commerce

(Maamar, 2003). Good examples of this would be a current website like amazon.com.

Standards such as PCI (Payment Card Industry) were created to ensure a standard

level of security when handling financial transactions electronically and storing the

transactional credentials of the customers in an implemented storage system. Visa and

Mastercard required higher standards of security from the payment card services industry,

which created the need for standards such as PCI. The standard applies to the protection

of credit card data that is processed, transmitted or stored. They have come up with 12

rules that must be complied with. The requirements include encrypting card data across

public networks and using anti-virus software. These standards also include advice on

how to develop software securely (Mathieu, 2006).

1.2 Mobile Commerce (M-Commerce)

1.2.1 Definition

Mobile Commerce (M-Commerce) is defined as the financial transactions for services or goods between trading parties through a mobile terminal (Weihui, Xiang, Haifeng, Weidong, & Xuan, 2011). More precisely, a mobile terminal is defined as a smartphone or tablet with Internet capabilities and software to enable financial transactions. M-Commerce is considered by some to be the next level of evolution of e-commerce. M-Commerce is an emerging discipline involving applications, mobile devices, middleware, and wireless networks. M-Commerce involves using mobile devices such as smartphones and tablets to complete financial transactions.

1.2.2 Background

While most existing e-commerce applications can be modified to run in a wireless environment, m-commerce also involves many new applications that are only possible due to wireless infrastructure (Malloy, Upkar & Snow, 2002). As an integration product of electronic currency and mobile communication, mobile payment has many advantages. First, it is more convenient and easier to use than other traditional payment methods. Second, it has a higher level of compatibility due to the smaller number of providers (Haifeng , Xuan, Weihui, & Weidong, 2010).

Chapter 2: The Mobile Commerce Problem

2.1 Standard Definition

For years, techies and phone companies have dreamed of turning a cell phone into a virtual wallet (Crockett, 2005).   Although there are several competing systems available for consumer use today, each having advantages and disadvantages, there are no standards established for how an m-commerce system should be implemented. The problem is that all of these systems are proprietary and closed. Another issue in the solutions available today is that these solutions are locking into individual platforms and will not work on competing mobile operating systems. An example of this is Google Wallet. The Google Wallet app is only compatible with a small number of Android based devices. The Google Wallet app is not available for the iOS, Windows, or Blackberry mobile platforms.

2.1.1 Specification

A standardized, open specification is needed for m-commerce systems. Current e-commerce and m-commerce designs need to be investigated, and a standardized model for how to build an m-commerce transaction processing system needs to be proposed, tested, and more widely implemented.

2.1.2 Security

Due to the sensitive nature of the data that a mobile commerce connection processes, a certain level of security must be added to the standard. This security will include several levels of data transferred across the network and the use of firewalls in the network to prevent the unauthorized use of that network.

2.1.2.1 PCI

A standard that is in use today across the financial sector is the PCI security standard. There are several subsections of this standard, including the Data Security Standard (PCI DSS) and the Payment Application Data Security Standard (PA-DSS). The standard developed as part of this thesis includes the implementation of the PCI DSS and PA-DSS standards. If a third party certifies the system as being compliant with both of these standards, the mobile commerce system should be acceptable for production and commercial usage.

Chapter 3: Solution

3.1 Mobile Commerce Open Standard

"Open-source software development is a production model that exploits the distributed intelligence of participants in the Internet community" (Kogut & Metiu, 2001). The currently available solutions are closed-source and proprietary. There are no mobile commerce platforms available today that are open-source and available on all of the major mobile platforms. An open-source standard for a mobile commerce system must be created to resolve this issue.

A mobile-commerce system will require several components to be functional. The system must be able to handle large amounts of transactions over the Internet, it must be secure, and it must be reliable. There are technologies that are already in use in financial and e-commerce systems that can be utilized to build the open source standard. Tokenization, cloud computing, open source software, and open source operating systems will be included in the standard.

"Tokenization involves the random generation of proxy numbers to replace actual credit card numbers at the point of sale to improve data security" (Heun, 2011). Tokenization technology replaces a primary account number with a surrogate value called a "token". If a token is properly used, then there would be no need retain the primary account number in the payment system (Heun, 2011). Tokenization is a method by which a primary account number (PAN) is associated with a reference number where a merchant only needs to keep the token and a trusted third party keeps the PAN and manages the association (Stapleton & Poore , 2011).

"A cloud is a pool of virtualized computer resources" (Pareek, 2001). A cloud can host a variety of different workloads, such as batch style back-end jobs and user-facing client applications. Examples of this are a payroll processing system or an e-commerce website such as amazon.com. Cloud computing allows an application to be deployed and scaled out quickly through the rapid provisioning of virtual machines or physical machines; allows for support of redundant, self recovering highly scalable programming models; and monitors resources in real time to enable reallocation of resources when needed (Pareek, 2011). Rapid provisioning is the process of deploying a server image onto a server through software such as VMWare or Amazon's EC2 interface. This process can be done in minutes, versus the hours or days it can take to complete a traditional server deployment. The utilization of a cloud computing system will be ideal for the implementation of the mobile commerce specification, as a mobile commerce system will require a system that is reliable, can handle large amounts of real-time transactions, and can reallocate resources dynamically as needed.

"Open source software lets users study, modify, and redistribute the source code" (Nash, 2009). With access to open source software and operating systems, there is a robust collection of applications that are required to build a mobile commerce processing system. Webserver software such as Apache server, application servers including JBoss and Tomcat, and database solutions like MySQL are components that can be utilized as an open source implementation of the specification.

3.1.1 Mobile Commerce Specification Definition

The initial definition defined includes the following components:

- System Architecture

- Network Architecture

- Server Software

- Database Definition

- Mobile Client Specifications

- Mobile Commerce Application Specification

When all of these components are implemented and combined, a base model of a mobile commerce system is created. The system architecture describes the overall server, network, and software requirements. The network architecture describes how the network is configured for the specification. The database definition is a description of the tables required for the system functionality. The mobile client specification describes the functionality of the mobile client software. Finally, the application specification describes the application that implements the required back-end functionality of the system.

