NOSQL DATA STORES

IN

PUBLISH/SUBSCRIBE-BASED

RESTFUL WEB SERVICES

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

HASHIM AL-ADHAMI

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

ABSTRACT

In the era of mobile cloud computing, the consumption of virtualized software and Web-based services from super-back-end infrastructure using smartphones and tablets is gaining much research attention from both the industry and academia. Nowadays, these mobile devices generate and access multimedia data hosted in social media and other sources in order to enhance the users' multimedia experience. However, multimedia data is unstructured which can lead to challenges with data synchronization between these mobile devices and the cloud computing back-end. The issue with data synchronization is further fueled by the fact that mobile devices can experience intermittent connectivity losses due to unstable wireless bandwidths. While previous works proposed Simple Object Access Protocol (SOAP) -based middleware for the Web services' synchronization, this approach is not efficient in a mobile environment because the SOAP protocol is verbose. Thus, the Representational State Transfer (REST) standard has been proposed recently to model the Web services since it is lightweight.

This thesis proposes a novel approach for implementing a REST-based mobile Web Service for multimedia file sharing that utilizes a channel-based publish/subscribe communication scheme to synchronize smartphone or tablet-hosted NoSQL databases with a cloud-hosted NoSQL database. This thesis evaluates the synchronicity and the scalability of a prototype system that was implemented according to this approach. Also, this thesis assesses the overhead of the middleware component of the system.

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Ralph Deters for his guidance and support. Not only has he been a mentor, but also a dear friend.

Also, I would like to thank all my committee members: Dr. Ralph Deters, Dr. Gord McCalla, Dr. Julita Vassileva and Dr. Chris Zhang for providing valuable advice and spending their precious time reading my thesis.

I would like to thank all MADMUC lab. students for their friendship and support.

Furthermore, I would like to thank my aunt, Dr. Batol Al-Adhami, for her financial and emotional support. I would not have been able to complete my studies without her extremely valuable assistance. She provided a place for me to stay at her home and spent many hours of her "resting time" proofreading my thesis.

I would like to thank my parents, Eman Al-Shaikhly and Abdulwahab Al-Adhami, and my two sisters, Zainab and Rahma, for their love and support.

Finally, I would like to thank Dr. Neil Chilton and his family for their support.

TABLE OF CONTENTS

page

Permission to Use i

ABSTRACT ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

INTRODUCTION 1

PROBLEM DEFINITION 4

LITERATURE REVIEW 8

3.1 The Model-View-Controller 8
3.2 Publish/Subscribe Systems 11
3.2.1 Publish/Subscribe Variations 12
3.2.1.1 Topic-Based Publish/Subscribe Scheme 12
3.2.1.2 Content-Based Publish/Subscribe Scheme 13
3.2.1.3 Type-Based Publish/Subscribe Scheme 13
3.3 Service Oriented Architecture (SOA) 14
3.4 Resource Oriented Architecture (ROA) 15
3.5 NoSQL Databases 18
3.5.1 Data Models 19
3.5.1.1 The Relational Data Model 19
3.5.1.2 The Key-Value Data Model 20
3.5.1.3 The Document Data Model 20
3.5.1.4 The Column-Family Data Model 21
3.5.2 The Graph Database 21
3.5.3 Examples of NoSQL Databases 22
3.5.4 Querying Mechanisms 23
3.5.4.1 SQL Statements 23
3.5.4.2 MapReduce 24
3.5.4.3 Secondary indexes 26
3.6 Summary 26

ARCHITECTURE 29

4.1 The Concept of The Model-View-Presenter 29
4.2 RESTful Publish/Subscribe Mobile Web Services 32
4.2.1 The Profile Server 34
4.2.1.1 The Profile Server's Resources 37
4.2.2 The Channel Server 44
4.2.3 The Media File's Server 48

4.3 Client Synchronization 51
4.3.1 The Local Presenter 51
4.3.2 The View 55
4.3.3The Local Model 57
4.4 Summary 58
EXPERIMENTS 60

5.1 Experimental Goals 60
5.2 Experiment Setup 61
5.3 List of Experiments 64
5.3.1 The Synchronization Test 64
5.3.1.1 Results and Discussion 65
5.3.2 The Scalability Test 69
5.3.2.1 Results and Discussion 70
5.3.3 The Overhead Test 72
5.3.3.1 Results and Discussion 74
5.4 Summery 76
SUMMARY AND CONTRIBUTION 79

FUTURE WORKS 82

7.1 Scaling Horizontally 82
7.2 User Modeling 82
7.3 Data Replication 83
LIST OF REFERENCES 84

LIST OF TABLES

Table page

Table 3-1 The Hypertext Transfer Protocol (HTTP) methods' properties 17

LIST OF FIGURES

Figure page

Figure 1-1 A smartphone uploading an image file and a tablet downloading a video file from a cloud-hosted web service that maintains a NoSQL database. 2

Figure 2-1 Mobile devices sharing multimedia files using a Web Service that implements the channel-based publish/subscribe communication scheme. 5

Figure 3-1 “Model-View-Controller State and Message Sending”. 9

Figure 3-2 “Passive View and Supervising Controller”. 10

Figure 3-3 Web Services actors, objects, and operations. 14

Figure 4-1 The overview of the architecture after applying the Model-View-Presenter concept. 30

Figure 4-2 The profile server, the profile server’s resources, the channel server and the media file’s server exchanging CRUD messages with a mobile device. 33

Figure 4-3 The profile server’s architecture with the travelling paths of the CRUD messages. 34

Figure 4-4 The webserver connecting the node proxy server to the mobile device with the travelling paths of the CRUD messages. 35

Figure 4-5 The resources’ server with the travelling paths of the CRUD messages and the channel’s notifications. 37

Figure 4-6 The global presenter’s components at the profile server’s resources with the travelling paths of the CRUD messages and the channel’s notifications. 38

Figure 4-7 The global model’s components at the profile server’s resources with the travelling paths of the CRUD messages and the channel’s notifications. 40

Figure 4-8 The channel server’s components with the travelling paths of the CRUD messages and the channel’s notifications. 45

Figure 4-9 The media file’s server with the travelling paths of the CRUD messages and the channel’s notifications. 49

Figure 4-10 The local presenter’s components. 52

Figure 4-11 The view’s components. 56

Figure 4-12 The local model’s components. 57

Figure 5-1 The experiment setup. 63

Figure 5-2 Step 1: the application's main menu registers the user automatically. 66

Figure 5-3 Step 2: the application captures a picture and stores it locally. 67

Figure 5-4 Step 3: after collecting the picture's metadata, the application uploads the picture and its metadata to the file and profile servers respectively. 67

Figure 5-5 Step 4: The application is uninstalled. 68

Figure 5-6 Step 5, 6 and 7: The application is synchronized automatically. 68

Figure 5-7 N of clients connecting to YAWS concurrently. 69

Figure 5-8 The throughput for the simulated WebSocket client groups. 71

Figure 5-9 The mean response time for the simulated WebSocket client groups. 71

Figure 5-10 The load generator sending WebSocket messages to YAWS. 73

Figure 5-11 The load generator sending CRUD messages to the registration unit. 73

Figure 5-12 An example of a CRUD message. 74

Figure 5-13 Phase 1 and 2 mean processing time. 75

LIST OF ABBREVIATIONS

Abbreviation

AMQP ..... Advanced Message Queuing Protocol

API ..... Application Programming Interface

CRUD ..... Create, Read, Update and Delete

GPS ..... Global Positioning System

HATEOAS ..... Hypertext As The Engine Of Application State

HQL ..... HyperTable Quarry Language

HTTP ..... Hypertext Transfer Protocol

MVP ..... Model-View-Presenter

NoSQL ..... Not only Structured Query Language

QoS ..... Quality of Service

REST ..... Representational State Transfer

ROA ..... Resource Oriented Architecture

RSI ..... Research Storage Interface

RSS ..... Research Storage System

SOA ..... Service Oriented Architecture

SOAP ..... Simple Object Access Protocol

SQL ..... Structured Query Language

UDDI ..... Universal Description, Discovery and Integration

URI ..... Uniform Resource Identifier

W3C ..... World Wide Web Consortium

WS-* ..... Web Services programming stack

WSDL ..... Web Service Description Language

YAWS..Yet Another Web Server

CHAPTER 1
INTRODUCTION

The popularity of smartphone and tablets is growing. Canalys© [22] predicted that the number of smartphone shipments will increase from 694.8 million in 2012 to 1342.5 million in 2016. Also, Canalys© predicted that the number of tablet shipments will increase from 114.6 million in 2012 to 353.5 in 2016. These mobile devices have access to web resources through multiple wireless communication interfaces such as 4G and Wi-Fi. However, these devices have limited storage and processing capacities. Also, they communicate through limited wireless bandwidth, which causes intermittent connectivity. The new advancement in the cloud computing services motivated researchers to host some of the data and information consumed by mobile applications in the cloud. Most of these cloud-based services are implemented as Web Services[20]; therefore these Web Services must scale to the increasing number of these mobile devices.

The popularity of these mobile devices motivated social media websites that host user generated multimedia content such as Facebook, YouTube and Instagram to enter the world of mobile applications. These applications allowed mobile users to upload and share their multimedia experiences from their mobile devices directly. The unstructured nature of multimedia files' data forced these websites to integrate NoSQL (Not only Structured Query Language) databases in their systems because these databases are more efficient at storing unstructured data than the traditional Relational DataBase Management Systems (RDBMS).

The major success of these mobile applications motivated the comedians in Saskatoon to search for a mobile application that publishes their videos directly to their fans. This thesis provides an example for designing similar applications.

Previous research in the field of Web Services has been conducted by Wickramarachchi et al.[37] that applied the publish/subscribe model to develop a scalable and persistent broker using

a NoSQL database called Andes. Andes implements the Web Service-Eventing and the Advanced Message Queuing Protocol (AMQP), which are based on the Simple Object Access Protocol (SOAP) messaging protocol. However, Pautasso et al.[26] stated that SOAP messages cause performance inefficiencies, whereas REpresentational State Transfer (REST) architectural style constraints offer better performance optimization possibilities for Web Services. Furthermore, Aijaz et al.[1] recommended the REST approach for mobile Web Services[1] because of their lightweight messages [25].

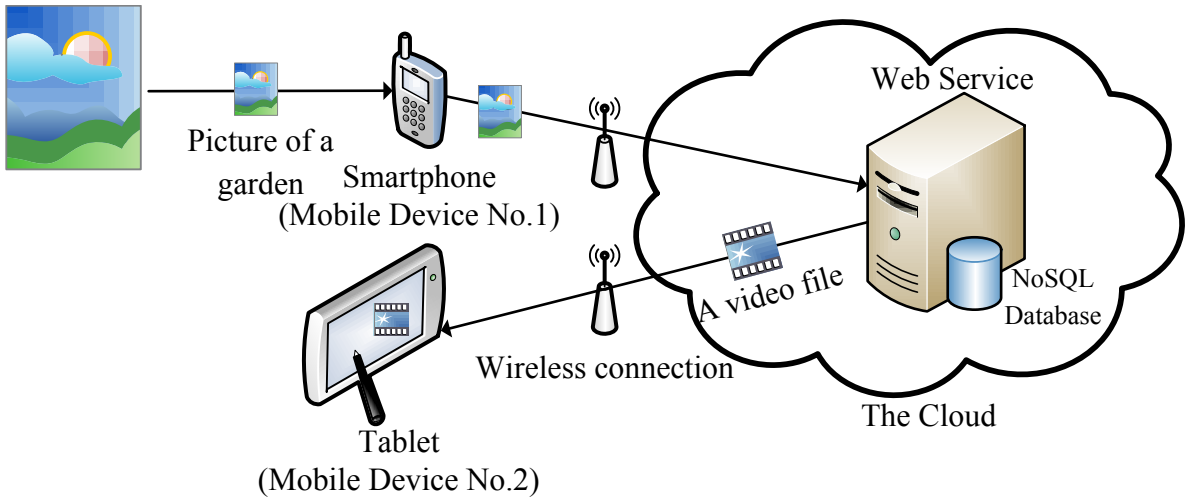

Figure 1-1 A smartphone uploading an image file and a tablet downloading a video file from a cloud-hosted web service that maintains a NoSQL database.

To understand the basic idea behind multimedia file sharing using a mobile Web Service, consider the two mobile devices shown in Figure 1-1. Mobile device number one captures an image using its embedded camera. Mobile device number one uploads a copy of the image to a cloud-hosted web service. Mobile device number two downloads a video from that web service. That video was either created previously by mobile device number two or shared by another mobile device.

Based on the reasons mentioned above, This thesis proposes a novel approach for implementing a REST-based mobile Web Service for multimedia file sharing that utilizes a channel-based publish/subscribe communication scheme to synchronize smartphone or tablet-hosted NoSQL databases with a cloud-hosted NoSQL database. This Web Service is designed according to the Model-View-Presenter model to reduce the complexity of the synchronization operations.

The rest of the proposal is organized as follows: Chapter 2 discusses the problem definition. Chapter 3 reviews research related to the Model-View-Controller and its successor the Model-View-Presenter architectural models, the different types of the publish/subscribe communication scheme, different types of NoSQL database data models, NoSQL database examples and some popular NoSQL querying mechanism. Chapter 4 describes the architecture. Chapter 5 presents the experiments.

CHAPTER 2
PROBLEM DEFINITION

This research focuses on mobile devices that share data using a mobile Web Service. From the scenario presented in Chapter 1 an issue emerges: “How can smartphone and tablet-generated multimedia files be shared and hosted efficiently both locally and in the cloud through wireless communication interfaces?” These devices’ wireless communication interfaces have limited bandwidth, which causes intermittent connectivity[20]. Also, the metadata (information about data) of the user generated multimedia files varies among different mobile devices. The unstructured nature of these files and their metadata raises another question. “How can a mobile Web Service accommodate the various types of multimedia files and their metadata?” To address this question the principles of NoSQL databases were adopted.

NoSQL databases are data stores that have fixable schemes. As we will see in Chapter 3, some of the cloud-hosted NoSQL databases’ features indicate that they are suitable for storing and retrieving multimedia files’ metadata globally, even if the metadata varies among different portable devices. Also, local NoSQL databases are designed for local data management and fast retrieval of locally stored data. However, local NoSQL databases are limited to the memory capacity of the local hardware and lack the support for online data distribution. To address these issues, this thesis presents a novel approach to achieve synchronization between local and cloud-hosted NoSQL databases.

In this approach, the mobile devices communicate through a channel-hosting Web Service. That Web Service will be based on the publish/subscribe communication scheme. Figure 2-2 shows a mobile device’s user creating and registering to a sharing channel illustrated as C (1). Other users will be able to register to channel C (2). When a registered and certified user uploads multimedia files to channel C (3), the other registered users will be notified (4).

However, there are two problems that need to be addressed to implement such systems:

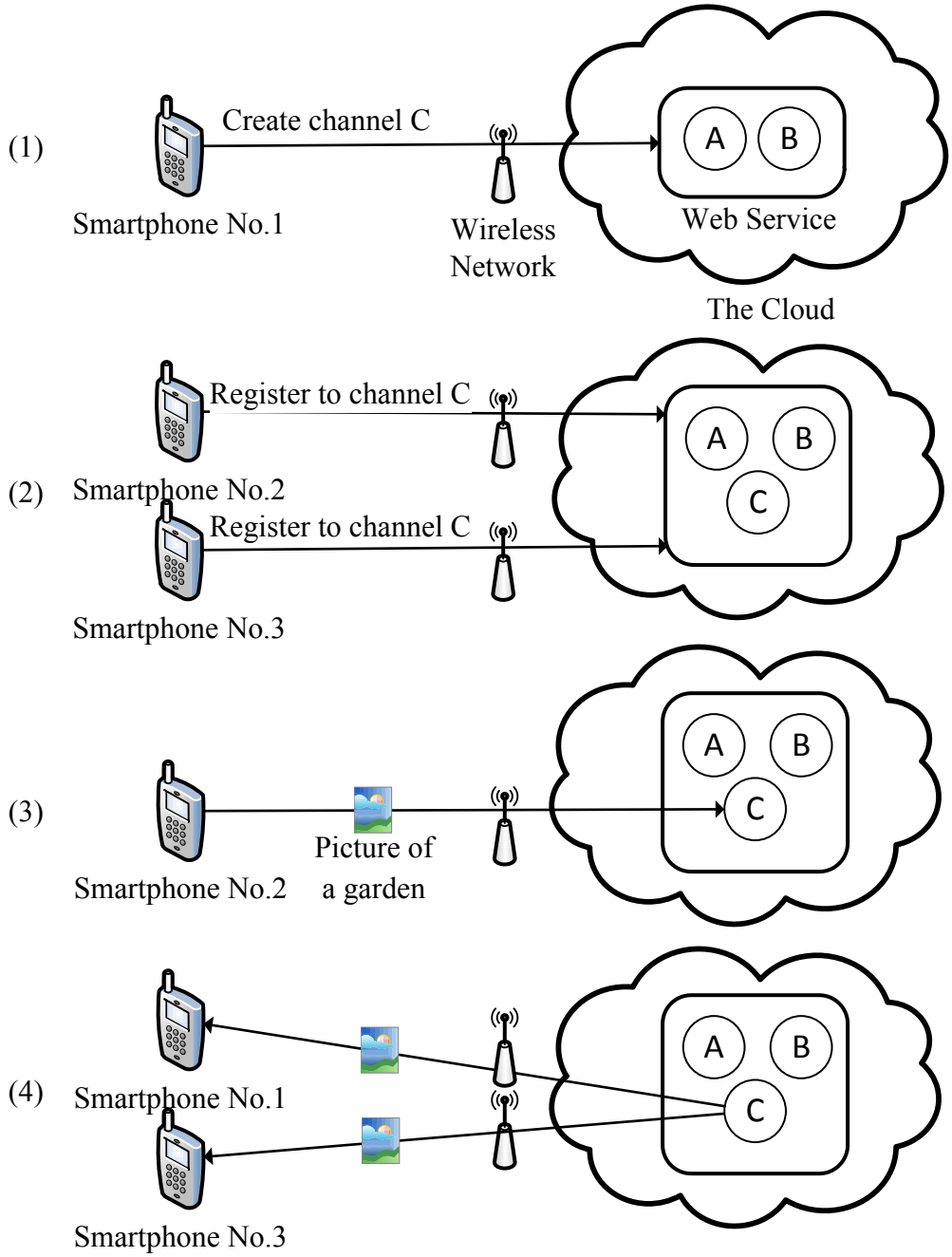

Figure 2-1 Mobile devices sharing multimedia files using a Web Service that implements the channel-based publish/subscribe communication scheme.

- How can the Web Service synchronize the objects of the local database with the data objects of the external database and *vice versa*?

- How can a channel-based publish/subscribe Web Service that maintains a NoSQL database share multimedia files and their metadata among multiple mobile devices?

To answer these questions, two challenges need to be considered:

1. **Data consistency:** There are two representations of the same data object in two different parts of the system (the local and the global databases). Any update to one object must be applied to all copies of that object in the system; therefore the system should distribute the data updates automatically and as fast as possible.
2. **Scalability and latency:** Large numbers of users will upload and download multimedia files. These multimedia files are expected to contain high volumes of data. At the same time, other users will be trying to update the metadata of their files; therefore the system should be able to expand according to the incoming requests.

The goal of this research is to develop a multimedia file sharing mobile Web Service that synchronizes local NoSQL databases with a cloud-hosted NoSQL database. This Web Service will have an Application Programming Interface (API) that enables web applications to benefit from the advantages of the local and the cloud-hosted NoSQL databases. This API uses create, read, update and delete (CRUD) functions of persistent storage to hide the complexity of the synchronization operations. This Web Service utilizes the publish/subscribe communication scheme to disseminate the synchronization messages. Furthermore, this Web Service stores the multimedia files and their metadata in the cloud to ensure their availability to the connected mobile devices. Also, this Web Service achieves scalability by producing parallel processes to serve multiple mobile devices concurrently. To achieve this goal, the following key questions emerge:

- What are the operations needed to synchronize between the local and the cloud-hosted NoSQL databases?
- How can the Web Service distribute the synchronization messages among the different parts of the system?

CHAPTER 3
LITERATURE REVIEW

At present, mobile applications' and Web Services' developers follow high-level architectural patterns to design their systems. These architectural patterns reduce the complexity of the development phase by dividing the system conceptually into smaller units and assigning a role for each unit. Section 3.1 reviews the literature related to designing applications using the Model-View-Controller and its successor the Model-View-Presenter architectural patterns. To understand the interactions within the Web Service itself and with the mobile devices, section 3.2 reviews the concept of the publish/subscribe communication scheme, the different approaches that emerges from that concept and the advantages and disadvantages of each approach. This Web Service represents the server and the mobile devices represent the clients in the proposed system; therefore sections 3.3 and 3.4 review two candidate client-server communication architectures for implementing Web Services to study the differences between those architectures and to choose the most suitable approach for implementing mobile Web Services. To understand the concept of NoSQL databases, section 3.5 reviews the different types of NoSQL databases and their advantages and disadvantages compared to the traditional SQL databases in terms of data structure and querying mechanisms.

3.1 The Model-View-Controller

The Model-View-Controller is an application design pattern. It was introduced by Trygve Reenskaug[29] as a group of rules that helps application developers to design reusable graphical user interfaces. He argued that these graphical interfaces would be independent from the presented data and allow the users to control a large and complex data set. The Model-View-Controller design pattern divides the implemented application into three parts:

- The model represents the stored data in the application (such as a string, a data object … etc.) and the functions needed to process the stored data.
- The view is the part of the interface that displays the data. The displayed data is provided by the model.
- The controller is the part of the interface that captures the user's interactions with the displayed data.

One model is displayed by one or more views while each view is paired with one controller. The interaction cycle can be explained as the following:

One of the controllers receives an action from an external source. That action is passed to the model as a message. That model notifies all the related views and controllers. The related views query the model for the changes and update its graphical interface. Figure 3-1 shows the exchanging of messages among the Model-View-Controller's components.

Krasner et al.[17] implemented the graphical interface of the Smalltalk-80 programming environment according to the Model-View-Controller programming paradigm. They argued that this paradigm helps in the conceptual development of the applications and code reusability.

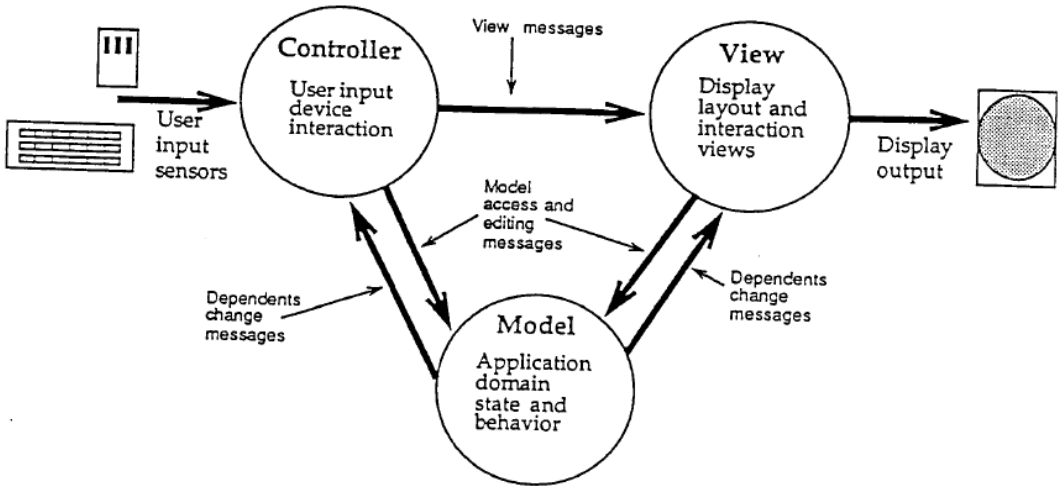

Figure 3-1 "Model-View-Controller State and Message Sending"[17].

Potel's report[28] presented the Model-View-Presenter (MVP), a generalized form of the Model-View-Controller. While the Model-View-Controller is applied when implementing graphical user interfaces, the Model-View-Presenter programming model can be applied when implementing a broad range of applications (such as client/server and multi-tier application architectures). The Model-View-Presenter describes the view as an interface that could be non-visual (such as, a bar code reader).The Model-View-Presenter has the following components

- The model is either the application's data objects or a source of data objects (such as a database);
- The view is considered to be passive when the view communicates with the model through the presenter or supervised by the controller when the view communicates with the model directly using data-binding operations[23]. Figure 3-2 displays the types of view interactions in the Model-View-Presenter Programming model and;

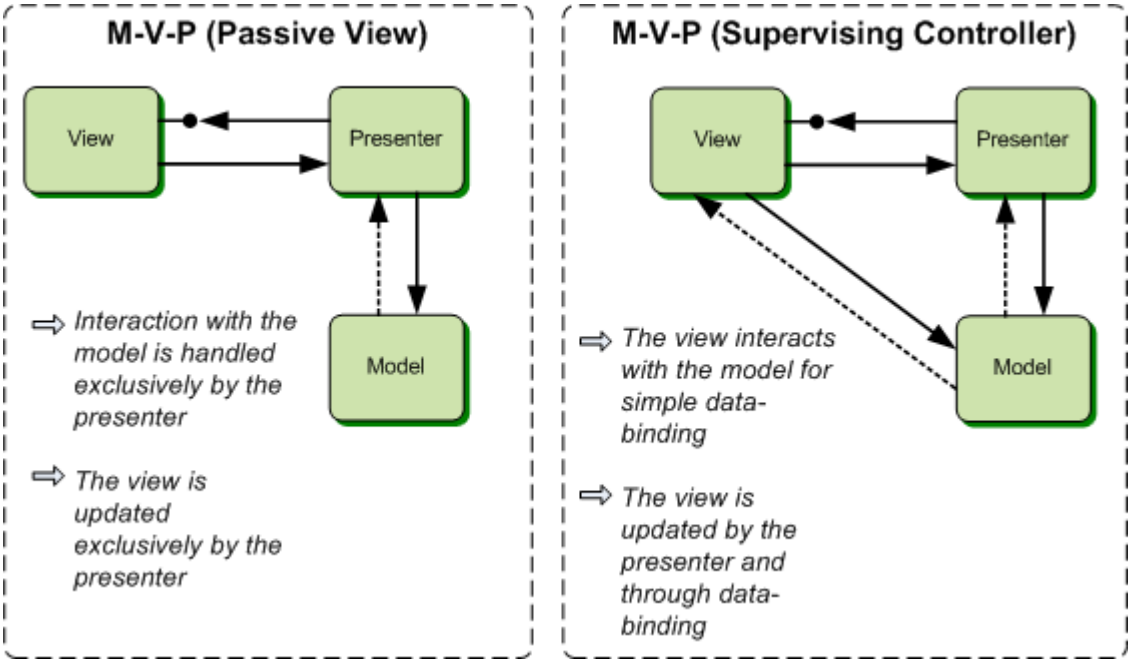

Figure 3-2 "Passive View and Supervising Controller"[23].

- The presenter connects the view with the model by mapping the events received from the view to the commands to be executed by the model. Also, the presenter is responsible for mapping and selecting the data to be displayed by the view.

The Model-View-Presenter integrates small program units to produce full applications. These program units can be developed and tested separately by multiple developers.

3.2 Publish/Subscribe Systems

As described by Eugster et al.[8], the publish/subscribe communication paradigm is an interaction scheme implemented in large-scale applications to loosely couple their distributed interactions. The basic components of the publish/subscribe communication paradigm are

- Event service: is responsible for receiving, storing, managing and delivering information. This information represented by the term event;
- Publisher: produces and publishes (sends) events to the event services; and
- Subscriber: subscribes (registers) at the event service to receive the publisher's events. The act of sending the event to the subscriber is denoted by the term notification. The subscription information is maintained by the event service.

The event service's behavior is similar to a proxy because when an event service receives an event from a publisher, the event service notifies all its subscribers.

The decoupling between publishers and subscribers can be expressed in the following situations:

- Space decoupling: The publisher and the subscriber can be unknown to each other;
- Time decoupling: The publisher can interact with the event service even if the subscriber is offline and *vice versa*; and

- Synchronization decoupling: there is no blocking during event production or notification.

This decoupling among the publishers and the subscribers increases scalability because distributed environments, such as the mobile environments, are asynchronous by nature[8][37].

3.2.1 Publish/Subscribe Variations

According to Eugster et al.[8], publish/subscribe systems can be divided into three groups based on the way of specifying the events of interest:

3.2.1.1 Topic-Based Publish/Subscribe Scheme

Subscribers express their interest in a topic by joining the topic's group of subscribers. Publishing an event is achieved by broadcasting the event to the members of the topic's group. Every topic is considered as an event service. These topics have a distinct communication channel used for publishing events, a unique name and an interface that offer the publish and the subscribe operations.

The topic-based publish/subscribe scheme is easy to understand. This scheme supports cross platform environments because it uses strings to uniquely identify the topics. This scheme's topics can be divided into subtopics and organized hierarchically. This organization introduces URL-like addressing and enable the users to publish and subscribe to multiple topics that have similar set of keywords[34]. However, the topic-based publish/subscribe scheme limit the system's expressiveness because strings are static and primitive. Also, this scheme forces the subscribers to remove irrelevant events; therefore it reduces the bandwidth consumption's efficiency. Furthermore, splitting topics into several subtopics increases the possibility of having redundant events.

3.2.1.2 Content-Based Publish/Subscribe Scheme

Subscribers select an event service according to the properties or the contents of its events[21]. The content of these events can be distinguished by the attributes of the event's data structure and the event's meta-data. Subscribers specify event property filters using a subscription language. There are different types of filters:

- String based filters: such as SQL statements;
- Template based filters: the subscriber provides a template object. This template contains the properties of the event; and
- Code based filters: the subscriber provides an executable code used to filter events at runtime or access the event object using reflection.

The content-based publish/subscribe scheme utilizes a subscription language that helps in implementing highly expressive event filters. Also, this scheme filters its events using the event service. However, this scheme forces the system to implement high-end protocols that have higher runtime overhead. Furthermore, optimizing the filters is an extremely hard task[8].

3.2.1.3 Type-Based Publish/Subscribe Scheme

Subscribers select an event service according to the data types of the event's content. The type-based publish/subscribe scheme adds an extra layer of control to the filtering mechanism used in the content-based publish/subscribe scheme. The type-based publish/subscribe scheme increases the integration between the subscription language and the event service. Also, this scheme ensures the event content's data type is consistent throughout the system at compile-time[7].

3.3 Service Oriented Architecture (SOA)

Gottschalk et al.[10] described a Web Service, in a distributed system that implements the Service Oriented Architecture, as a program or a collection of programs accessible through a network connection by exchanging standardized XML format messages. Web Services support machine-to-machine interaction because they are described by a standard and formal XML representation. This representation is defined using the Web Service Description Language (WSDL).

SOA utilizes the Web Services programming stack (WS-*), which provides standardized protocols and application programming interfaces to Web Services. WS-* applies design restrictions to how Web Services interact with the Web Service consumer (Figure 3-3).

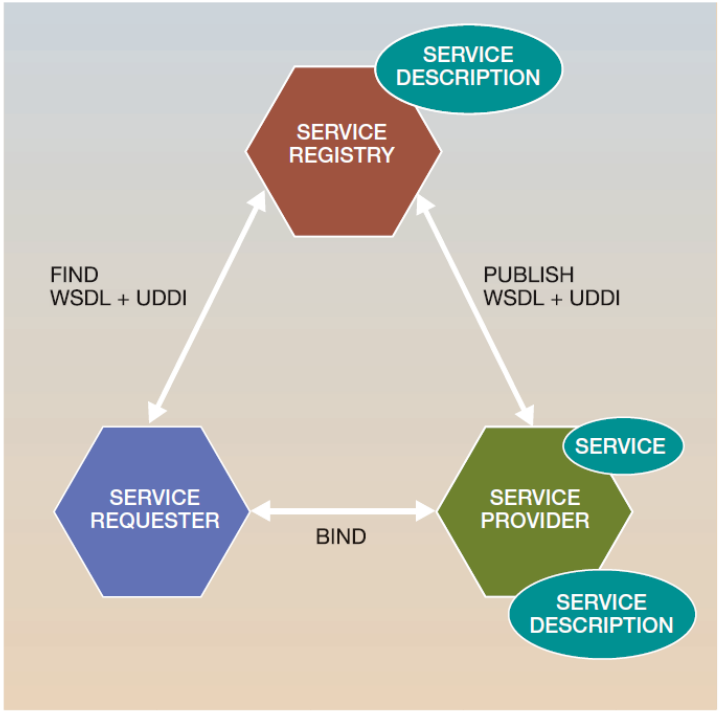

Figure 3-3 Web Services actors, objects, and operations [10].

To consume a Web Service, the service provider publishes the service description at the service registry. The service description must comply with the Universal Description, Discovery

and Integration (UDDI) specifications. The service provider maintains and monitors the Web Service. The service consumer queries the service registry and retrieves the service description. The service consumer translates the description and learns how to contact and use the Web Service. The service consumer interacts with the Web Service through a network connection.

Implementing the WS-* standards allows protocol[25]

- Transparency: Transferring a message through different middleware systems without changing its format; and
- Independence: Maintaining quality of service (QoS) aspects even if the transfer protocol changes along the way of the message.

Because of the WSDL, the Web Service is independent from the message transition details, the implementation environment and the execution environment.

The development environments that work according to these standards are programmer friendly because they minimize the time needed to learn the required specifications that connects different platforms to one service [26]. The SOAP is an example of a communication protocol that implements the WS-* standards.

3.4 Resource Oriented Architecture (ROA)

A resource in a web environment is a fully and clearly expressed entity (e.g. a web page, a file, an application … etc.) accessible through the network using a unique identifier[25]. Each resource maintains a resource state. Resources expose and perform a task or multiple tasks on their resource states. The resource consumer interacts with the resource using a unique Uniform Resource Identifier (URI). That interaction is described as stateful because the resource maintains its own state.

According to SOA[10], the client sends a message to the service. That message contains

- The current client state;
- The description of the client state;
- The changes that will be applied to that state; and
- The description of these changes.