3.1.1.1 System Architecture

"Web-based e-commerce applications commonly employ multiple tiers (3-tier client server architecture) and a combination of technologies such as HTML, XML, JavaScript, Java (JSP, Servlets), ASP, dynamic html, CGI, and relational databases" (Meshram & Rane, 2012). The proposed system architecture shall consist of a 3-tier architecture. The top layer will contain multiple webservers, which will be load balanced by a top-level load balancer. The top-level load balancer will receive traffic from the mobile clients over the Internet and divide the traffic between the webservers. The middle layer will consist of several servers that will run the mobile commerce application. This layer will be load balanced by a load balancer connected between the top and middle tiers. The

database component of the system will be located in the bottom tier of the system. A

replication method that will make multiple copies of the database simultaneously will be

implemented.

Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

>GET SALES TOTAL

>GET SALES TOTAL
4 TOTAL SALES

GET LIST OF ALL SALES MADE LAST YEAR

ADD ALL SALES TOGETHER

QUERY

SALE 1
SALE 2
SALE 3
SALE 4

Database

Storage

Figure 3.1: 3 Tier Architecture Diagram

A common technology stack should be utilized to simplify and to keep the

implementation consistent. This should be done, as it is a normal practice to implement a

technology stack using common technologies that were developed to function together.

For example, an IIS webserver should not be used in combination with a JBoss

application server. The recommendation is to utilize all Java-based technology in one

stack, or to use all Microsoft technology in the implementation stack. Common examples

of this type of implementation would be an IIS webserver instance, a Windows Server

instance running IIS as an application server, and a Microsoft SQL webserver instance.

The Java-based example of this implementation would be an Apache webserver instance,

a JBoss or Tomcat application server instance, and an Oracle, MySQL, or IBM DB2

database server instance.

3.1.1.2 Network Architecture

A three-tier networking architecture was the standard utilized for this implementation. Load balancers were used to ensure that an even amount of transactional data flows to and from the available web, application, and database servers.

Firewalls were implemented between each of the tiers and in between the Internet and the top tier. Only the ports required for communications between the layers were opened, all other ports were closed. This network can be implemented either with an internal network, enterprise network, or with a cloud service, such as Amazon Web Services (AWS) or Microsoft Azure.

3.1.1.3 Database Specification

The underlying database for the system was implemented using the relational diagram and table definitions shown below.
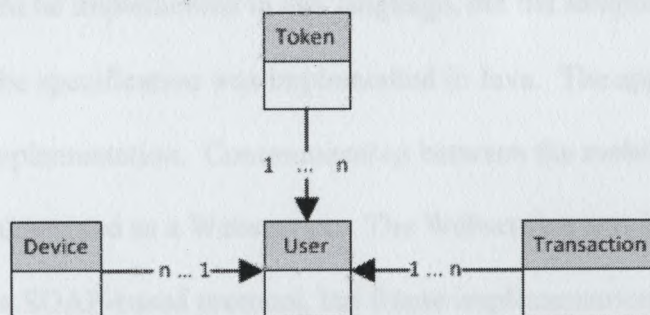
Figure 3.2: Database Relational Diagram

Table Transaction

id	timestamp	amount	transactionType	tokenId	authNumber	result

Table 3.1: Transaction Table Definition

Table Device

id	deviceId	userId	status

Table 3.2: Device Table Definition

Table User

id	username	password	name	address	city	state	zipCode	phone	email

Table 3.3: User Table Definition

Table Token

id	tokenNumber	tokenExp	userId	cardType

Table 3.4: Token Table Definition

3.1.1.4 Application Specification

The application was designed to handle communication between the mobile client and the financial processor. The application managed the storage of transactional data. The application could be implemented in any language, but the sample of the implementation of the specification was implemented in Java. The application itself ran on a JBoss server implementation. Communication between the mobile client and the application was implemented as a Webservice. The Webservice was initially implemented using a SOAP-based protocol, but future implementations will support a Representational State Transfer (REST) implementation, utilizing JavaScript Object Notation (JSON) to encapsulate the data transferred.

"JSON is a popular format for data serialization. Programmers use it extensively to encode data for transfer between a server and an Asynchronous JavaScript and XML (AJAX) application, to connect two servers communicating via Web services, and in many other similar scenarios. The most common structures used in programming are scalar variables, linear lists, and key-value pairs. JSON represents these structures in the

most natural and direct serialization, greatly reducing the impedance mismatch between in-memory structures in applications and the serialization format. JSON is not only convenient but also efficient" (Severance, 2012).

The webserver was responsible for implementation of the SSL protocol to encrypt the data between the mobile client and the server m-commerce gateway. Client certificates were implemented along with the SSL certificate to employ an additional layer of security.

The server application implemented the following functionality:

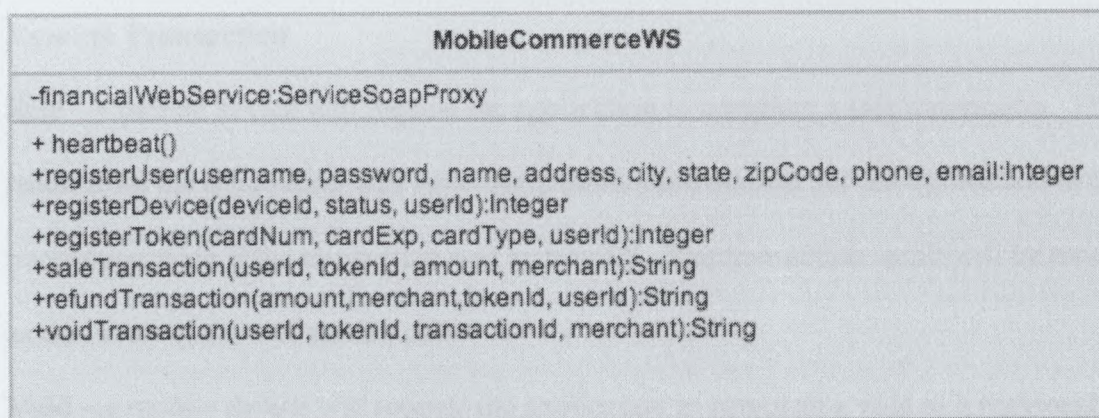| MobileCommerceWS |
| --- |
| -financialWebService:ServiceSoapProxy |
| + heartbeat()
+registerUser(username, password, name, address, city, state, zipCode, phone, email:Integer
+registerDevice(deviceId, status, userId):Integer
+registerToken(cardNum, cardExp, cardType, userId):Integer
+saleTransaction(userId, tokenId, amount, merchant):String
+refundTransaction(amount,merchant,tokenId, userId):String
+voidTransaction(userId, tokenId, transactionId, merchant):String |

Figure 3.3: Mobile Commerce Implementation UML Diagram

Authentication

Register User – a user is registered with the system. A user will be able to register multiple devices under the same username. A username must be unique and cannot be duplicated. This limitation was enforced at the database level and not at the mobile client application level.