This concludes that services maintain stateless interactions. Stateless interactions help in minimizing the service's memory consumption and the number of connections needed to apply multiple updates to the shared state [9]. Instead, ROA uses the URI as a light weight representation of the client state. The message received by the resource is stateless but it contains enough information to change or retrieve the state of the resource. This stateful interaction reduces the network's bandwidth consumption and allows the users to have direct interaction with the resource provider[25].

The client communicates with the resources using predefined classification system. One of the currently used classification systems for the ROA is the Hypertext Transfer Protocol's (HTTP) methods such as GET, DELETE, PUT and POST.

The HTTP protocol classifies the exchanged messages between the client and the server (the resource provider), according to the source of the message, into two types[11]:

1. HTTP request: these are messages sent from the client to the server. Each HTTP request has
    - A Request-Line: consists of a HTTP method and a URI.
    - A group of HTTP request headers: they describe the properties of the request.
    - A HTTP request body: contains data to be used by the server to execute the HTTP request.

2. HTTP response: these are messages sent from the server to the client after executing the request. Each HTTP response has
    - A Status-Line: contains a status code, which is a numeric code that defines the type of the response
    - A group of HTTP response headers: it describes the properties of the response.
    - A response body: it contains data or a description of the data emerged as a result of executing the client's request.

Roy Fielding presented in his dissertation[9] a new concept based on the ROA called the Representational State Transfer (REST) architectural style. Roy Fielding applied REST to the HTTP protocol to identify problems the HTTP/1.0 implementations had before HTTP/1.1.

According to Fowler's webpage[30], there are multiple steps to implement a HTTP based RESTful system:

Table 3-1 The Hypertext Transfer Protocol (HTTP) methods' properties

The method	The effect on the resource's state	idempotent	Cacheable
GET	No effect	Yes	Yes
DELETE	Stop using the resource	Yes	No
PUT	The resource will receive the data	Yes	Yes
POST	A new resource is created	No	No

1. Use the HTTP as a transport system for remote interactions.
2. Define the services as individual resources identified by unique URIs.
3. Implement the HTTP request methods POST, GET, PUT and, DELETE as Create, Read, Update and Delete (CRUD) operations respectively to interact with the resources. Table 3-1 shows the different side effects of these methods on the resource and the methods' behavior and in the network.

4. Use HATEOAS (Hypertext As The Engine Of Application State) approach by including the URIs needed to update the state of the client in the HTTP response.

Researchers and developers consider systems that satisfy the first three constraints to have a lower form of the REST architecture.

One of the advantages of the REST architecture is the ability to cache the server's response at the client side on demand[9]. Table 3-1 shows some of the cacheable HTTP methods. Also, the HTTP server has the option to decide if the response to a specific request is cacheable or non-cacheable. The flexibility of the cache in RESTful systems reduces the number of the unnecessary requests received by the server.

Implementing RESTful Web Services reduces the communication's payloads of the resource constraint mobile applications[1]. Other applications integrate with both SOAP and RESTful HTTP Web Services[19].

3.5 NoSQL Databases

NoSQL databases were implemented to have fast data access and store large volumes of heterogeneous datasets[15]. Cloud hosted NoSQL databases are capable of reaching high scalability levels while responding to queries regarding large blocks of sparse data [32]. These features indicate that NoSQL databases are suitable for storing and retrieving multimedia files' metadata globally.

The purpose behind the emergence of NoSQL databases was to customize the behavior of data stores to serve specific application requirements instead of relying on the traditional generic RDBMS. Most of the relational databases support the traditional Structured Query Language (SQL) for data retrieval. However, NoSQL databases implement their own query languages such

as Cassandra's CQL or support both native and the standard query language such as OrientDB[24].

3.5.1 Data Models

While relational databases used the relational data model, the NoSQL databases depend on a variety of different data models that are efficient when retrieving massive sparse data sets with loosely defined structures[15]. These data models fulfill the requirement for storing and retrieving multimedia files' metadata. Data models like key-value, document, column-family and graph allowed horizontal scalability and scheme flexibility to be the main characteristics of NoSQL databases while running on clusters[13].

3.5.1.1 The Relational Data Model

The relational data model[4] consists of a set of tables. A database table is a set of stored rows (tuples). A row is a collection of related data values. Each data value is defined by a column. A column defines the data type (text, number, etc…) of a data value and has a fixed data size. Columns can be shared among different tables to represent a relationship among the tables' rows. The organization of columns and tables in a relational database is decided by the database administrator using the process of Normalization. The Normalization process minimizes data redundancy and dependency by defining a structure to the stored data.

Relational databases expect the data to have a well-defined structure; to be dense and largely uniform; and to maintain predefined data properties. Also, the relationship among the data properties is expected to be well established and systematically referenced. These rules enable relational databases to have transactional integrity and flexible indexing and querying, but lack the support for the largely sparse data of the Internet[15].

3.5.1.2 The Key-Value Data Model

The key-value data model is used by key-value NoSQL databases. The key-value data model consists of unrelated tables. The key-value table stores rows in the form of key-value tuples. In a key-value table, the data stored in the key data field are used to retrieve the data stored in the value data field[13]. There are two types of key data fields:

- Simple key: consists of one data field.
- Composite key: consists of multiple data fields.

The value data field consists of blobs of data that have no defined data structure. The value data field has no size limitation in most of the key-value NoSQL databases. The advantage of using the key-value data model is data opacity. Data opacity enhances the decoupling between the database and the application's data. Also, key-value databases are suitable for intensive read operations[13].

3.5.1.3 The Document Data Model

The document data model is used by document NoSQL databases. The document and the key-value data models organize data in similar ways. While the key-value data model provides no structure for the stored values, the document data model stores the value data field as a document. Each document consists of multiple attribute-value fields[15]. The document data model presents two types of fields:

- Data fields that have data type definition and size limitation. They are used for indexing.
- Data fields that have neither data type definition nor size limitation. They are used for storing the data as one atomic value.

Documents are independent and can have different attribute-value field sets. This data model has less data opacity but provides more access flexibility than the key-value model. Also, Document NoSQL databases have a fixable scheme and overcome the problem of impedance mismatch in the relational databases[14][15].

3.5.1.4 The Column-Family Data Model

The column-family data model is used by wide column databases[18]. The column-family data model stores data as rows. Each row contains a key field to retrieve the data stored in the row's value field. The row's data are stored as multiple columns. Each column has a key field used to retrieve the data stored by multiple rows that share the same column. Related columns can be grouped into a super column. Super columns used to store complex data types. Each column and super column must belong to a singular column family. A column family is a set of columns and super columns that can be accessed as a group.

This data model introduces the ability to retrieve partial data objects from highly sparse data collections. Wide column databases are utilized by high performance applications in the business intelligence domain[15].

3.5.2 The Graph Database

The Graph database is a special case of the NoSQL databases. It focuses on building complex relations among nodes of stored data. The Graph database is designed to solve the problem of storing highly interconnected data structures, such as the data generated from the social networks[15]. The Graph database retrieves the data without the need for explicit join operations (such as joining records from related tables in relational database). Each node stored in the graph maintains direct reference to the nearest related nodes. However, the graph database is inefficient

in terms of data partitioning or distributing the graph over a network because it increases the number of the steps needed to retrieve all the related nodes[31].

3.5.3 Examples of NoSQL Databases

Some examples of NoSQL databases are briefly reviewed below[32]:

1. Cassandra is a structured key-value data store that features scalability, eventual consistency and data distribution. Cassandra's data model is based on the column family data model. Each column is referenced by a key and contains
   a. A name, a value and a timestamp represented as a tuple or,
   b. Sub columns then it is named super column.

   A keyspace is a one-per-application container for family groups.
2. HyperTable is an associative-array data store that achieves scalability and distribution for structured and unstructured data on a large cluster of commodity servers. The user can create, modify and quarry the multi-dimensional tables of information from the system using HyperTable's low-level API and the HyperTable Quarry Language (HQL).
3. CouchDB is a document-oriented database accessible through RESTful HTTP API. Each CouchDB document consists of named fields that have string, number, date, ordered list or associative map data types and a unique ID. CouchDB has an optimistic update model where a new version of a document refuses updates to any of the previous versions of the same document and all or nothing document updates that don't allow partially updated or saved documents. JavaScript is used as a quarrying and indexing language in CouchDB because of its MapReduce programming model. The MapReduce programming model supports cheap-cost hardware. Also, it is fault-tolerant. Furthermore, it is highly parallel.

4. IndexedDB is an indexed key-value NoSQL database. IndexedDB provides local data storage for web applications. IndexedDB's API was standardized by the World Wide Web Consortium (W3C) in 2010. According to the W3C, this API "performs advanced key-value data management that is at the heart of most sophisticated query processors"[12]. Furthermore, IndexedDB was categorized as a NoSQL database because it uses a non-relational data model.

However, the NoSQL systems have limited support for ad-hoc querying and analysis operations and transactions. Also, these systems are still in pre-production stages. Furthermore, there are few NoSQL database experts[32].

3.5.4 Querying Mechanisms

Traditional relational databases share the standardized SQL. However, NoSQL languages revolutionized their data access mechanisms. The following data access techniques have been reviewed.

3.5.4.1 SQL Statements

A SQL statement[33] is a collection of query blocks. Each query block is represented by a:

- SELECT List: a list of fields to be accessed;
- FROM List: a list of tables to be accessed; and
- WHERE tree: a group of simple predicates combined using Boolean operations.

Any of the mention variables could be a query block; therefore these statements can grow in complexity and processing time.

SQL statements are interpreted in four stages:

1. Parsing: the parser search for syntax errors,

2. Optimization: the optimizer verifies the given lists and retrieves statistics about the relations among the list of tables. The optimizer checks for semantic errors and type compatibility using the obtained data type and length of each field. The optimizer builds an execution plan by selecting the minimal cost solution from a list of alternative paths to join the SELECT list,
3. Code generation: the code generator translates the execution plan into an executable machine language code.
4. Execution: the Research Storage System (RSS) communicate with the code generator using the Research Storage Interface (RSI) to execute the generated machine language code.

SQL statements hide the physical storage operations and the indexing of the stored data from the users. Although the optimization stage tries to reduce the number of database retrievals, the need to run these queries on a single machine introduces a physical hardware limitation in scalability.

3.5.4.2 MapReduce

MapReduce is one of the popular data retrieval techniques implemented by key-value, document and column family NoSQL databases. MapReduce was introduced by Google in 2004[5] to process large amounts of raw data. The raw data is stored in the form of key-value pairs. The MapReduce query is expressed using two functions:

- Map: the input of this function is the key-value pairs stored in a file. The map function filters these pairs using logical statements. The output of this function is a set of intermediate key-value pairs. The values of the intermediate key-value pairs are grouped together if they have the same key.

- Reduce: the input of this function is the filtered intermediate key-values groups. The reduce function iterates through all the intermediate keys and combines their values to produce a set of values. The final product of the reduce function is a generic view (e.g. average, count, summation …etc.) of the intermediate key-value pairs' values.

MapReduce is able to run on clusters because the input data is partitioned into key-value groups. These groups can be processed on different nodes in parallel using the map function. The resulting intermediate key-value pairs are grouped and buffered in the memory of the processing nodes. The nodes use the reduce function to process the buffered groups in parallel and return an intermediate result to a central node. The central node combines the nodes' intermediate results and sends the final result back to the user's application. When most of the nodes finish their tasks, the central node starts backup tasks similar to the remaining tasks on idle nodes to reduce the completion time of large MapReduce operations.

MapReduce is fault tolerant because the processing nodes have independent input data and output results during the execution of the map function. Any node failure can be resolved by re-executing of the map function in the same node because the output of the map function is stored locally. However, the failed reduce function does not need to be re-executed because the central node commands an idle node to read the result of the map function. The central node notifies all the reduce functions related to this query before the execution of the reduce function by the new node.

However, using multiple nodes to process each query causes high resource and bandwidth overhead[36]; therefore MapReduce is mostly suitable for batch and preprocessing operations.

3.5.4.3 Secondary indexes

Secondary indexes are references to data locations stored as additional information in NoSQL databases[36]. These references are stored in the form of key-value pairs. The value field stores non-primary key attributes of the stored data objects. The non-primary key attributes are used to identify the location of the queried data objects without the need to broadcast the query to all the nodes in the system. According to von der Weth et al.[36], there are two variations of secondary indexes:

- Inverted indexes: the key field of the secondary indexes is derived from the value of the attributes used for indexing the stored data object. The value field contains the primary key used to retrieve the stored data object. Inverted indexes support exact-match queries if the database does not interpret the stored key otherwise they support range queries.
- Tree-based indexes: such as Distributed B-Trees and Prefix Hash Trees where the key-value pairs are stored in the form of a logical tree. The edges of the tree points to the location of the data objects.

Secondary indexes result in maintenance cost overhead during the update operations and additional storage overhead.

3.6 Summary

The explored research literature can be summarized as the following:

- Independent units of the application can be integrated using the MVP design pattern[28]. Separating the database (the model) from the interface of the mobile device (the view) allows the server to be independent from the large varieties of databases and interfaces implemented by different mobile devices. Also, it allows the mobile devices to be independent from the

server's database. Furthermore, this separation reduces the complexity of the synchronization operations among the proposed system's local and the global NoSQL databases.

- The expected metadata from the multimedia files (video, audio … etc.) generated by the mobile devices have different structure (resolution, frequency … etc.). Also, the server must adapt to any futuristic metadata formats; therefore NoSQL databases are good candidates to store the data at the server side because of its support for sparse data[15] meaning that the data does not have to match a specific structure before storing it.
- Key-value based NoSQL databases have high data opacity and simple retrieving mechanism compared to document based and column-family based NoSQL databases respectively[15]. While Graph NoSQL databases limit the scalability of the system to a single machine[31], Key-value based NoSQL databases can scale horizontally; therefore key-value based NoSQL databases allow the proposed system to accommodate the large numbers of concurrently connected mobile clients.
- The expected data queries for the stored multimedia file's metadata will not be statistical queries and simple key data fields do not support range queries. Also, MapReduce causes high resource and bandwidth overhead[36]; therefore the inverted secondary indexes approach can be used to satisfy these queries.
- Publish/subscribe communication scheme solves the problem of intermitted connectivity by decoupling the publishers and the subscribers in the time dimension[8]. This time decoupling allows the registered users to access the multimedia files and their metadata even if the creator of this data is disconnected. However, the type-based publish/subscribe approach reduces data opacity. Also, the multimedia file's type and its metadata's structure differ from one mobile device to another; therefore the proposed system implements the content-based

publish/subscribe approach to support various types of mobile devices. Furthermore, the proposed system implements the topic-based publish/subscribe approach to allow the users to organize and share their published content according to the content's topic.

- RESTful messages are cacheable[9], which can be used to synchronize the proposed data storages because the data storage at the client side will behave as a cache of the data stored by the data storage at the server side. Also, RESTful systems are recommended for mobile Web Services[1] because of their lightweight messages [25].

The following are questions left unanswered by the literature review:

- How to take advantage of both the content-based and the topic-based publish/subscribe systems?
- How to utilize RESTful messages as a synchronization and a notification mechanism?
- How to distribute the presenter between the server and the client?

CHAPTER 4
ARCHITECTURE

This chapter expands on Chapter 1 and Chapter 2's scenarios by applying the concluded approaches of the previous chapter. Section 4.1 applies the Model-View-Presenter concept to the scenario presented in Chapter 1 (Figure 1-1) by identifying the client's and the Web Service's responsibilities in the proposed system. Sections 4.2 and 4.3 apply the REST and the publish/subscribe concepts to the scenario presented in Chapter 2 (Figure 2-1). Section 4.2 applies the REST concept to the server side model presented in Section 4.1 by dividing the Web Service into three resources. Also, Section 4.2 applies the publish/subscribe concept to two of these resources. Section 4.3 explains the mobile client's behavior in three subsections. Section 4.4 includes this chapter's summary and conclusion.

4.1 The Concept of The Model-View-Presenter

The Model-View-Presenter approach provides a general overview of the architecture (Figure 4-1). This overview distinguishes five main components: two models, two presenters and the view.

The models represent the storage components used in the system. Users are expected to share locally generated and stored data through a Web Service. Potel[28] suggests a viewless Model-View-Presenter to design client-server based systems; therefore this thesis proposes two model components to store the generated data:

- The global model: it represents globally accessible data storage. It stores the mobile devices' uploaded data in a server. The global model is based on the viewless Model-View-Presenter approach; and

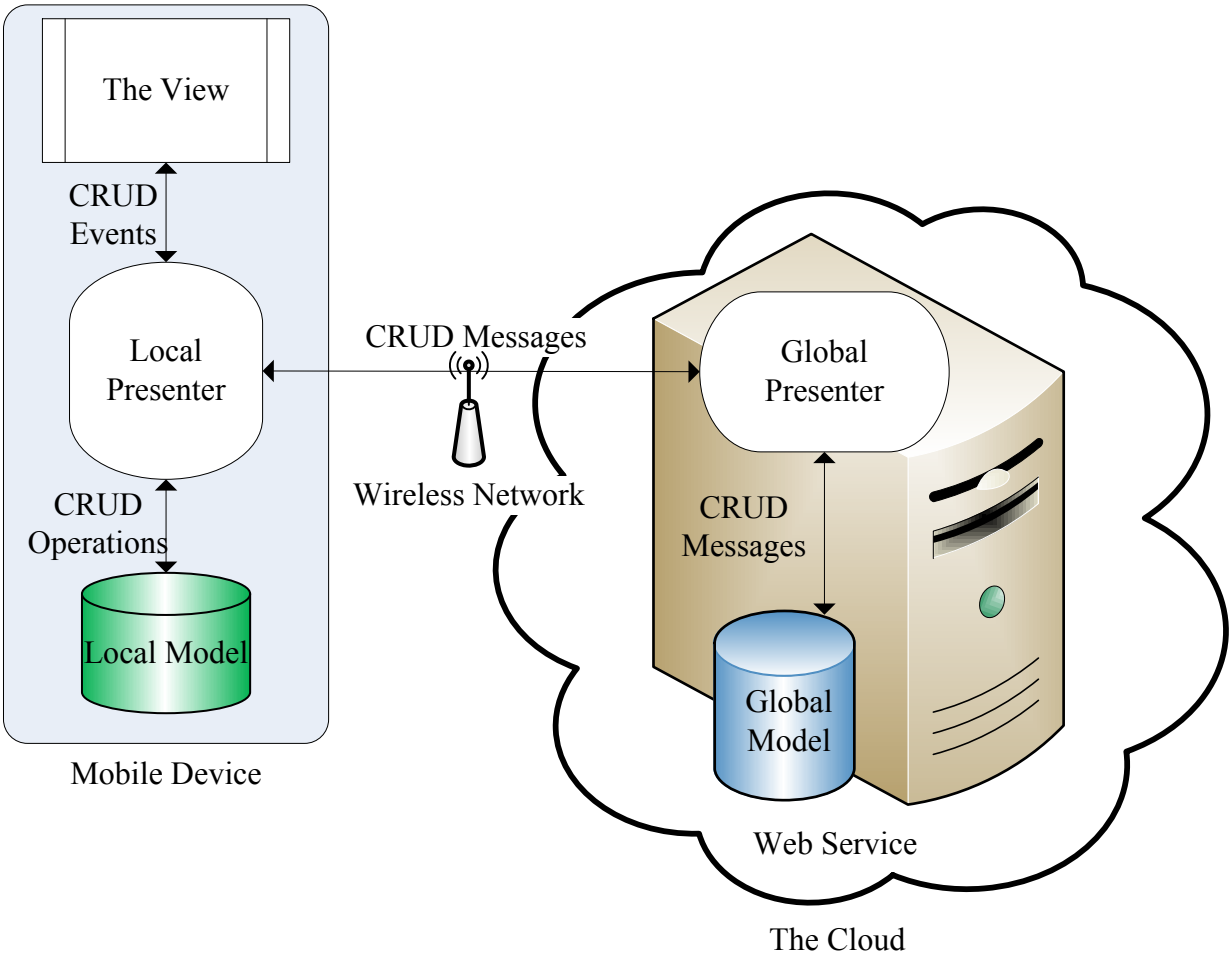

Figure 4-1 The overview of the architecture after applying the Model-View-Presenter concept.

- The local model: it represents locally accessible data storage. It stores the locally generated data and a subset of the globally shared data that are related to that specific mobile device.

The view represents the graphical user interface of the mobile device. The view's responsibility is to capture the user's input and display the presenter's data. The Passive View approach[23] is selected because of the difference in the access method between the local model and the global model. Also, the intermittent connectivity of the wireless network causes the global model to be inaccessible by the view during network cuts.

The presenter represents the narrator between the model and the view. The presenter transforms the exchanged messages to the destination's compatible form. The presenter ensures the consistency of the models' shared data and the availability of the data to the view. The presenter generates the messages that enclose the exchanged data and transfers them to the models. In this project, the term Create Read Update Delete (CRUD) message is used to represent the messages exchanged between the presenter and the global model. The term CRUD operation is used to represent the messages exchanged between the presenter and the local model. The term CRUD Events is used to represent the messages exchanged between the view and the presenter. However, there are two identified models in the proposed system. The presenter communicates with these models using different types of messages; therefore the overview of the architecture assigns the rule of the presenter to two system components

- The global presenter: it represents the sharing mechanism of the system. The global presenter implements the publish/subscribe communication scheme; and
- The local presenter: it transforms the system's messages to their destination forms.

Let's revisit Chapter 1's example, shown in Figure 1-1, to identify these components. Mobile device number one captures an image using its embedded camera. Mobile device number one's graphical user interface generates a Create event. This interface represents the view. This interface passes that event to the local presenter. The local presenter stores the image in the local file system and collects the image's metadata. The local presenter generates a Create operation and passes it to the local NoSQL database. Both the local file system and the local NoSQL database represent the local model. The local presenter generates two Create messages

- The first message contains the image. The local presenter passes it to the media file's server; and

- The second message contains the metadata. The local presenter passes this message to the profile server.

The profile and the media file's servers represent the global presenter. The profile server passes the first Create message to the mobile device number one's resource hosted at the profile server. The media file's server extracts the image from the second Create message and store that image in the server's file system. The profile server's resources and the media file server's file system represent the global model.

Mobile device number two's local presenter receives a Create message from the global presenter. The local presenter generates a Create operation and passes it to the local model. The local presenter generates a Read event and passes it to the view. The view displays the Read event to the user. The user agrees to download the video. The view generates a Read event and passes it to the local presenter. The local presenter generates a Read message and passes it to the global presenter. The global presenter interprets the Read message and sends the video to the local presenter. The local presenter passes the video to the local model. The local presenter generates a Read event. The local presenter passes this event to the view. The view displays the video to the user.

4.2 RESTful Publish/Subscribe Mobile Web Services

The REST architecture requires the service provider to implement each Web Service as a uniquely accessible resource. Also, the proposed system is required to store user profiles' data, media files, the metadata of these files and the shared channels' data; therefore the proposed system must create a uniquely accessible resource for each of these data objects. This thesis implements the middleware approach to map the CRUD messages' URIs to their respective resources. Figure 4-2 Display the middleware components.

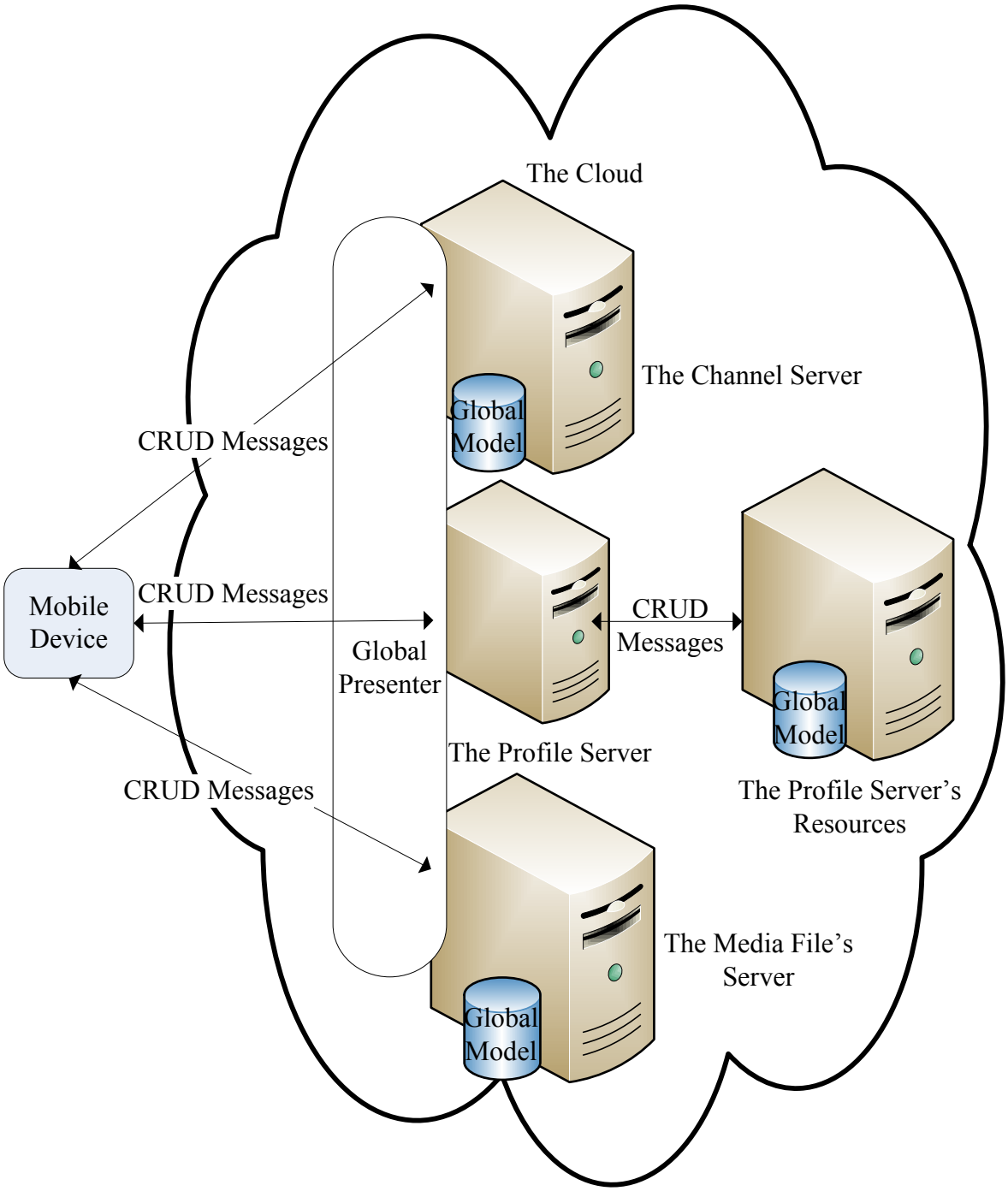

Figure 4-2 The profile server, the profile server's resources, the channel server and the media file's server exchanging CRUD messages with a mobile device.

The proposed system's middleware contains multiple components running in three servers.

The middleware servers' addresses are stored at the local presenter. These servers are:

4.2.1 The Profile Server

The profile server's components include a webserver and a node proxy server. The profile server is the access point to the following resources

- The message brokers,
- The users' profiles and,
- The media files' metadata.

The profile server hosts these resources on the resources' node. The structure of the data stored at the resource's node depends on the mobile device's type because different mobile devices have different media file's metadata; therefore the mobile device's registration process requires the user's profile from the local presenter because the message broker and the media files' metadata build and process the exchanged CRUD messages according to the user's profile resource. Also, the profile server creates these resources for each registering user; therefore the

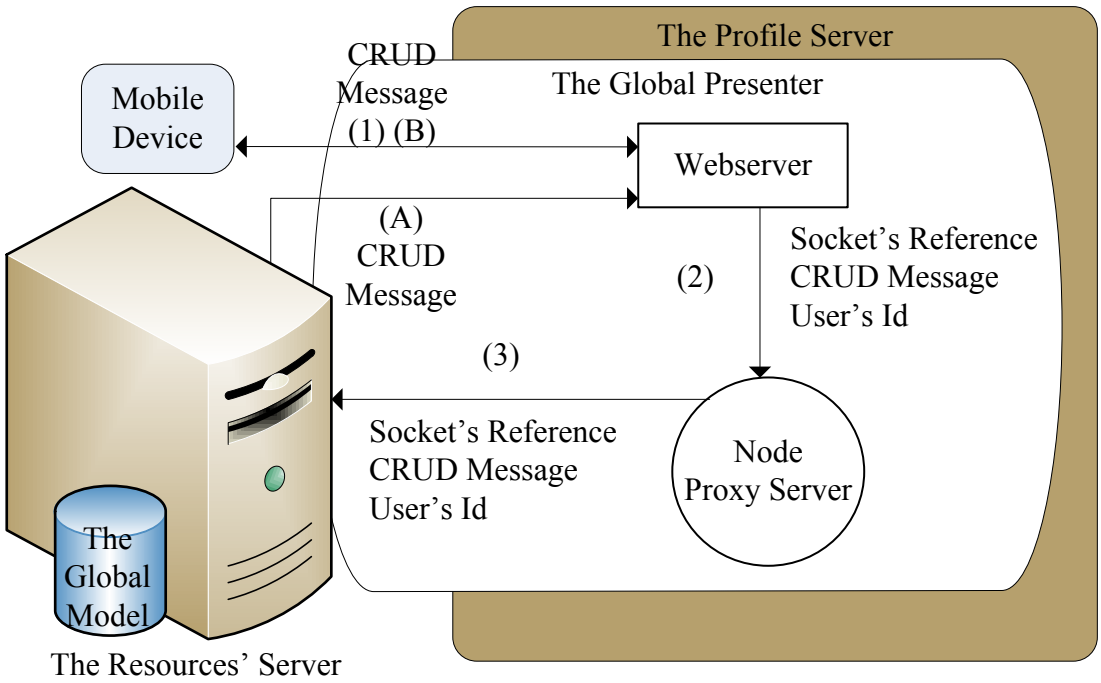

Figure 4-3 The profile server's architecture with the travelling paths of the CRUD messages.

local presenter generates a unique id to identify the registering user. Also, The CRUD message's URI must include that id and a reference to the receiving resource. Figure 4-3 shows the profile server's components and the CRUD messages' travelling directions. Steps 1-3 illustrate the flow of the CRUD messages from the mobile deice to the Resources' server. The steps A and B illustrate the flow of the CRUD messages from the Resources' server to the mobile deice.

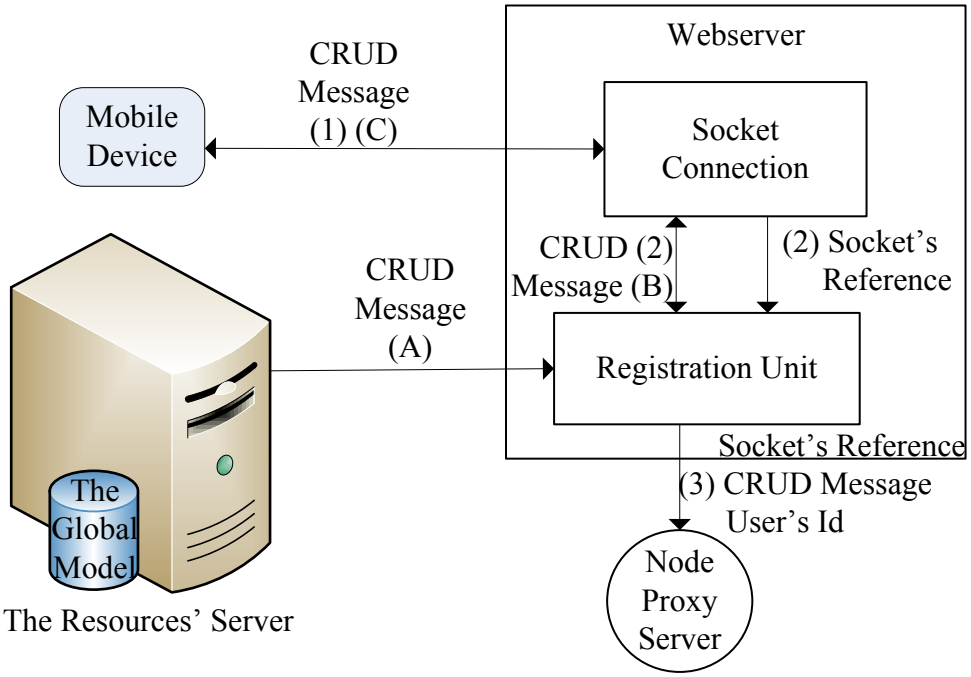

Figure 4-4 The webserver connecting the node proxy server to the mobile device with the travelling paths of the CRUD messages.

The entry point of the profile server must be accessible and compatible with all the connecting mobile devices; therefore the proposed system implements a webserver as the entry point. This webserver connects the node proxy server to the local presenter. Figure 4-4 shows this webserver's design. Also in this figure, the steps 1-3 and A-C are the CRUD messages'

travelling paths toward the resources' server and the mobile device respectively. This webserver is responsible for:

- Receiving the CRUD messages from the mobile device using the socket connection;
- Passing the CRUD messages to the registration unit;
- Sending the resources' CRUD messages through the socket connection to the mobile devices; and
- Notifying the registration unit when the socket connection is disconnected.

This webserver opens a socket for each connecting mobile device. Also, this webserver starts a registration unit for each socket connection; therefore this webserver is required to scale to the number of the connected devices. Also, this webserver provides a reference of the socket connection to the registration unit. This webserver uses this reference to send CRUD messages to their intended mobile device.

The registration unit extracts the URI from the CRUD message. The registration unit orders the node proxy server to check the validity of the URI:

- If the node proxy server has located the resource, then the registration unit orders the node proxy server to pass the CRUD message and the socket's reference to the resource proxy server running at the resources' node,
- If the node proxy server did not locate the resource, the message is a create message and the URI points to a user's profile resource; then the registration unit orders the node proxy server to start a new registration process at the resources' node and pass the create message and the socket's reference to the new process otherwise,
- The registration unit orders the webserver to return an error message to explain that no such resource was found and to disconnect from the mobile device.

The registration unit maintains the user's unique id in its state. Also, when the mobile device is disconnected, the registration unit envelops the webserver's notification in a CRUD message. The URI of this message contains the user's unique id and the message broker's resource id. The node's and the resource's proxy servers rout this message to the specified message broker.