Register Device – the mobile device will register itself with the application. Each device shall supply a unique identifier based on the device in question. The uniqueness of the identifier will be enforced at the database level. This identifier is a 50 character maximum alphanumeric string that can also include special characters.

Register Token – any credit card number associated with a user is tokenized through a third-party tokenization system. The returned token will be stored in the database and the original number will only be utilized one time to obtain the token. Credit card numbers will not be stored by the system, only the tokenized value.

Execute Transaction

Sale – a mobile device will request the application to complete a sale transaction. The result from the transaction will store the authorization number for the transaction in the transaction table if the transaction was successful. The transaction result will be retuned as the result of the webservice call.

Void – a mobile device will request the application to complete a void of a previously completed transaction. The authorization number of a previously successful sale or refund transaction will be required to execute a void command.

Refund – a mobile device will request the application to complete a refund transaction. A successful refund transaction will store the results of the transaction in the transactional database and return the results to the webservice caller.

Multi-Pay Tokens

The original concept of the token meant that the merchant could not use this random number to perform a subsequent financial transaction, because it is not a valid PAN. However, a multi-pay token adds the ability to perform an authorized financial

transaction under strict control measures within the merchant environment. The merchant

submits a token that it already has on file for a specific consumer/card to the transaction

processor who accesses an external database to retrieve the PAN and complete the

transaction. By using this type of token in the payment authorization process, the

merchant reduces the risk of having the real PAN stolen as it is being collected from the

consumer or stored by the merchant.

"Multi-pay tokens are especially valuable in e-commerce and other card not present

(CNP) environments that tend to store payment card information in a virtual wallet or on

their website for repeat customers. The multi-pay token allows a merchant to tokenize the

payment card information, associate that token with the consumer profile stored on the

merchant side, and then use the token with the processor gateway that holds the token

vault in order to run subsequent transactions. This is done without the need to prompt the

customer for his card account number again, and without having to store the actual card

number" (How Multi-Pay Tokens, 2012).

3.1.1.5 Mobile Client Specification

The mobile client will implement all of its functionality through the

implementation of the mobile commerce webservice. This revision of the specification

does not specify what the interface must look like, but only what minimum functionality

is required to be compliant with the specification.

A mobile client can be implemented in any programming language. A language

native to platform, such as Objective-C for iOS or Java for Android, is acceptable. A

fully web-based approach is acceptable, as the state of the application is handled solely

on the server side. A hybrid approach, where the interface is built upon web standards

and only portions of the code are implemented in a native language, is acceptable. An example of a framework for this approach would be PhoneGap or Xamarian. PhoneGap and Xamarian are two examples of HyperText Markup Language (HTML) based frameworks that can be used to produce mobile applications.

The mobile client must implement an interface to collect user input for use with the mobile commerce webservice. The results from the mobile commerce webservice must be interpreted and displayed in a user-friendly manner for the end-user. A platform-independent algorithm is required for the creation of a unique identifier for the devices. The Universally Unique Identifier (UUID) identifier standard was utilized as the device identifier in the implementation. The initial implementation will put no restrictions on the type of characters used for the username and passwords for the users. Alphanumeric and special characters were acceptable when creating a username and password. In the initial implementation, the password was passed without encryption, and was stored in the database without encryption. This feature is outside of the scope of the initial specifications and will be included in a later revision.

3.2 Implementation

3.2.1 System Implementation

The sample implementation used an Apache webserver for all of the webserver instances. For the operating system of the implementation instances, default configured Amazon EC2 Linux instances were utilized. To implement the firewalls, Amazon's soft firewall service was implemented. The port of 443 was the only port the top load balancer allowed traffic to travel over. The top and middle layer only allowed port 5634 to open to allow traffic between the application and the webservers. Port 2674 was the

only port opened between the middle and bottom layer, allowing data to be transferred

between the application and the database.

Figure 3.4: Amazon EC2 Server Instances

3.2.2 Network Implementation

Load balancers are one of the unique features of cloud computing and implemented in

Amazon's EC2. EC2 includes a load balancer in which an auto-scaling algorithm is used

based on threshold values of network traffic. When the setup threshold value is exceeded,

Amazon's EC2 will spin a new webserver and automatically roll it into the load balancer

pool. Similarly when the traffic falls below the threshold value, Amazon will take a

server from the allotted pool (Anandhi & Chitra, 2012). To maximize the effectiveness

of the Amazon load balancer, it is important to split the instances in the load balancer

evenly between the 4 available zones. If it is possible, instances should be added in

multiples of 4, with one instance added to each zone simultaneously.
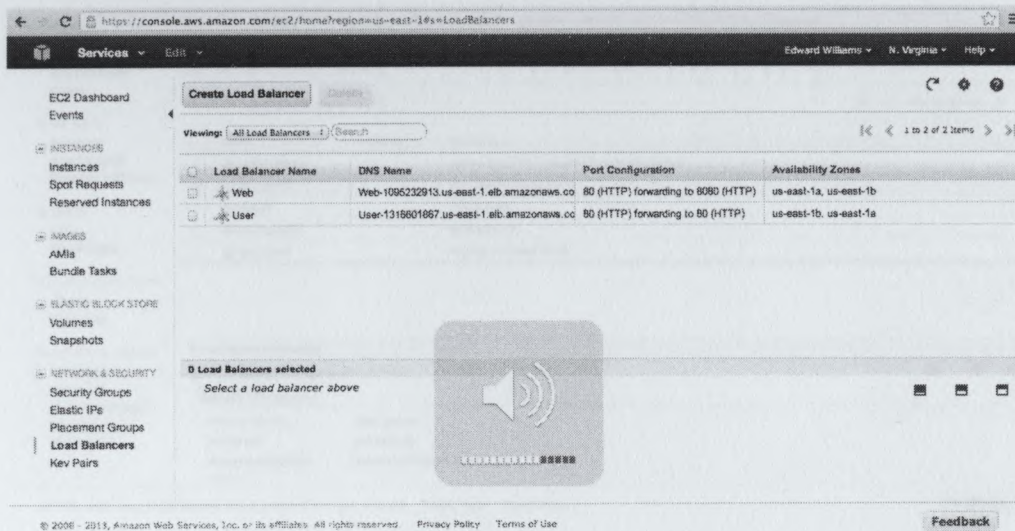
Figure 3.5: Amazon Load Balancer

"Amazon EC2 provides a complete firewall solution; this mandatory inbound firewall is configured in a default deny mode and the Amazon EC2 customer must explicitly open any ports to allow inbound traffic. The traffic may be restricted by protocol, by service port, as well as by source IP [Internet protocol] address (individual IP or classless inter-domain routing block). The classless inter-domain routing (CIDR) block was created to do away with the restrictive Class A, B, and C network distinction" (Passmore, 1994). Class A, B, and C network divisions are used to assign groups of IP addresses in a standardized fashion.

"The firewall can be configured in groups permitting different classes of instances to have different rules, for example the case of a traditional three-tiered web application" (Jamil, 2011).
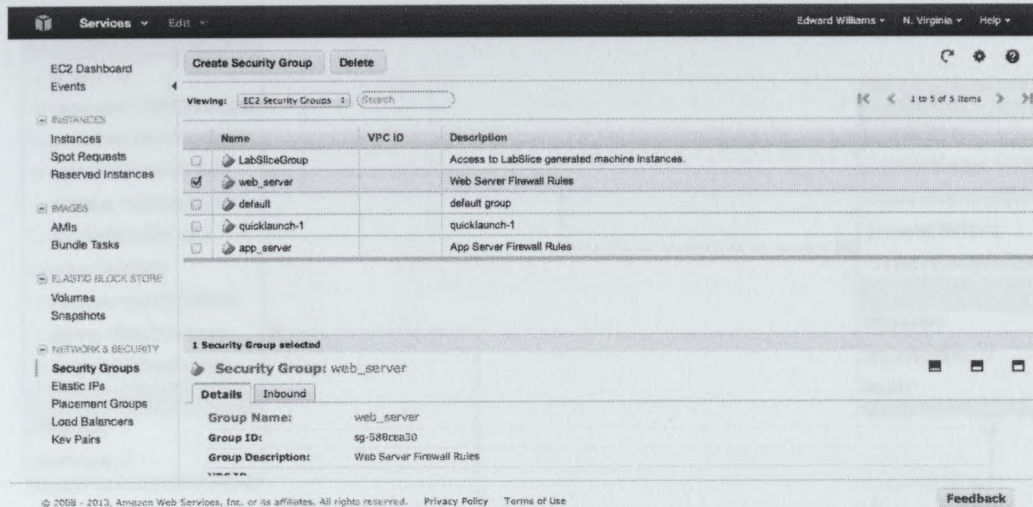
Figure 3.6: Amazon EC2 Firewall Rules

3.2.3 Database Implementation

The following is the definition for the database tables. All credit card numbers stored in the system were tokenized, a non-tokenized option was not available, and tokenization was not enforced.

As part of the standard, a standard for interconnecting the database to the application is recommended. The sample implementation used Hibernate for its database API. Hibernate is a collection of related projects enabling developers to utilize plain old Java objects (POJO) style domain models in applications (hibernate.org, 2013).

The following is the Entity Relationship Diagram (ERD) for the database implementation:

User
- id INT(11)
- username VARCHAR(50)
- password VARCHAR(50)
- name VARCHAR(100)
- address VARCHAR(100)
- city VARCHAR(100)
- state CHAR(2)
- zipCode VARCHAR(10)
- phone VARCHAR(10)
- email VARCHAR(255)

Indexes
- PRIMARY
- username_U

Token
- id INT(11)
- tokenNumber VARCHAR(50)
- tokenExp VARCHAR(10)
- userId INT(11)
- cardType VARCHAR(25)

Indexes
- PRIMARY
- tokenNumber_U
- userId

Device
- id INT(11)
- deviceId VARCHAR(255)
- userId INT(11)
- status TINYINT(1)

Indexes
- PRIMARY
- DeviceId_U
- userId

Transaction
- id INT(11)
- timestamp TIMESTAMP
- amount DECIMAL(10,2)
- transactionType VARCHAR(25)
- tokenId INT(11)
- authNumber VARCHAR(50)
- result CHAR(3)

Indexes
- PRIMARY
- tokenId

Figure 3.7: Database ERD

Amazon Relational Database Service (RDS) was used to create the database instance in the implementation. This was done so that a server instance with a MySQL database instance already pre-installed and pre-configured could be spawned quickly. By using the Amazon RDS service instead of creating an EC2 instance and manually installing a MySQL server, a Graphical User Interface (GUI) is made available for managing the server instance. To ensure that enough simultaneous connections are available for the system, a database instance of the type shown in figure 3.8 was used.

DB Instance

High-Memory Quadruple Extra Large DB Instance: 68 GB of memory, 26 ECUs (8 virtual cores with 3.25 ECUs each), 64-bit platform, High I/O Capacity, Provisioned IOPS Optimized: 1000Mbps

One ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Figure 3.8: DB Instance Specifications
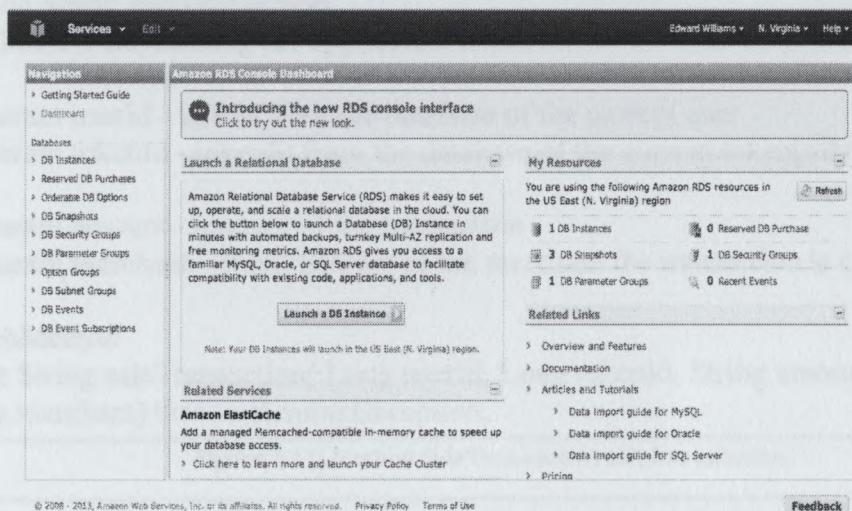
Figure 3.9: Amazon RDS Database Console