4.2.1.1 The Profile Server's Resources

The node connected to the profile server maintains a resource proxy server and a resources' index in addition to the resources themselves. This proxy server is the entry point for the profile server's CRUD messages. This proxy server extracts the URIs from the incoming CRUD messages and maps these messages to their respective resources accordingly. Also, the resource

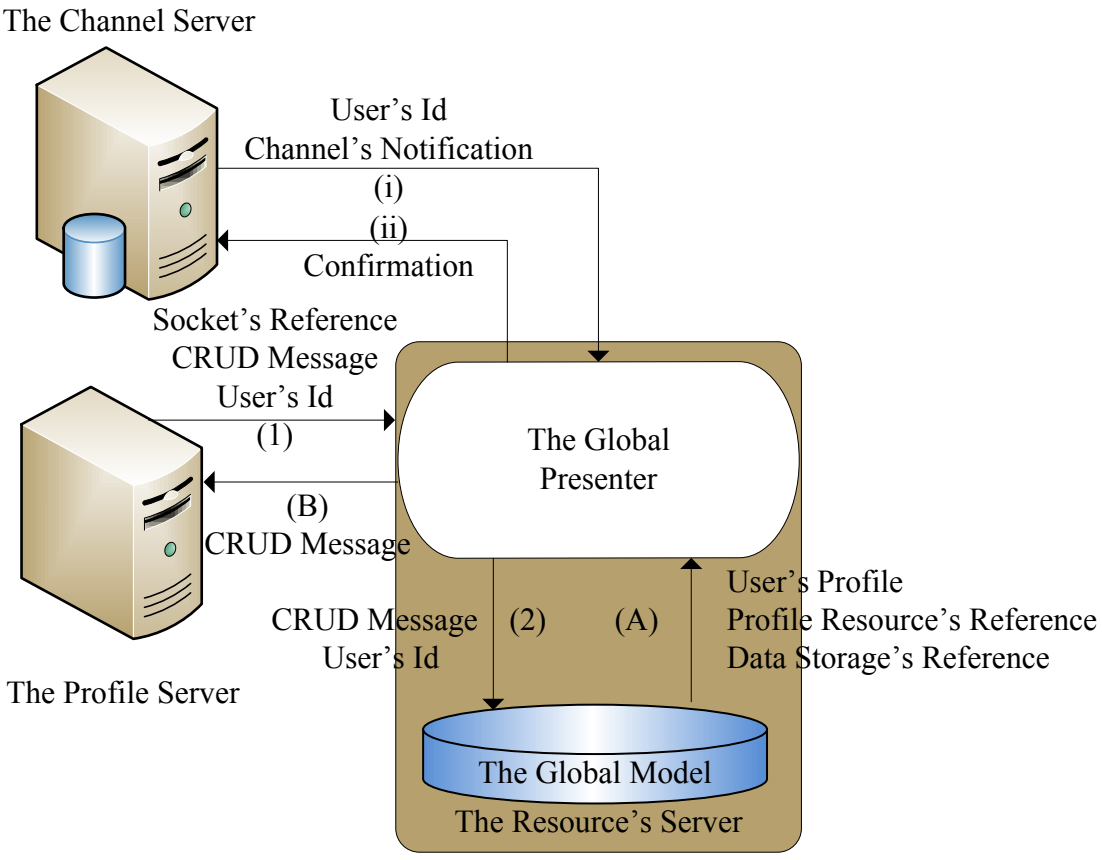

Figure 4-5 The resources' server with the travelling paths of the CRUD messages and the channel's notifications.

proxy server creates new user's profile, message broker and metadata resources for each registering mobile device. The resource proxy server stores a reference to these resources in the resources' index. Figure 4-5 shows the profile server resource's components. Also in this figure,

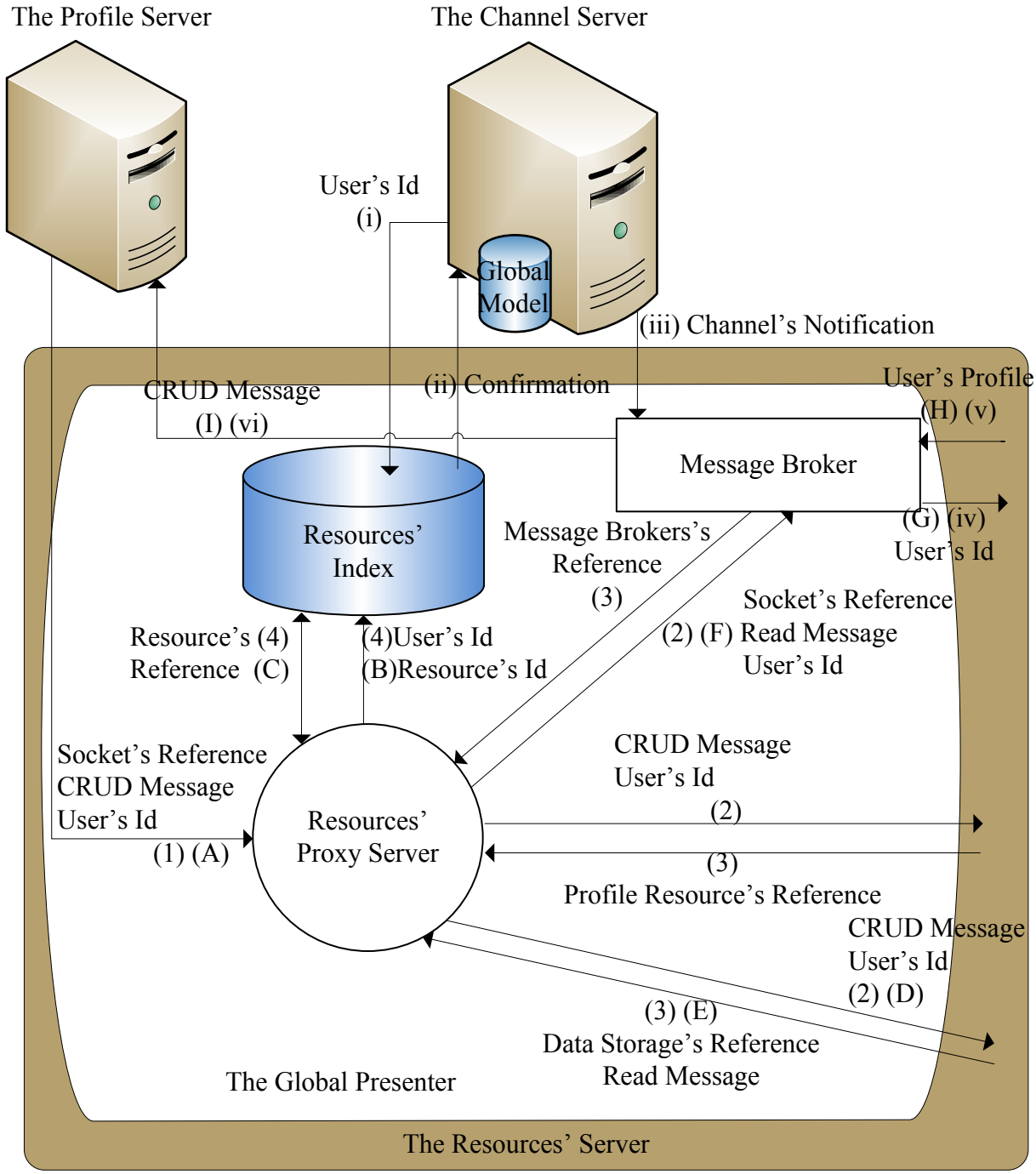

Figure 4-6 The global presenter's components at the profile server's resources with the travelling paths of the CRUD messages and the channel's notifications.

the steps 1-3 and A-C are the CRUD messages' travelling paths toward the global model and the profile server respectively. The steps i-ii are the travelling path of channel's notification and its confirmation from and back to the channel's server. The architecture of this thesis models the resource proxy server, the resources' index, the message broker and the profile server node as parts of the global presenter and the other resources as parts of the global model. Figure 4-6 shows the components of the global presenter at the profile server's resources with three data travelling paths:

- The steps 1-4 represent the registration operation of a new mobile client.
- The steps A-I are the CRUD messages' and their responds' travelling paths toward the global model and back to the profile server respectively.
- The steps i-vi are the travelling path of channel's notification to the profile server.

The user's profile resource maintains a description of the structure of the data objects stored by the other resources and the structure of the exchanged CRUD messages. This description is expected to vary depending on the type of the mobile device; therefore this resource requires flexible data storage. This resource's URI includes the user's unique id because for each user there is one profile resource. The content of this resource is stored as a record in the user profiles' data storage. The other resources query the resources' index for a reference to that record using the URI of this resource. The user profiles' data storage supports create, read and update operations.

The message broker implements the notification mechanism of the publish/subscribe communication scheme. There is one message broker for each mobile device. The message broker maintains a reference to the webserver's socket connection in its state. The message broker receives notifications regarding the mobile device's connection state from the registration

unite. The message broker receives the channel resource's notifications. The message broker envelopes the channel resource's notifications in a CRUD message compatible with the user's profile. The URI of this message contains the socket connection's reference and the channels id. The message broker stores these notifications until the mobile device is connected; therefore the message broker requires flexible data storage because the structure of these notifications varies according to the mobile devices' types. Furthermore, the message broker implements the time and synchronization decoupling between the publisher and the subscribers. The message broker's

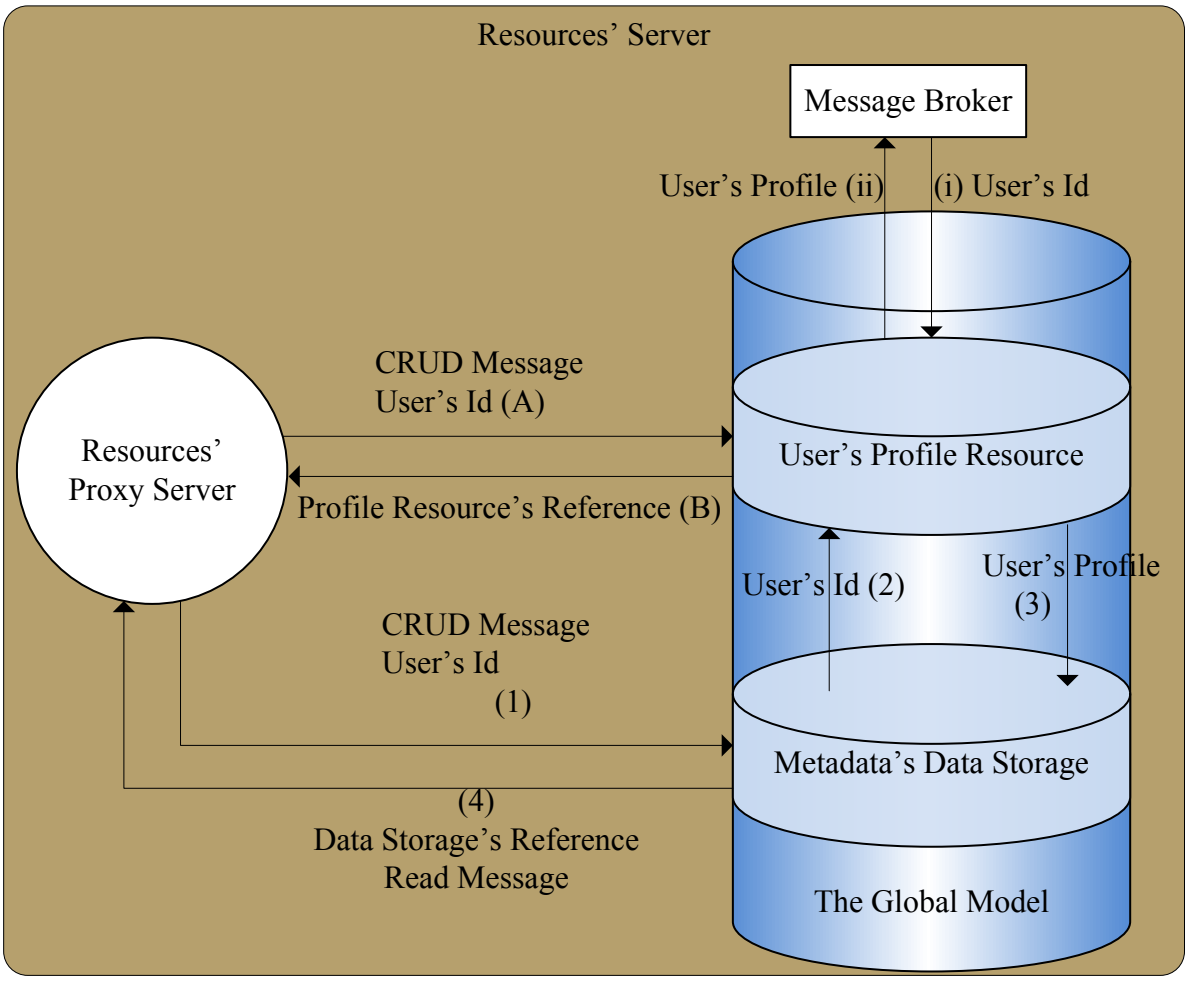

Figure 4-7 The global model's components at the profile server's resources with the travelling paths of the CRUD messages and the channel's notifications.

data storage supports create, read and delete operations. The media file's metadata resource is created when the user registers in the proposed system. This resource represents a cloud-hosted storage for the data objects stored at the local model. These data objects contain the additional data collected during the media file's creation process (such as the Global Positioning System's (GPS) coordination values). There is one metadata resource for each mobile device; therefore this resource's URI includes the user's unique id in addition to the metadata resource's id. Also, only the user who generated the media files has access to the media files' metadata because some of the collected data are considered private. Figure 4-7 shows the components of the global model at the profile server's resources. In this figure, the steps 1-4 and A-B are the CRUD messages' travelling paths from and back to the resources' proxy server. Also, the steps i-ii shows the message broker retrieving the user's profile. This resource receives CRUD messages generated by the local presenter. These messages synchronize the data stored at the local model with the data stored in this resource's data storage; therefore in case of data loss, this resource provides a backup plan for the media files' metadata. The metadata's structure and content varies among the different types of mobile devices and among different types of media files from the same mobile device; therefore this resource requires flexible data storage. The media file's resource creates a record for each uploaded media file. The references to these records are expected to be dynamic because the more new sensors will be embedded in mobile devices, the more new types of data will be collected by the mobile applications; therefore this resource's data storage implements the multi-term indexing approach. Meaning, a reference to a stored record contains a list of multiple values. These values are extracted from the stored metadata record. The user's profile contains the list of the data fields that points to these values. This list can be updated according to the needs of the mobile application at that time. Some of the

currently expected data fields are: file titles, upload date and file names because file names are unique for one user.

The profile server implements the push communication style to send the notifications to the connected mobile devices. Also, the profile server implements the REST architecture restrictions by mapping the URIs to their respective resources and maintaining the resources' states. However, the profile server does not implement a fully stateless communication because the message broker maintains the mobile devices' connection state (connected or disconnected). Also, the profile server implements the content-based publish subscribe system because the user's profile defines the structure of the exchanged messages.

When a mobile device connects to the profile server for the first time, the mobile device provides its user's profile wrapped in a create message. The profile server extracts the user's id from the create message. The profile server uses the user's id to checks weather this user is connecting for the first time by consulting resources' proxy server. The profile server forwards the create message, a reference to the mobile device's connection and the user's id to the resources' proxy server. The resources' proxy server passes the create message to the user's profile resource, creates a new metadata's data storage and creates a new message broker. The resources' proxy server receives a reference for these three new resources. The resources' proxy server stores these references in the resources' index as values by using the user's id and the resource's id (such as the resource's name) as two secondary keys. The message broker maintains the reference to the mobile device's connection for future interaction with the mobile device. The user's profile resource stores the user's profile as a value by using the user's id as a key. The metadata's data storage configures its table according to the user's profile.

When a registered mobile device connects to the profile server, the mobile device forward its user's id wrapped into a read message. The profile server extracts the user's id, collects the reference to the mobile device's connection and forwards them to the resources' proxy server. The resources' proxy server retrieves the reference to the message broker that is responsible for this mobile device from the resource's index using the user's id and the message broker's id. The resources' proxy server notifies this message broker that the connection with the mobile device was reestablished and forwards the new connection reference to this message broker. The message broker forwards the undelivered notifications to the mobile device using the new connection reference as CRUD messages. The message broker retrieves the user's profile server from the user's profile resource using the user's id and builds the CRUD message according to this profile.

When a mobile device passes a metadata-related CRUD message (such as a create message that contains the metadata of a new joke video), the profile server extracts the user's id and forward this message the resources' proxy server. The resources' proxy server retrieves the reference to the metadata's data storage of this mobile device from the resources' index using the user's id and the id of the metadata's data storage. The resources' proxy server forwards the CRUD message and the user's id to the metadata's data storage. The metadata's data storage retrieves the user's profile from the user's profile resource using the user's id. If the CRUD message is a create, update or delete message, then the metadata's data storage applies the changes to the metadata records using the user's profile. Otherwise, if the CRUD message is a read message, the metadata's data storage uses the user's profile and read message's query to search through the metadata records. The metadata's data storage sends the result of the query to the message broker through the resources' proxy server. The message broker either forwards the

results as a CRUD message to the mobile device through the profile server using the connection reference, or maintains the results until the mobile device reestablishes the connection. The message broker retrieves the user's profile server from the user's profile resource using the user's id and builds the CRUD message according to this profile.