3.2.4 Application Implementation

The application communicates with the mobile client by way of a webservice. By submitting a webservice request via a POST call or SOAP request, the mobile client is able to make transactional and account requests. The following URL is the link utilized by the mobile client to communicate with the mobile commerce application:

Webservice wsdl URL: http://mobilecommerce.code-sandbox.org/MobileCommerceWS/MobileCommerceWS?wsdl

The following are the descriptions of the implemented webservice class:

```
/**
* public String heartbeat
* method for announcing the application is active
*
*/
@WebMethod()
public String heartbeat();
```

Figure 3.10: Method Heartbeat Implementation

```
/**
* public String saleTransaction
* method for announcing the application is active
*
* @param userId - userId from the database of the current user
* @param tokenId - tokenId from the database of the current token(tokenized credit
card)
* @param amount - amount of the transaction
* @param merchant - email address of the merchant the transaction is completed with
*/
@WebMethod
public String saleTransaction( Long userId, Long tokenId, String amount,
String merchant) throws RemoteException;
```

Figure 3.11: Method SaleTransaction Implementation

```
/**
 * public String voidTransaction
        * method for a void transaction type
        *
        * @param userId - userId from the database of the current user
        * @param tokenId - tokenId from the database of the current token(tokenized
credit card)
        * @param transactionId - transactionId from the database of the transaction to
be voided
        * @param amount - amount of the transaction
        * @param merchant - email address of the merchant the transaction is
completed with
        */
        @WebMethod
        public String voidTransaction(Long userId, Long tokenId, Long transactionId,
                String merchant) throws RemoteException;
```

Figure 3.12: Method VoidTransaction Implementation

```
/**
        * public String refundTransaction
        * method for a refund transaction type
        *
        * @param userId - userId from the database of the current user
        * @param tokenId - tokenId from the database of the current token(tokenized
credit card)
        * @param amount - amount of the transaction
        * @param merchant - email address of the merchant the transaction is
completed with
        */
        @WebMethod
        public String refundTransaction( Long userId, Long tokenId, String amount,
                String merchant) throws RemoteException;
```

Figure 3.13: Method RefundTransaction Implementation

```
/**
        * public int registerDevice
        * method for registering a mobile device with the system
        *
        * @param userId - userId from the database of the current user
        * @param deviceId - deviceId for the device being registered
        * @param transactionId - transactionId from the database of the transaction to
be voided
        * @param status - status of the device, true = active, false = inactive
*/
        @WebMethod
                public int registerDevice(Long userId, String deviceId, Boolean status)
        throws RemoteException;
```

Figure 3.14: Method RegisterDevice Implementation

```
/**
        * public int registerUser
        * method for a registering a user with the system
        *
        * @param username - username requested, will throw an exception if the
username is already taken
        * @param password - password requested
        * @param name - name of user
        * @param address - address of user
        * @param city - city of user
        * @param state - state of user
        * @param zipCode - zip code of user
        * @param phone - user phone number, xxxxxxxxxx formatted
        * @param email - email address of user
        */
@WebMethod
public int registerUser (String username, String password, String name,
        String address, String city, String state, String zipCode, String phone,
        String email) throws RemoteException;
```

Figure 3.15: Method RegisterUser Implementation

```
/**
        * public int registerToken
        * method for registering an tokenizing a credit card
        *
        * @param userId - userId from the database of the current user
        * @param cardNum - card number of the credit card
        * @param cardExp - card expiration date of the credit card, mmyy formatted
        * @param cardType - credit card type, valid values are 'Visa', 'Mastercard',
'American Express', 'JCB', 'Discover', 'Diners Club'
        */
@WebMethod
public int registerToken(Long userId, String cardNum, String cardExp,
        String cardType) throws RemoteException;
```

Figure 3.16: Method RegisterToken Implementation

3.2.5 Mobile Client Implementation

A native Android app was built as a sample mobile client. The mobile client was

built using the standard Android development environment, Eclipse. The app has all of

the basic system functionality. This includes the ability to register the user, register the device, register a credit card (includes tokenization), and to execute a financial transaction (sale, void, or refund).
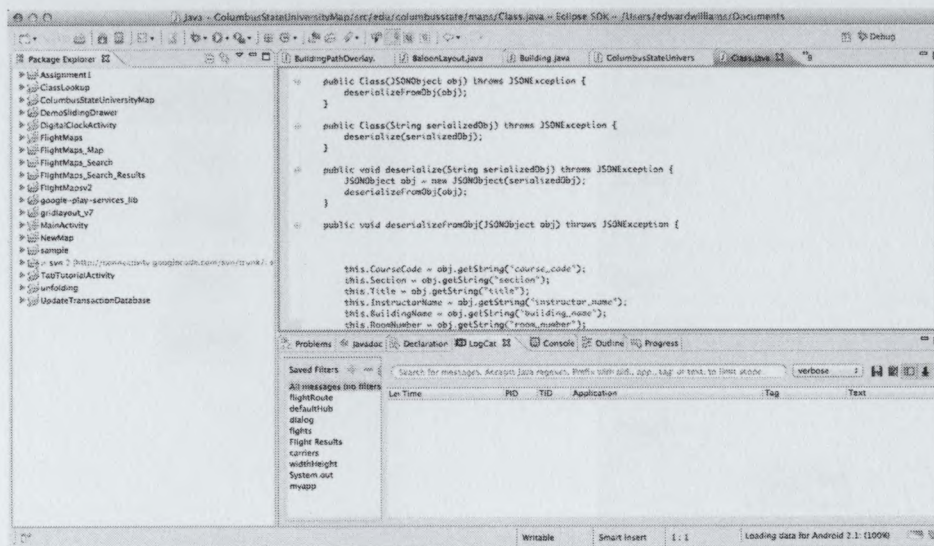

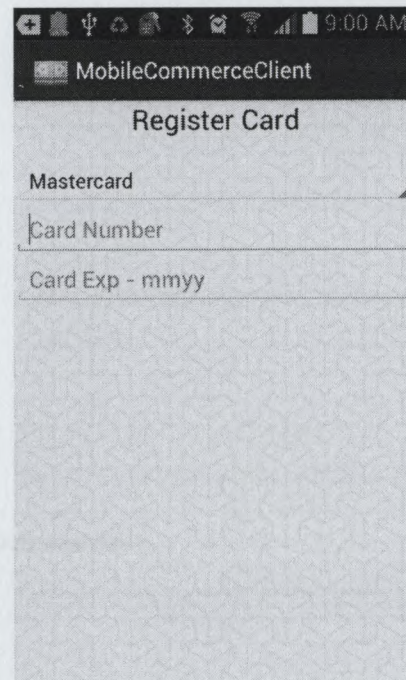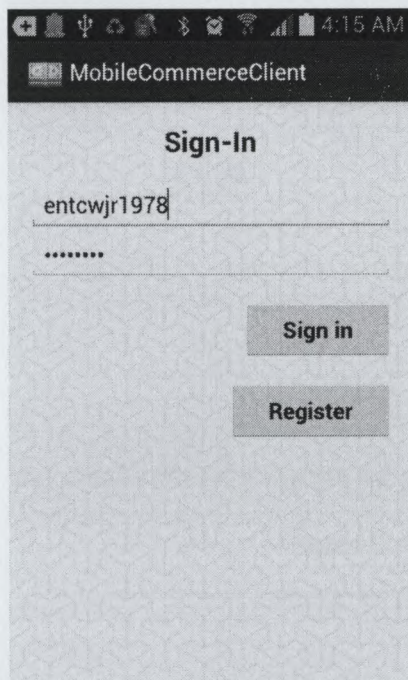Figure 3.17: Eclipse IDE, Android Development Environment

Figure 3.18: Mobile Client Screenshots

Chapter 4: M-Commerce Open Standard Evaluation

4.1 System Test

An implementation of the first draft of the standard was built for the purpose of demonstrating the capabilities of the system. Tests for the capacity of the system and standard security tests were carried out to test the capabilities of the initial implementation of the standard.

4.1.1 Load Testing

The first test is a transaction capacity test. The number of webservers were changed between 1 server minimum, and up to 4 servers maximum. The number of application servers were changed between 1 server minimum and up to 4 servers maximum. The test produced 16 sets of results. Each set of results contained the minimum number of transactions per second, the maximum number of transactions per second, and the average number of transactions per second. The tests were completed with 10,000 simultaneous users. This test was completed via the SoapUI application to simulate connecting to the service with a large number of clients, and to collect the results. "SoapUI is a free and open source cross-platform Functional Testing solution. SoapUI allows easy creation and execution of automated functional, regression, compliance, and load tests" (What is SoapUI, 2013).

The second test was comprised of attempts to compromise the sample implementation at multiple levels. At the top layer, a Denial of Service attack was committed with 1-4 webservers in service in combination with 1-4 application servers in service. The test was also comprised of attempts to access the servers in each layer directly.

4.1.1.1 SoapUI

SoapUI was used once again to simulate 25 simultaneous users connecting to the

system. The write capabilities of the system was examined by having all 25 test clients

run sales transactions, refund transactions, void transactions, and registration

transactions. These tests were done to test the write capabilities of the system.
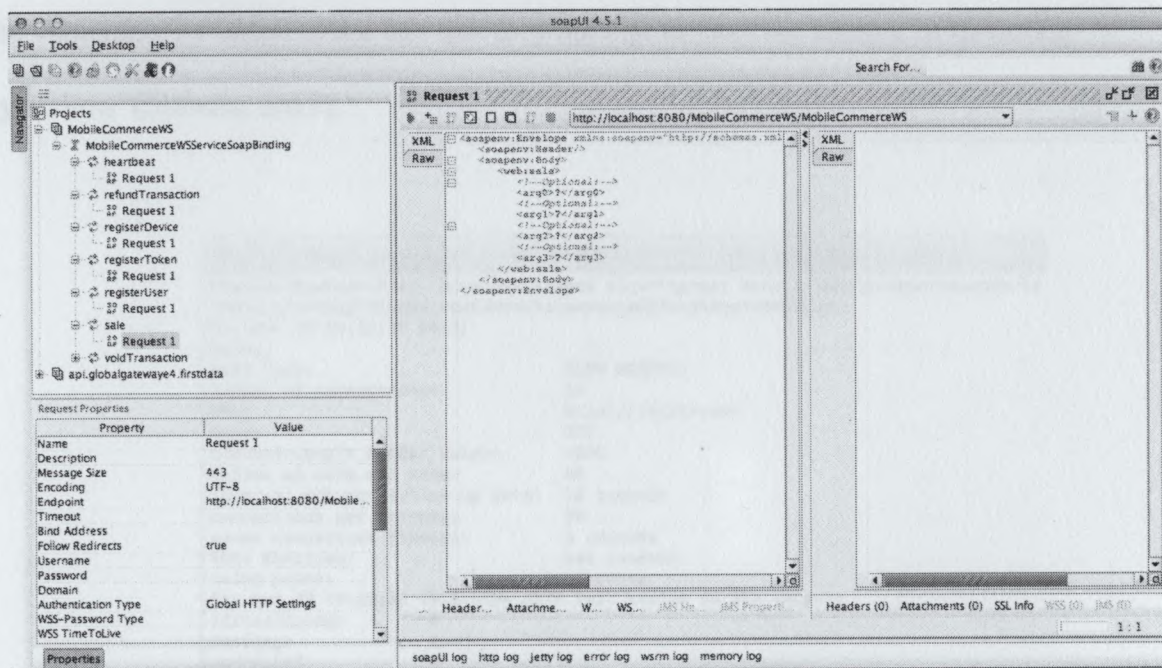
Figure 4.1: SoapUI GUI Interface

4.1.2 Denial of Service Testing

4.1.2.1 slowhttptest

A Unix line-command utility called slowhttptest was used to run denial of service

tests on the implemented system. For all tests, 5000 simultaneous connections were used.

The slow headers and slowloris tests were run over a time period of 240 seconds.

The slow headers test slows down the transmittal of the http header, which locks a

connection to the webserver for a longer than normal time period. This can create a

scenario where no connections to the webserver are available, causing the webserver to lock up and be non-responsive to the end-user.