4.2.2 The Channel Server

The channel server implements the topic-based publish/subscribe communication scheme by assigning a channel resource for each topic. These channel resources are created by the publishers to share the publishers' media files with the subscribers of these channel resources; therefore this server expects the incoming CRUD messages to have a unified structure among all the registered mobile devices. The URI of these messages must include the user's unique id. The channel server's components include a webserver, a channel proxy server and a group of channel resources. The channel resources are available to the mobile device on demand to avoid overwhelming the mobile device with irrelevant channels. Also, the channel server provides a list of the available channels to the user. This list serves as a channel discovery mechanism; therefore the channel server must implement the pull communication style. Meaning, the requesting mobile device must wait until the server respond back with the result of the request. Also, the requesting mobile device is expected to be connected to the channel server until the respond is sent back; therefore the channel server's webserver implements a fully RESTful Application Interface (API). Figure 4-8 shows the channel server's components. In this figure, the steps 1-8 are the CRUD messages' travelling path from and back to the mobile device.

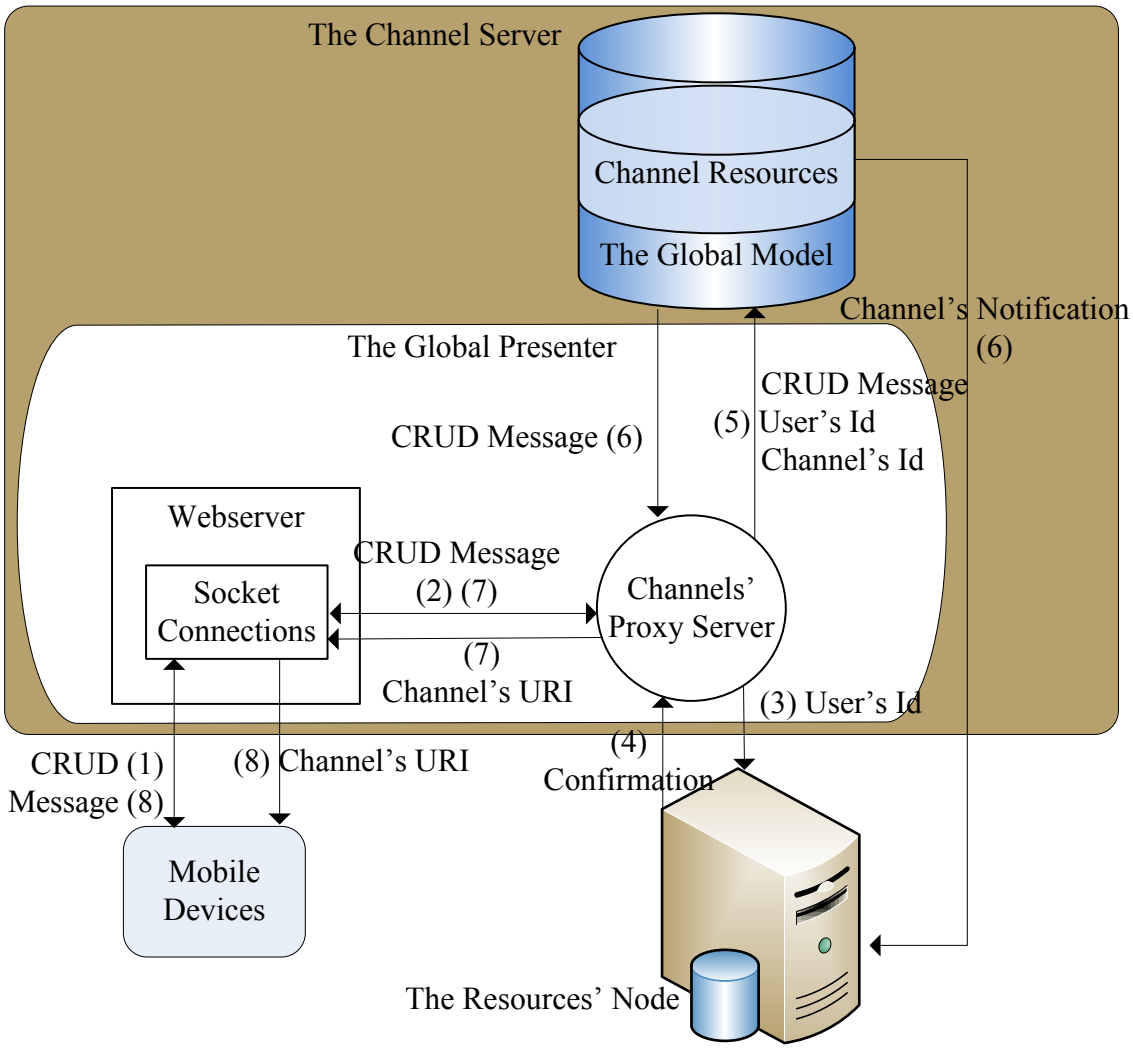

Figure 4-8 The channel server's components with the travelling paths of the CRUD messages and the channel's notifications.

The entry point of the channel server is a webserver. This webserver connects the channel proxy server to the local presenter. The responsibilities of this server include:

- Receiving the CRUD messages from the mobile device;
- Passing the received CRUD messages to the channel proxy server;
- Receiving the result of the CRUD message from the channel proxy server; and
- Sending the result of the CRUD message to the mobile device.

This webserver opens a socket for each connecting mobile device. Also, this webserver starts a channel proxy server for each socket connection; therefore this webserver is required to scale to the number of the connected devices. Also, this webserver uses a synchronous processing style. Meaning, this webserver waits for the channel proxy server to respond back with the result as long as the mobile device is connected.

The channel proxy server extracts the URI from the incoming CRUD message. The channel proxy server responsibilities include:

- Validating the registration of the mobile device by querying all the profile server resources' indexes using the user's unique id;
- Serving the list of the available channel resources' URIs;
- Creating new channel resources;
- Registering mobile devices to the channel resources;
- Validating the registration of the mobile device to a specific channel using the user's unique id;
- Passing the CRUD message to the channel resource using the channel id; and
- Generating error messages for the channel server's webserver.

This server provides a registration mechanism for the created channels. During the publication of a new channel resource, the creator of the channel specifies the new channel's security level. The security level is identified using two variables and has four possibilities:

- Subscribing and publishing to this channel is public. Meaning, any user can publish and subscribe to this channel;
- Subscribing to this channel is public but publishing is private. Meaning, any user can subscribe to this channel but only this channel's creator can publish to it;

- Subscribing to this channel is private but publishing is public. Meaning, the creator of the channel decides who can subscribe to this channel. The creator and the subscribers of this channel can publish to it; and
- Subscribing and publishing to this channel is private. Meaning, the creator of the channel decides who can subscribe to this channel. Also, only this channel's creator can publish to it.

When one of the channel's security variables is set to private, the channel proxy server refuses the attempted action unless the user's unique id is included in the registered users' list. However, only the channel's creator is allowed to remove the channel's published materials. The channel proxy server registers new users by passing a channel registration notification to the channel resource. This notification contains the channel id and the registering user's unique id.

The channel proxy server fulfill the space decoupling requirement of the publish/subscribe system because the publisher and the subscribers communicate using the channel id without the need for the id of the channel's creator.

Each channel resource's state contains the channel's unique id, the channel creator's id, a set of registered users' ids and a set of media files' URIs. When a publisher shares a new URI, the channel resource sends update notifications to the registered users' and the channel creator's message brokers. Also, the channel resource sends the registration notifications to the channel creator's message brokers. The message brokers' notifications are sent as update messages. These messages' URIs contain the receiving user's id and the message broker's resource id.

The user must create a channel before sharing the uploaded media files with the other users. When the mobile device sends a CRUD message (such as create a new channel) to the channel server, the CRUD message must contain the channel's id and the user's id. If the CRUD message is:

- A create message then the message will contain the entire channel's data.
- An update message then the message will contain only the updated fields.
- A read message then the message will contain the query.
- A delete message then the message will be empty.

The webserver receives the CRUD message and forward this message to the channel's proxy server. This proxy server extracts the user's id and the channels id from the message. This proxy server confirms the registration of the user by sending the user's id to the resource's server. This proxy server forwards the user's id, the channels id and the CRUD message to the channel resources. The channel resources store the channel's data as a value using the channel's id and the user's id as two secondary keys. If the CRUD message requires a respond then the channel resources respond contains the channels data or a list of channels. The channel's proxy server generates a URI for every included channel and add it that respond. The channel's proxy server forwards the CRUD message to the mobile device through the webserver. The mobile device uses these URIs to reference these channels during the publication of new media files. Also, (when requested) the channel's proxy server provides the registered mobile devices with a list of all the channels that are available to these mobile devices' users.

4.2.3 The Media File's Server

The media file's server is a cloud-based file hosting service. This server's components include a webserver, a file proxy server and a group of file resources. Figure 4-9 shows the media file server's components. In this figure, the steps 1-8 are the CRUD messages' travelling path from and back to the mobile device.

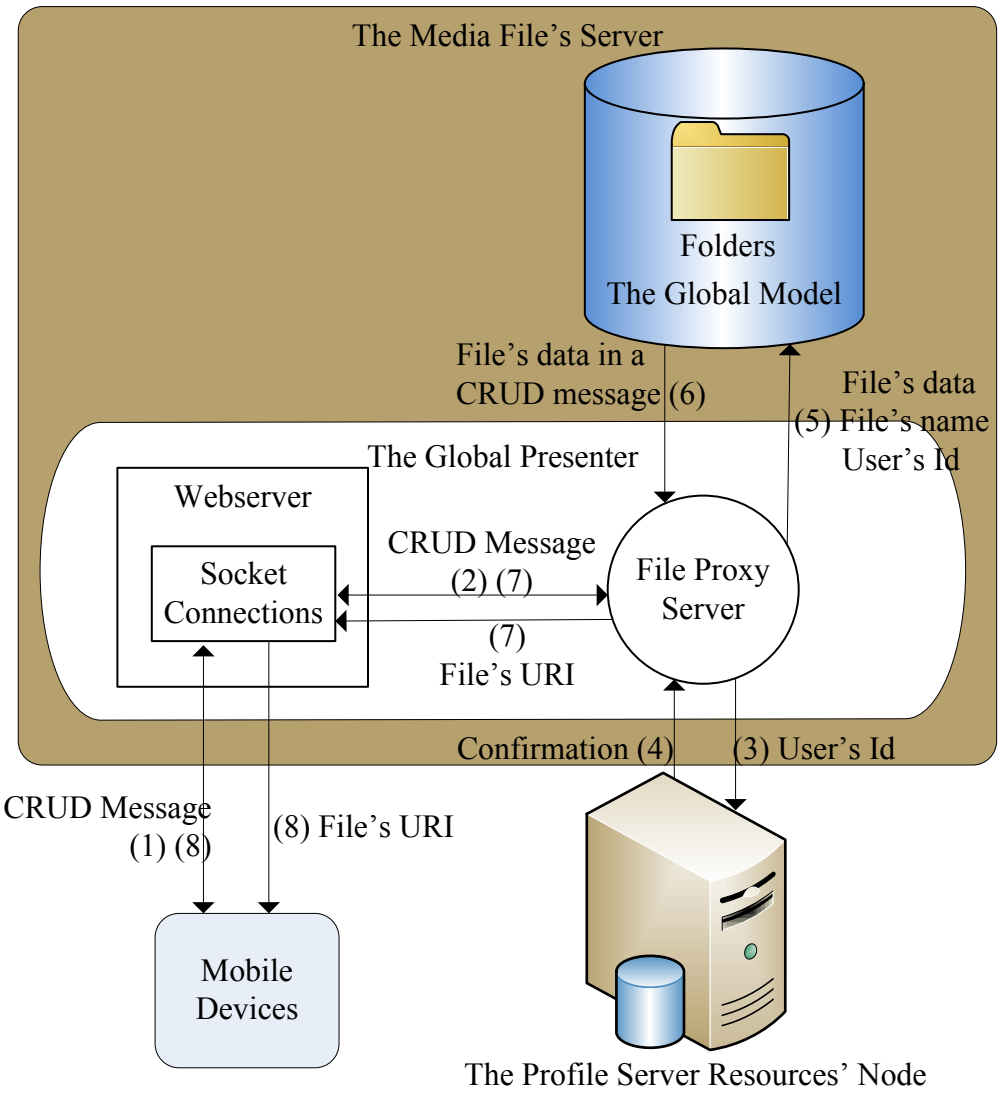

Figure 4-9 The media file's server with the travelling paths of the CRUD messages and the channel's notifications.

The local presenter uploads the locally generated media files to this server's webserver using RESTful create messages. This webserver creates a new resource for each of the uploaded files. Also, the local presenter receives a URI for each of these files. The local presenter incorporate this URI in the creation of the RESTful read and delete messages that manipulate the uploaded media file. The users share their uploaded files by publishing the media files' URIs at the channel server. Any registered mobile client can download these files using the published URIs.

When, the user uploads a file for the first time, this server's webserver creates a new folder in the server's file system. This folder's name is the user's unique id. This folder contains all the files uploaded by a specific user. This folder is considered as resource for the list of the media files uploaded by the user. Also, the uploaded media files have unique file names; therefore these files' URIs contain the user's id and the file's name. This server's webserver maps these URIs to their related media files.

When the mobile device sends a CRUD message (such as create a new media file) to the media file's server, the CRUD message must contain the media file's name and the user's id. If the CRUD message is:

- A create message then the message will contain the file's data.
- A read or a delete message then the message will be empty.

The webserver receives the CRUD message and forward this message to the media file's proxy server. This proxy server extracts the user's id and the media file's name from the message. This proxy server confirms the registration of the user by sending the user's id to the resource's server. This proxy server stores the user's id and the media file's name in a table and stores the multimedia file in the file system. The stored multimedia files can be retrieved from the file system using the user's id as a folder name and the file's name as the file's reference. If the CRUD message requires a respond then this respond contains the multimedia file or a list of multimedia files' names. The media file's proxy server generates a URI for every included multimedia file and add it that respond. The channel's proxy server forwards the CRUD message to the mobile device through the webserver. The mobile device share these multimedia files by posting these URIs at the user's channels. Also, (when requested) the media's proxy server

provides each registered mobile device with a list of all the files that have been uploaded by this mobile device.

4.3 Client Synchronization

The Model-View-Presenter overview in section 4.1 divides the mobile application into three major components. These components are

4.3.1 The Local Presenter

The global presenter communicates with the local presenter through a wireless network. Also, the global presenter is hosted in the cloud as a RESTful Web Service; therefore the local presenter is required to implement a RESTful web client that can open a socket connection with the cloud-hosted webservers. However, the global presenter communicates with the local presenter using two different communication styles: push and pull. The connection of the pull communication style is expected to be temporary and only lasts until the response is sent back. However, the push communication style requires the mobile device to be connected and waiting for incoming CRUD messages; therefore the proposed system implements two web client components in the local presenter:

- Pull-based web client: is responsible for sending CRUD messages to the channel server and uploading media files. The CRUD message handler creates a new instance of this web client for each socket connection.
- Push-based web client: is responsible for sending the data storage's synchronization CRUD messages to the profile server and receiving the message broker's notifications. This web client maintains an open socket connection with the webserver as long as the mobile device

is connected to the network. This web client notifies the CRUD message handler when the mobile device is disconnected or reestablished the connection with the profile server. Figure 4-10 shows the components of the local presenter.

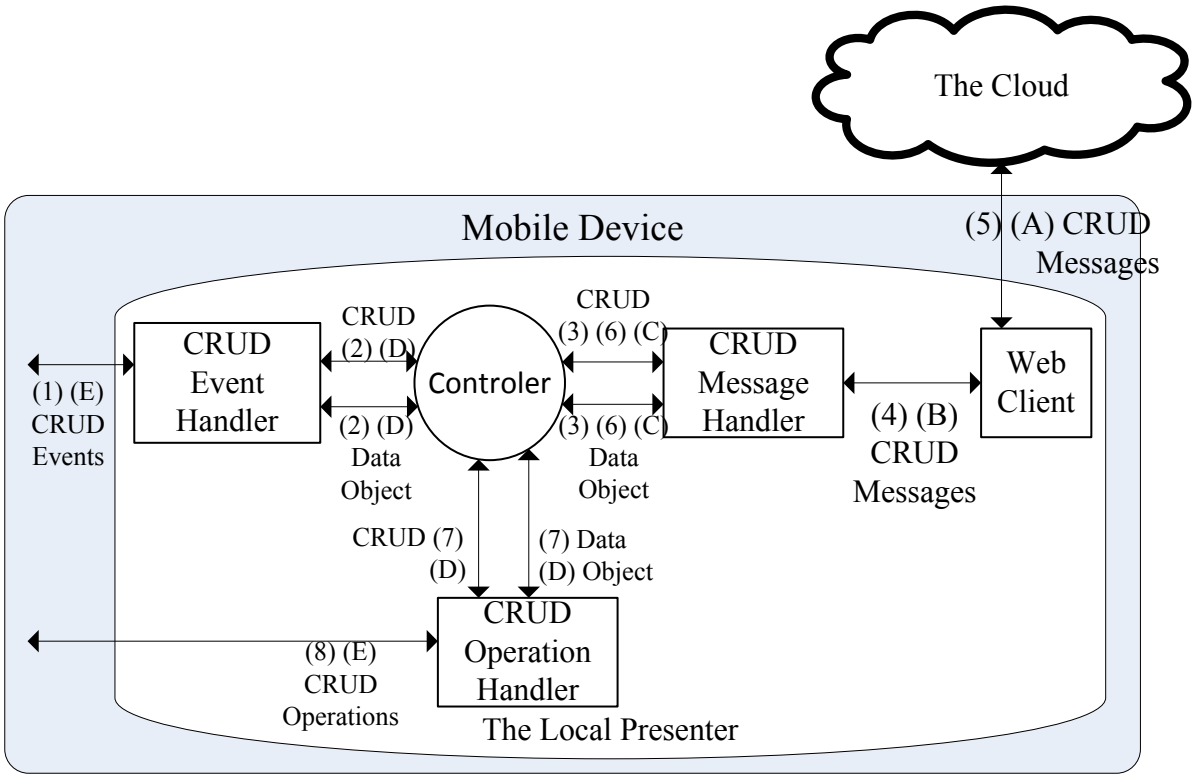

Figure 4-10 The local presenter's components.