"Slowloris holds connections open by sending partial HTTP requests. It continues to send subsequent headers at regular intervals to keep the sockets from closing. In this way webservers can be quickly tied up. In particular, servers that have threading will tend to be vulnerable by virtue of the fact that they attempt to limit the amount of threading they'll allow"(Slowris, 2013).

```
● ○ ○  ⌂ edwardwilliams — ec2-user@ip-10-202-39-45:~ — bash — 80×37
Edwards-MacBook-Pro:~ edwardwilliams$ slowhttptest http://mobilecommerce.code-sa
ndbox.org/MobileCommerceWS/MobileCommerceWS?wsdlmmerceWS?wsdl
Fri Mar 29 10:12:38 2013:
Using:
test type:                          SLOW HEADERS
number of connections:              50
URL:                                http://localhost/
verb:                               GET
Content-Length header value:        4096
follow up data max size:            68
interval between follow up data:    10 seconds
connections per seconds:            50
probe connection timeout:           5 seconds
test duration:                      240 seconds
using proxy:                        no proxy
Fri Mar 29 10:12:38 2013:slow HTTP test status on 0th second:
initializing:         0
pending:              1
connected:            0
error:                0
closed:               0
service available:    YES
Fri Mar 29 10:12:39 2013:
Using:
test type:                          SLOW HEADERS
number of connections:              50
URL:                                http://localhost/
verb:                               GET
Content-Length header value:        4096
follow up data max size:            68
interval between follow up data:    10 seconds
connections per seconds:            50
probe connection timeout:           5 seconds
test duration:                      240 seconds
using proxy:                        no proxy
Fri Mar 29 10:12:39 2013:Test ended on 1th second
status:                             Connection refused
```

Figure 4.2: Slowhttptest Interface

4.2 Test Results

4.2.1 SoapUI Load Testing Results

Web Instances	App Instances	Threads	Transaction Type	Transactions Per Second	Total Transactions	Bytes	Bits Per Second	Errors	Min	Max	Avg	Test Delay
1	1	25	Sale	8.54	523	123633	2058	0	520	7165	2109.66	1000
1	2	25	Sale	5.95	358	86278	1436	0	972	12419	3336.86	1000
1	3	25	Sale	6.43	387	93267	1551	0	846	16879	3005.06	1000
1	4	25	Sale	7.49	994	108450	1806	0	653	14437	2506.29	1000
2	1	25	Sale	14.95	900	216900	3603	0	454	4863	905.86	1000
2	2	25	Sale	15.57	937	225817	3753	0	417	6375	826.12	1000
2	3	25	Sale	15.7	943	227263	3784	0	442	2158	812.91	1000
2	4	25	Sale	16.52	992	239072	3981	0	444	2115	753.71	1000
3	1	25	Sale	17	1021	246061	4097	0	431	1527	649.1	1000
3	2	25	Sale	15.62	939	226299	3766	0	444	2566	825.12	1000
3	3	25	Sale	14.18	853	205573	3419	0	428	2845	987.88	1000
3	4	25	Sale	14.21	857	206537	3425	0	450	5527	978.79	1000
4	1	25	Sale	15.64	942	227022	3770	0	444	4356	820.79	1000
4	2	25	Sale	15.2	913	220033	3665	0	441	3860	867.66	1000
4	3	25	Sale	15.82	952	229432	3813	0	444	1988	808.08	1000
4	4	25	Sale	14.21	853	205573	3425	0	434	2647	972.37	1000

Table 4.1: SoapUI Test Results

The results of the testing yielded several interesting results. First, the number of transactions was severely limited based on the type of transaction being executed. The system is able to handle over 5000 concurrent connections when executing the heartbeat or user registration webservice commands. The heartbeat command's only function is to return a pre-determined response from the service so that the client can recognize that the application is functional and reachable over the internet. Any webservice calls made where a webservice call to the payment processor was required caused the transactions

per second to be limited to approximately 15 transactions per second. Any attempt to execute a test with more than 100 concurrent threads would overload the capacity of the test implementation and cause all four of the JBoss application servers to crash. This result was only achievable when two or more webservers were in service with the load balancer. With only one webserver in service, the application was only able to process approximately 7-9 transactions per second. By having more than one webserver, the transactions per second effectively doubled, but did not increase when a third and fourth webserver instance were added to the tier one load balancer. Adding more webservers decreased the average response time of the application in a linear fashion. The minimum response time stayed fairly constant across all tests, only the maximum response time decreased with an increase in the number of webservers put into service. The additional application servers appeared to have little to no effect on the system. With a financial transaction processor that could handle more simultaneous transactions, perhaps a greater significance in the number of application servers in service would have made a difference. One important thing to note is that even though having multiple application servers in service didn't increase the transactions per second, it does allow the system to have redundancy. All of the application instances utilized were located in different data centers (east 1a, east 1b, east 1c, east 1d). East 1a, 1b, 1c, and 1d are all located in northern Virginia. In its current configuration, the application could continue to run effectively if 3 out of 4 of the application servers failed.

4.2.2 Denial of Service Testing Results

The results for the slow body Denial of Service tests showed a loss of service availability at 32 seconds, 86 seconds, and finally at 96 seconds. The webservice had a total failure at 108 seconds.
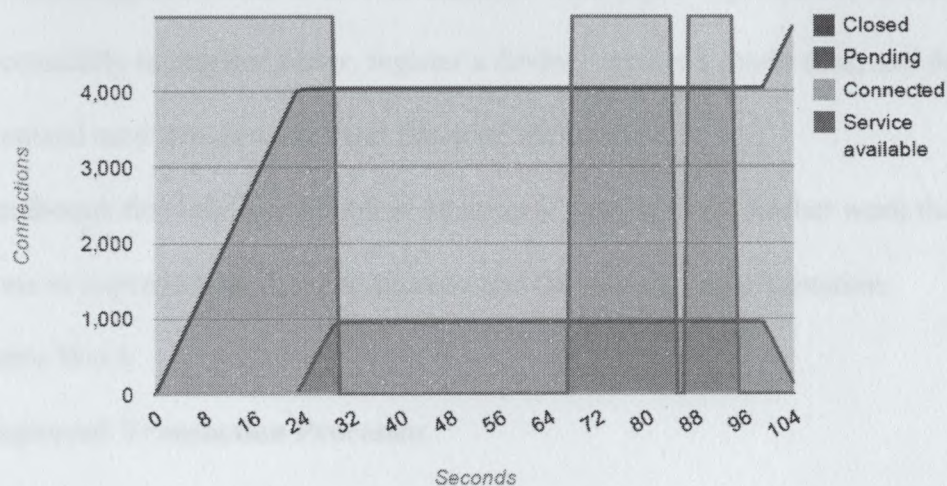