The CRUD message handler is responsible for converting the method of the profile server's or the channel server's CRUD message to target the CRUD event handler. However, The CRUD message handler converts the method of the media file server's CRUD message to target the CRUD operation handler. The CRUD message handler extracts the method's data objects from the body of the incoming CRUD messages. Also, this handler converts outgoing methods and their data objects to CRUD messages. This handler maintains the addresses of the global presenter's webservers. Also, this handler extracts the user's and the resource's id from the outgoing data objects. This handler combines the webserver's addresses with the two ids to

create the URI of the outgoing CRUD messages. This handler resolves the problem of intermitted connectivity by maintaining one of three modes of operation depending on the network connection's status. These modes are:

- Online mode: is operational when the mobile device is connected to the network. This handler orders the web client to send the converted messages to the specified URI. Also, this handler passes the translated CRUD messages to the controller.
- Offline mode: is operational when the mobile device is disconnected from the network. This handler notifies the controller that the connection is lost. Also, this handler converts these methods to methods that target the operation handler. This handler passes the converted messages and the data objects back to the controller.
- Reconnection mode: is operational when the mobile device reconnects to the network. This handler requests all the rerouted methods and their data objects from the controller. This handler changes to online mode when the web client finishes sending the outgoing CRUD messages.

This handler receives the media file's metadata in a data object. This data object contains a monitoring variable. This variable is set to zero when the view collects the metadata. This handler extracts the media file's location from its metadata. This handler creates a CRUD message for the media file and for its metadata separately because they are hosted on different servers. This handler orders the web client to upload the media file before uploading its metadata because the media file's size is expected to be larger. There are three scenarios during the upload operations:

- The media file's uploading operation fails. This failure indicates that the connection is weak; therefore this handler notifies the controller that both uploading operations should be

delayed. Also, the CRUD operation handler maintains the monitoring variable's default value;

- The media file's uploading operation succeeds but the metadata's uploading operation fails. This handler notifies the controller about the failure. The metadata's uploading operation is delayed. Also, the CRUD operation handler sets the monitoring variable to one; and
- Both operations finished successfully. This handler notifies the controller about the success. Also, the CRUD operation handler sets the monitoring variable to two.

The controller is responsible for routing methods and their data objects to their targeted handlers. Also, the controller is responsible for initializing the application when it is started for the first time. The controller ensures that the CRUD operation handler is ready before routing any methods and their data objects. Also, the controller is responsible for resolving application errors. The controller ensures that the CRUD message handler finish its duties before committing the changes to the CRUD operation handler. The controller notifies CRUD operation handler if there was an error while uploading the media file or uploading the metadata of that media file. Also, the controller notifies the CRUD operation handler when the CRUD message handler changes to the reconnection mode.

The CRUD operation handler signals the model to start the initialization routine when the application starts for the first time. This handler translates the incoming methods and their data objects to CRUD operations. Also, this handler translates the responds to these CRUD operations to data objects. Furthermore, this handler requests the stored records that have their monitoring variable set to a value other than 2. This handler translates these records to data objects. This handler passes these data objects using the upload method to the controller. The controller passes this method and its data objects to the CRUD message handler.

The CRUD event handler is responsible for converting the CRUD event's user actions to methods that targets the CRUD message handler. Furthermore, this handler translates the CRUD event's data to data objects. This handler passes these methods and their related data objects to the controller. This handler translates the controller's methods and their data objects to CRUD events. This handler passes these CRUD events to the view.

The local presenter ensures that the local model is synchronized with the global model. The local presenter overcomes the problem of intermitted connectivity by monitoring the network connection and setting the monitor variable to a value that represents the current state of the uploaded data. Also, the local presenter is responsible for initializing the model when the user starts the application for the first time. The local presenter changes the structure of the exchanged data in the system to match its destination; therefore the view, the model and the global presenter can be implemented and upgraded independently. Figure 4-7 shows the mobile application's components.

4.3.2 The View

The view maintains the mobile application's graphical user interfaces. There is a graphical user interface for each resource type. The user interacts with these interfaces using CRUD interface elements. The view interprets the user's interactions as events. Also, the view collects the user's input data related to these events. These events and their related data invoke specific routines at the event handler.

The event handler changes the views according to the user's events. This handler loads the registration view when the user starts the application for the first time. This handler collects the user's profile data from that view. This handler ensures that the collected data from that view is

complete and the user's id is unique. The proposed system collects the user's email address as the user's id because email addresses are expected to be unique for each user.

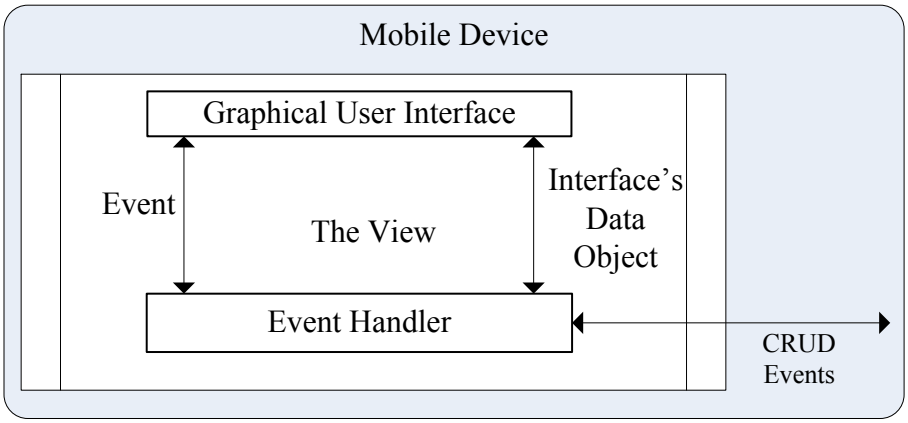

Figure 4-11 The view's components.

Also, this handler loads an intermediate view after unloading the registration view, when there is a user profile stored in the model and after unloading the other views. The intermediate view's interface elements generate events that load the resources' views. Also, this view is responsible for notifying the user when there is an update to the user's registered channel resource. However, this handler loads the media file's metadata view directly after unloading the media file's view. This handler collects the media file's metadata from the user and the mobile device. This handler converts the collected data and the user's events to CRUD events. Also, this handler passes the CRUD events to the controller. However, this handler converts CRUD events to events. These events are executed by the graphical interface. This handler extracts the CRUD events' data. This handler maps these data to the related interface elements. Figure 4-11 shows the components of the view.

4.3.3The Local Model

The local model maintains the user's data. However, the mobile device's data types are expected to defer among different mobile devices; therefore this model decides the structure of the stored data. The proposed system implements this model as the data storage operation handler in addition to the data storage itself to isolate the data structure from the data storage in terms of future data structure or data storage updates. Figure 4-12 shows the components of the local model.

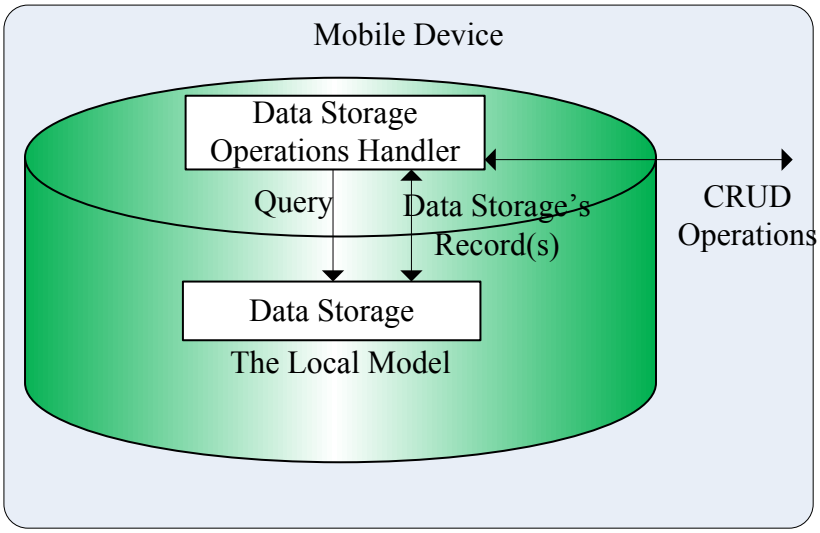

Figure 4-12 The local model's components.

The data storage operation handler initializes the model by creating an empty user's profile data storage and an empty media files' data storage. This handler maps CRUD operations to their equivalent data storage queries. Also, this handler translates data storage queries to CRUD operations. This handler passes these operations to the CRUD operation handler.

The data storages contain key-value tuples. The user's profile data storage stores the user's id as a key and the profile's data as a value. However, the media files' data storage stores the file's name as a key and the media file's metadata as a value. The proposed system collects the file's

title, the file's location according to the mobile device's file system, the file's URI and the file's monitoring variable as that media file's metadata.

4.4 Summary

This architecture presents a novel approach to synchronize the mobile's local storage with the server's global storage using RESTful publish/subscribe based Web Services. The architecture answered the three questions raised in Chapter 3:

- How to take advantage of both types of publish/subscribe systems?
    - The profile server's node decides the message broker's and the metadata's data structure according to the user's profile resource. Thus, implementing the content-based publish/subscribe communication scheme. Also, the channel server creates the channel resources according to their topic. Leading to the implementation of the topic-based publish/subscribe communication scheme.
- How to design the RESTful messages to synchronize the system's data storages and to work as a notification mechanism?
    - The local presenter communicates with the global presenter using RESTful CRUD messages. The local presenter with the help of the view translates the user's events to RESTful CRUD messages before transferring them to the global presenter. Also, the global presenter translates the channel's notifications to RESTful CRUD messages before transferring them to the local presenter.
- How to distribute the presenter between the server and the client?

- This architecture splits the presenter to a local presenter running on the mobile device and a global presenter running on three servers in the Cloud. The communication messages are distributed using a wireless network and a RESTful communication protocol.

The proposed system violates the REST architecture by maintaining the client communication state at the message broker. However, all the exchanged messages follow the REST architectural constraints.

This architecture leaves some open questions. The following represents questions that should be answered during the implementation phase of the proposed system:

- The data storage must support key-value tables; therefore which data storage should be used to implement the global model and the local model?
- The proposed system's webservers must scale to the number of connected mobile devices. Also, these servers must support the REST architecture; therefore which webservers should be used to implement the global presenter?
- The global presenter and the local presenter communicate using both the push and pull communication style. Although the RESTful HTTP protocol supports the pull communication style, it does not support pushing data directly from the server side; therefore which protocol should be used to exchange CRUD messages between the local presenter and the profile server?

CHAPTER 5
EXPERIMENTS

This chapter applies three experiments to test the synchronization capability, the scalability and the overhead of the proposed system. This chapter expresses the proposed system's synchronization capability by executing Chapter 2's scenario using the proposed system. Also, this chapter measures the proposed system's scalability by simulating concurrent clients initializing a connection then registering to the system. The third test measures the proposed system's overhead by generating multiple messages from a fixed number of concurrent clients for processing with and without including the webserver. These experiments aim to answer the questions that were raised at the end of Chapter 4.

5.1 Experimental Goals

The synchronization test focuses on implementing Chapter 4's architecture. This test executes Chapter 2's scenario using the implemented system to demonstrate the ability of a RESTful publish/subscribe system to synchronize mobile devices' local data store with a cloud-hosted data store. Also, this test studies the possibility of using WebSocket communication protocol to push data from the server to the client in the proposed system. Furthermore, this test studies the possibility of using Erlang's ETS and DETS tables [6] as a data storage that supports key-value tables. The scalability test measures the implemented system's limitation in terms of the number of concurrently connected mobile devices. This test demonstrates the system's ability to serve multiple clients concurrently. The overhead test measures the system's overhead caused by the webserver. This test measures the effect of the webserver on the turn-around time of the synchronization messages. The scalability and overhead tests study the capability of YAWS (Yet Another Web Server) [39] to handle multiple clients in the proposed system.

5.2 Experiment Setup

The global components of the proposed system are hosted in one of the MADMUC lab's machines to simulate a cloud-hosted machine. This machine has the following specifications:

Operating system:

Windows 7 Enterprise 32-bit, Service Pack 1

Hardware specifications:

Processor: Intel® Core™2 Quad CPU Q9400 @ 2.66GHz 2.67 GHz

RAM: 4.00 GB (3.37 usable)

Pre-installed applications:

Erlang 5.10/OTP R161

The global components of the proposed system are implemented using the Erlang programming language. Erlang is an open source programming language used to implement highly scalable robust distributed applications that runs in an Erlang virtual machine [6]. The system maintains two virtual machines. One of these virtual machines hosts a webserver called YAWS. YAWS is an open source highly scalable webserver implemented using Erlang[39]. YAWS supports RESTful HTTP API. YAWS is used as the webserver component for the profile server, the channel server and the media file's server. Also, this webserver supports the WebSocket communication protocol. This protocol enables two-way message exchange between the client and the server over a TCP/IP socket connection[35]. The proposed system uses this communication protocol to:

- Push notifications from the profile server's webserver to the client directly;

- Receive the client's profile; and
- Detect whether the client is online or offline.

YAWS server runs according to the following specifications:

Server:

Version: 1.96

Maximum number of connections: no limit

Number of virtual servers: 3

Erlang Virtual Machine:

Maximum number of processes: 1000000

Maximum number of simultaneously existing ports: 1000000

Maximum number of Erlang atoms: 5000000

In order to demonstrate the proposed system's horizontal scalability, the profile server's resources are hosted in an independent Erlang virtual machine. This machine represents the profile server's node described in Chapter 4. This machine runs according to the following specifications:

Maximum number of processes: 1000000

Maximum number of simultaneously existing ports: 1000000

Maximum ETS tables: 5000000

Maximum number of Erlang atoms: 5000000

The two virtual machines communicate using Erlang's remote procedure calls. The users' profiles, the media files' metadata and the channels data are stored as key-value tuples in two types of Erlang's NoSQL data stores called ETS and DETS. ETS and DETS tables are stored in the main memory and the secondary memory respectively. Figure 5-1 shows the experiment setup.

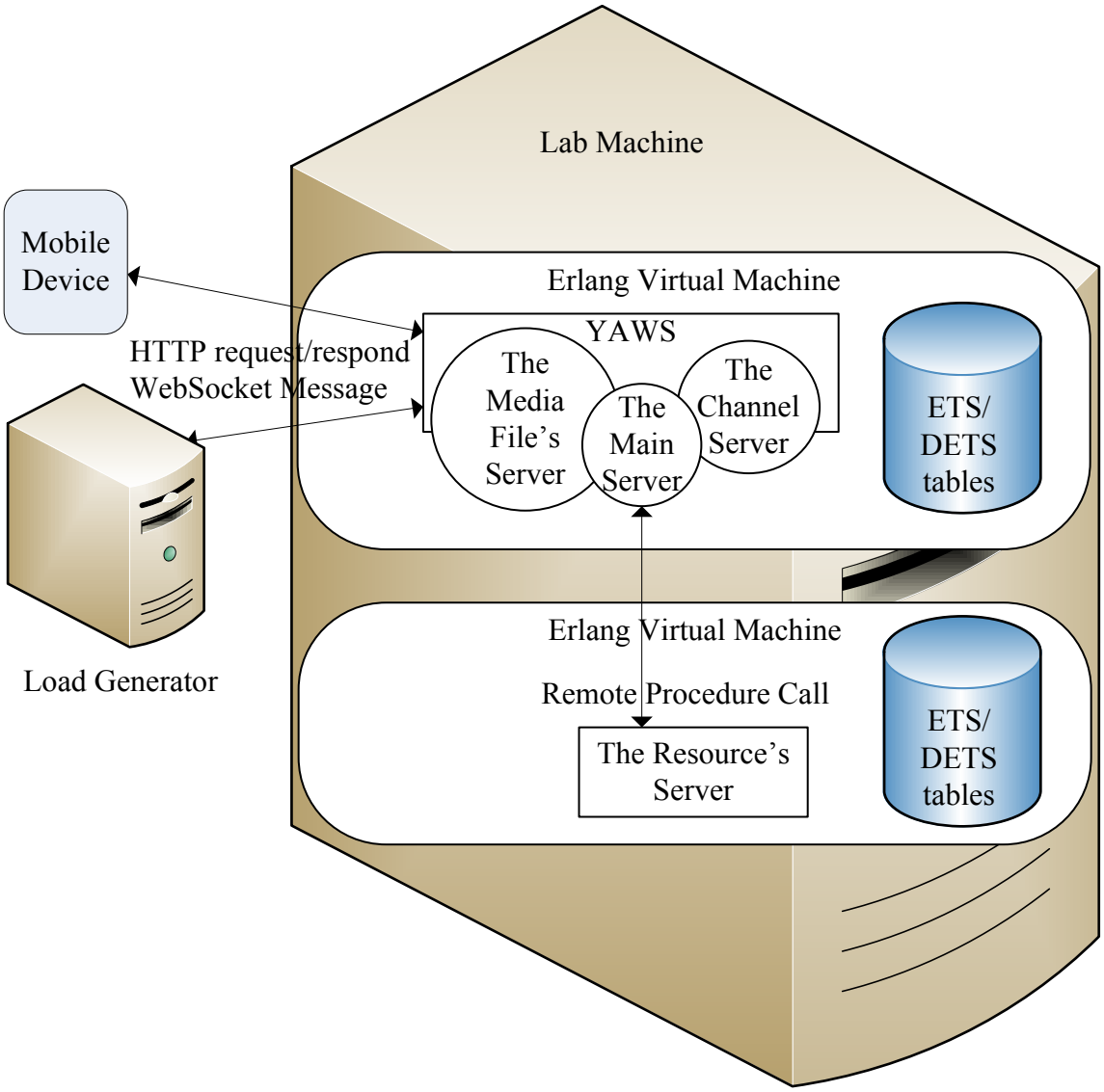

Figure 5-1 The experiment setup.