Figure 4.3: SLOW BODY Test, 5000 Connections

The slowlowris test resulted in the webservice being unavailable at several intervals during the test, occurring at 36, 72, 109, 146, 164, 182, 200, and 218 seconds. The test did run through the entire 240 second interval, so the webservers we able to withstand the test.
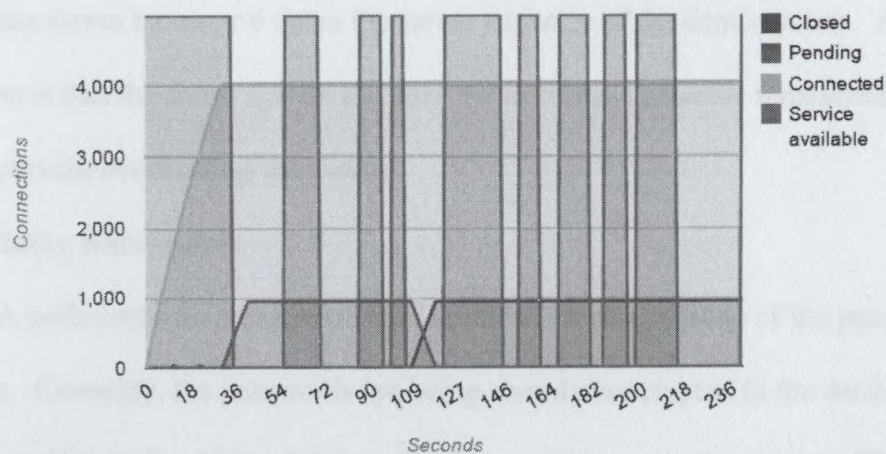

Figure 4.4: SLOW RIS Test, 5000 Connections

Chapter 5: Conclusions and Directions for Further Work

5.1 Conclusions

A base mobile wallet system was created successfully and the system was tested for load balancing and denial of service attacks. The sample implementation could be used successfully to register a user, register a device, register a credit card, and then use that tokenized card to execute several financial transactions.

Even though this implementation is functional, there is much further work that needs to be done to improve both the specification and the sample implementation.

5.2 Future Work

5.2.1 Improved Transaction Processor

The implementation of the specification for this thesis was severely limited by the performance of the transaction processor used. The account used for this implementation is a demo account and not a full production account. Using this approach allowed for a valid set of transactional responses, but the demo system's performance is severely limited compared to the production system. According to the processor, the demo system only has one webserver and one application server instance in its implementation. The production server has over 6 times the server capacity of the demo system. Another limitation is that the demo system throttles the maximum possible transaction throughput to help prevent overloading the system.

5.2.2 Adding Encryption

A sufficient encryption protocol is required for the storage of the password in the database. Currently, the passwords are being stored unencrypted in the database. Also, an SSL certificate should be added to all of the webservers in the system. This is

required to ensure that the data between the mobile device and the mobile commerce system is encrypted. Another suggestion would be the addition of client certificates. By modifying the mobile commerce system to deliver client certificates on a per-device basis, an added level of security could be added to the specification. Also, multi-layer security could be added. Requiring a password and some other secondary form of identification is an example of this.

5.2.3 PCI Compliance

Adherence to PCI standards was not implemented in the sample implementation due to time constraints and cost. A future implementation should follow these guidelines to be considered commercially usable. Assistance from a third party company that certifies systems for PCI compliance would most likely need to be a part of any sample implementations.

5.2.4 Additional Samples

An Android native client was built for this sample implementation, but in future revisions it may be a good idea to implement mobile clients for other mobile operating systems, such as iOS or Windows Mobile. Eventually, there should be a base sample available for every major mobile OS and a fully HTML-based sample.

Bibliography

Anandhi, R., Chitra, K. (2012). A Challenge in Improving the Consistency of Transactions in Cloud Databases - Scalability. *International Journal of Computer Applications*, 52(2), 12-14.

Crockett, R. O. (2005). Will That Be Cash, Credit, or Cell? *Businessweek*, (3939), 42.

Haifeng, W., Xuan, L., Weihui, D., & Weidong, Z. (2010). Mobile Payment Framework Based on 3G Network. *Proceedings Of The International Symposium On Electronic Commerce & Security Workshops*, 172-175.

Heun, D. (2011). PCI council offers tokenization advice. *American Banker*. Retrieved from http://search.proquest.com/docview/884306274?accountid=35812.

How Multi-Pay Tokens Can Reduce Security Risks. (2012). Retrieved March 21, 2012, from http://www.firstdata.com/downloads/thought-leadership/MultipayTokensWP.pdf.

Jamil, D. (2011). Cloud Computing Security. *International journal of engineering science and technology*, 3(4), 3478-3483.

Kogut, B., & Metiu, A. (2001). Open Source Software Development and Distributed Innovation. *Oxf Rev Econ Policy*, 17(2).

Maamar, Z. (2003). Commerce, E-Commerce, and M-Commerce: What Comes Next? *Communications Of The ACM*, 46(12), 251-257.

Malloy, A. D., Upkar, V., & Snow, A. P. (2002). Supporting mobile commerce applications using dependable wireless networks. *Mobile Networks and Applications*, 7, 225–23.

Mathieu, G. (2006). The PCI standard and its implications for the security industry. *Computer Fraud & Security*, 2, 6-9.

Meshram, B., & Rane, P. (2012). Application-Level and Database Security for E-Commerce Application. *International Journal of Computer Applications*, 41(18).

Nash, J. (2009). Directions for open source software over the next decade. *Futures*, 42(4), 427-433.

Pareek, R. (2011). Cloud Computing. Journal of global research in computer science, 2(7).

Passmore, D. (1994). Quick fixes for the Internet address shortage. *Business Communications Review*, 24(12), 24.

Relational Persistence for Java and .NET. (2013). Retrieved April 13, 2013, from http://www.hibernate.org/.

Severance, C. (2012). Discovering JavaScript Object Notation. *Computer (Long Beach, Calif.)*, 45(4), 6-8.

Slowris HTTP DoS (2013). Retrieved April 14, 2013, from http://ha.ckers.org/slowloris/.

Stapleton, J., & Poore, R. (2011). Tokenization and Other Methods of Security for Cardholder Data. *Information Security Journal: A Global Perspective*, 20(2), 91-99.

Vaughan-Nichols, S. (2008). The Mobile Web Comes of Age. *Computer*, 41(11), 15-17.

Weihui, D., Xiang, C., Haifeng, W., Weidong, Z., & Xuan, L. (2011). An Integrated Mobile Phone Payment System Based on 3G Network. *Journal Of Networks*, 6(9), 1329-1336.

What is SoapUI? (2013). Retrieved April 14, 2013, from http://www.soapui.org/About-SoapUI/what-is-soapui.html.