5.3 List of Experiments

The following is a list that explains the experiments. These experiments are conducted to test the synchronization capability of the system, the scalability of the server and of the architecture.

5.3.1 The Synchronization Test

The goal of this test is to run the system using different mobile devices; therefore The ASUS Nexus 7 tablet and the SAMSUNG Nexus S smartphone devices are used in this experiment. The tablet's operating system is the Android operating system version 4.2.2 and the smartphone's operating system is the Android operating system version 4.1.2.

The client side of the architecture is implemented using PhoneGap plugin for cross-platform mobile applications[27] and the "HTML5-based user interface system", jQuery Mobile [16]. The PhoneGap plugin and the jQuery Mobile interface are used to implement the local presenter and the view respectively. The local presenter required 1325 lines of JavaScript code. The view required 594 lines of HTML5 and JavaScript code. WebSQL database is used to simulate the mobile client's NoSQL data store because the mobile device's Android internet browser does not support IndexedDB; therefore the generated records are stored to and retrieved from the database using a list of fields. These fields simulate the IndexedDB's index. The local model required 667 lines of JavaScript code. This experiment executes the following steps:

1. Both mobile devices register in the proposed system;
2. The tablet creates a new channel;
3. The smartphone requests the channel list;
4. The smartphone registers for the created channel;
5. The tablet creates a media file;
6. The tablet uploads the media file to the server;

7. The tablet publishes the media file in the channel;
8. The smartphone is expected to receive a channel update notification; and
9. The smartphone downloads the media file.

This routine is executed using video, audio and image media files separately. The profile server is expected to notify the smartphone for each update as long as the WebSocket connection is established. Also, the profile server is expected to maintain the update notification until the connection is reestablished before notifying the smartphone.

5.3.1.1 Results and Discussion

This test was executed to study the possibility of running the implemented system on the Android tablet and smartphone devices and to demonstrate the synchronization capabilities of the implemented system. However, the Android changes from version 4.1 to version 4.2 by Google changed the default system directory of the tablet device to add support for multiple users[2]. This update caused the smartphone client application to have a missing-directory error. To overcome this problem, the system must be implemented to add the default system directory to the user profile, which will allow the application to adapt to the version differences; therefore the test was executed on the tablet device. Also, the channel server and its graphical user interface are still under development; therefore steps 2, 3, 4 and 7 of the synchronization test are postponed until the development of the channel server and its graphical user interface is completed. However, the application's synchronization capabilities were tested using the recovery routine, which is designed to recover the user's multimedia files when the data stored locally is lost. The profile's server required 230 lines of Erlang code. The global presenter at the resource's server required 446 lines of Erlang code. The global model i.e. the ETS/DETS tables required 463 lines of Erlang code. The media file's server required 242 lines of Erlang code. The

channel's server currently contains 471 lines of Erlang code. This test is executed according to the following steps:

1. The tablet mobile devices register in the proposed system.
2. The tablet captures a picture.
3. The tablet uploads the picture and its metadata to the file and profile server respectively.
4. I delete the locally stored picture and uninstall the local application from the tablet.
5. I reinstall and run the local application.
6. The tablet requests the list of the multimedia files' metadata records from the file server and store them in the local database.
7. The tablet downloads the missing multimedia files from the file server and stores them in the local file system.

   Figures 5-2, 5-3, 5-4, 5-5 and 5-6 show screenshots of the graphical user interface for the local application during the execution of the test.

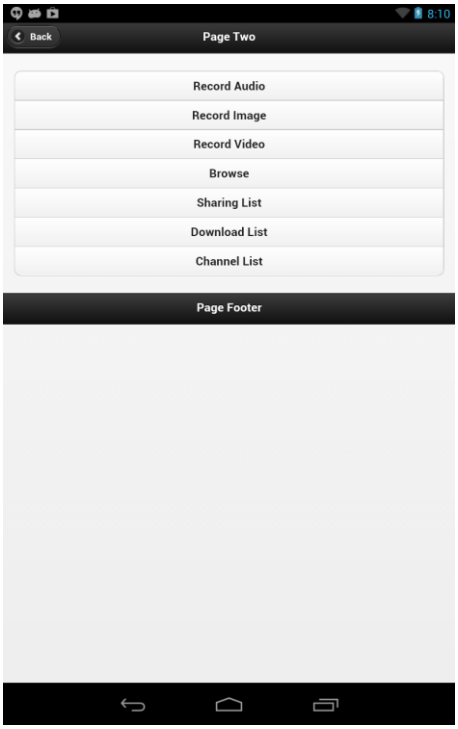

Figure 5-2 Step 1: the application's main menu registers the user automatically.

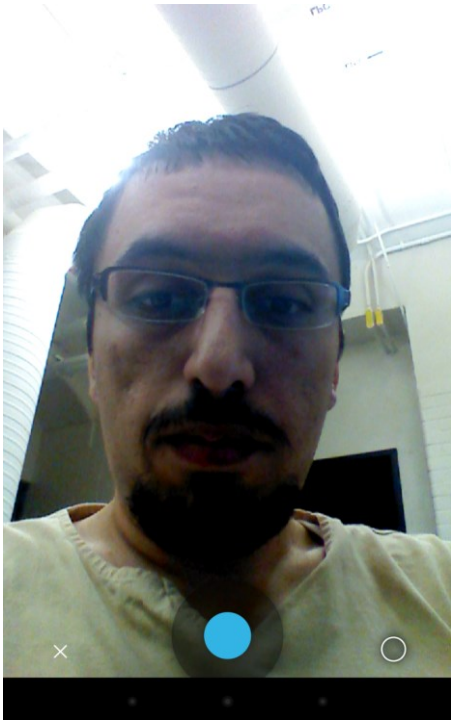

Figure 5-3 Step 2: the application captures a picture and stores it locally.

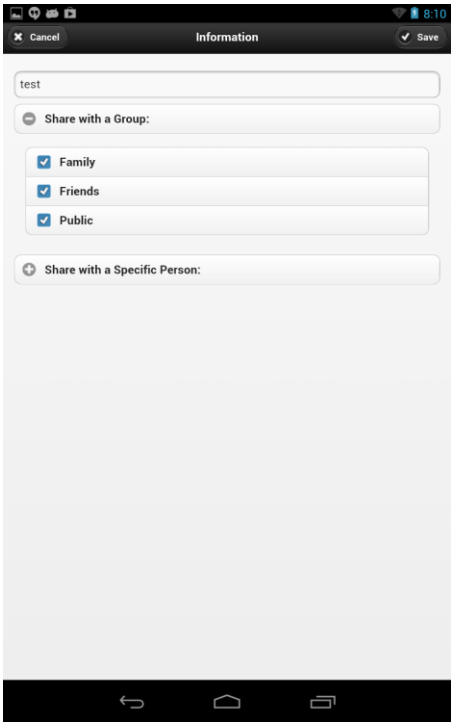

Figure 5-4 Step 3: after collecting the picture's metadata, the application uploads the picture and its metadata to the file and profile servers respectively.

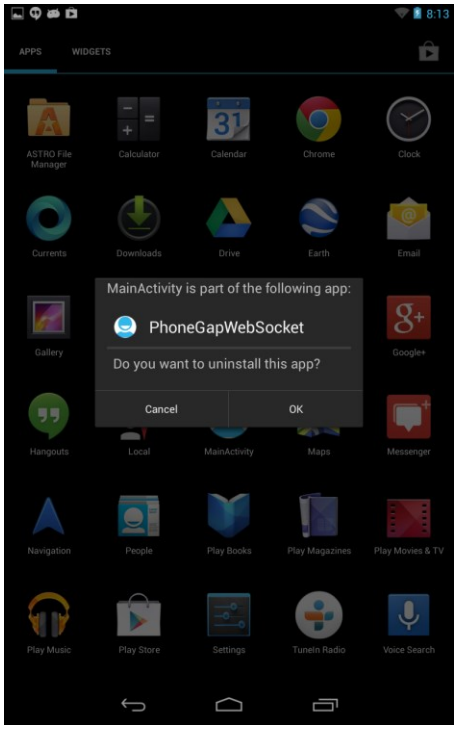

Figure 5-5 Step 4: The application is uninstalled.

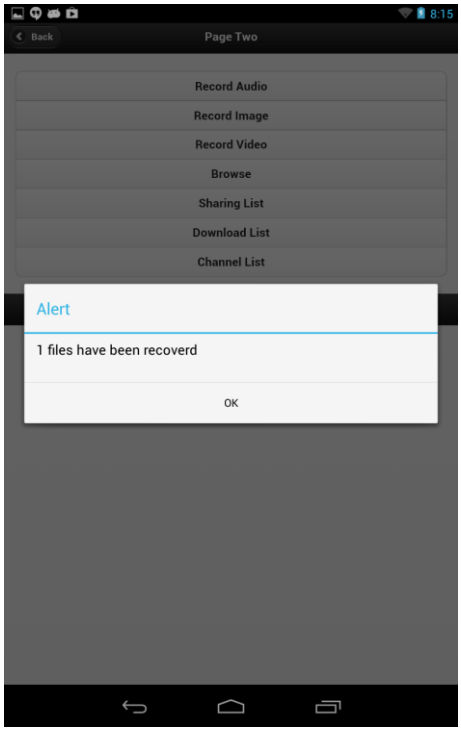

Figure 5-6 Step 5, 6 and 7: The application is synchronized automatically.

This experiment demonstrates the synchronization capabilities of the proposed system in the case of local data recovery. Also, this experiment reveals one of the problems that might face the developer during the implementation of cross-platform applications and suggests the user's profile approach, which is based on the content-based publish/subscribe approach as a solution to these problems. However, further experiments that test the notification synchronization and the connection loss synchronization cases are required.

5.3.2 The Scalability Test

This test measures the maximum number of concurrent users the proposed system can handle. An Erlang-based load generator is implemented to simulate a group of mobile devices connecting and registering in the proposed system concurrently. This load generator required 113 lines of Erlang code. The proposed system creates a user profile for each registering mobile device. This test is conducted by gradually increasing the number of the registering clients until the number of created user profiles does not match the number of registered clients. The load generator records the starting time and the respond time for each WebSocket message. The effect of increasing the number of the registered mobile devices is observed by calculating the time the

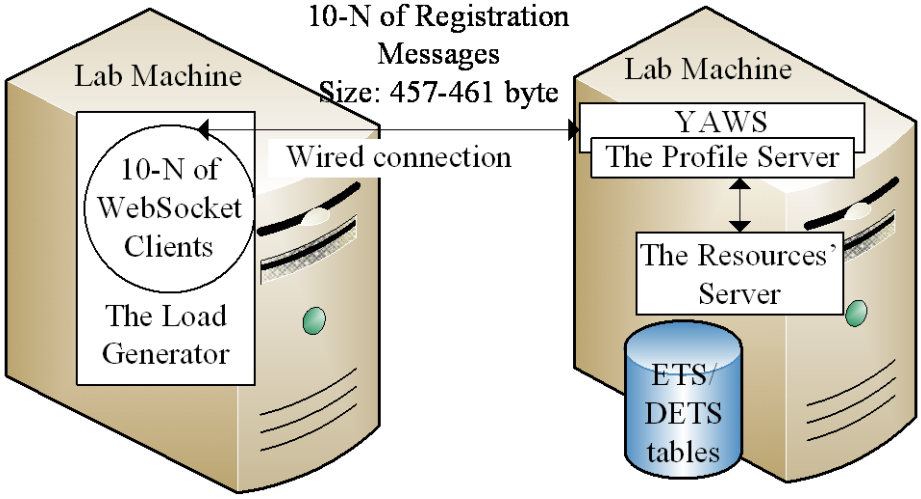

Figure 5-7 N of clients connecting to YAWS concurrently.

proposed system needs to register that number of connected mobile devices. Figure 5-2 shows N of clients connecting to the server concurrently.

5.3.2.1 Results and Discussion

The number of simulated WebSocket clients registering to the profile server started from 10 concurrent clients and incremented by 10 per test cycle until they reached 100 clients. Then, the increment increased to 50 concurrent clients per test cycle until they reached 650 clients. The load generator generated the test cycles and recorded the profile server's response time. The response time represents the amount of time needed to register all the simulated WebSocket clients in the system. To measure the system's performance at different workloads, the system's throughput was calculated for each group of concurrent clients. Generally, the throughput represents the number of processed commands during a static period of time. In this experiment, the throughput represents the number of registered clients during a period of one second. Each test cycle was repeated 4 times. However, some of the test cycles produced abnormal response time readings; therefore the test was repeated for the fifth time for the abnormal cycles. 3 out of 4 or 5 of the repeated test cycles with the lowest response time were used to calculate the mean response time and the throughput for that group of clients.

The maximum number of concurrent clients the server was able to register with no errors is 636 clients. Also, the maximum throughput is 202.841 simulated clients per second (Figure 5-8). Furthermore, the mean response time for all the selected test cycles is 3.054 seconds with a variance of 0.002. The maximum and the minimum response times among all the requests are 3.152 and 3.011 seconds respectively. Figure 5-9 shows a graphical representation of the mean, the maximum and the minimum response time for the simulated WebSocket client groups.

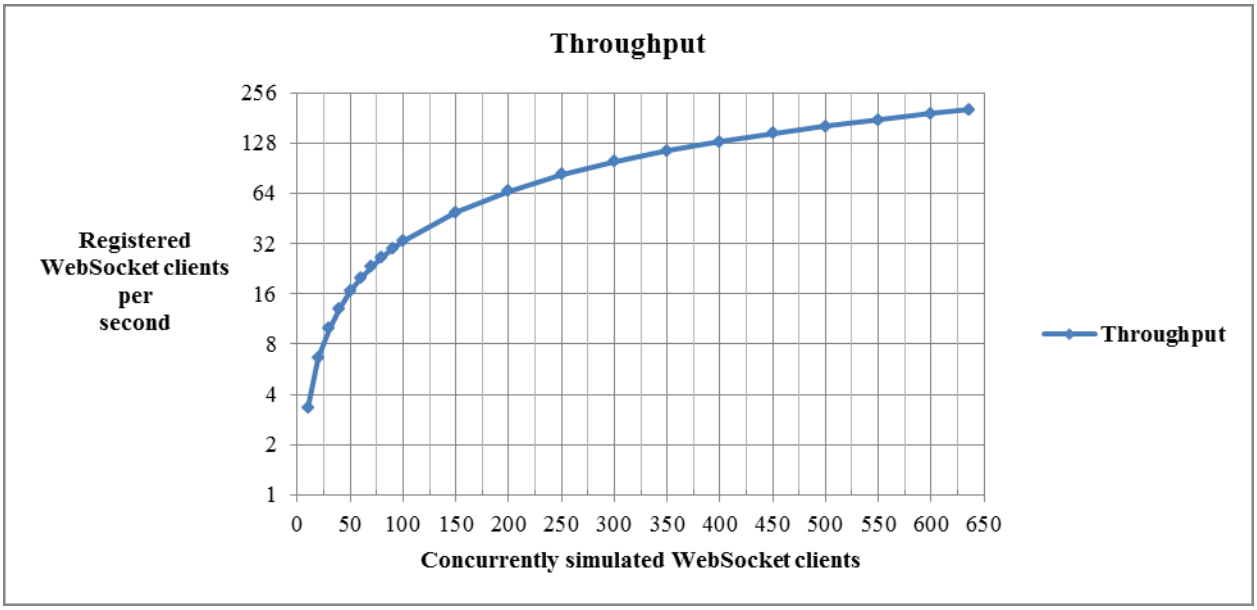

Figure 5-8 The throughput for the simulated WebSocket client groups.

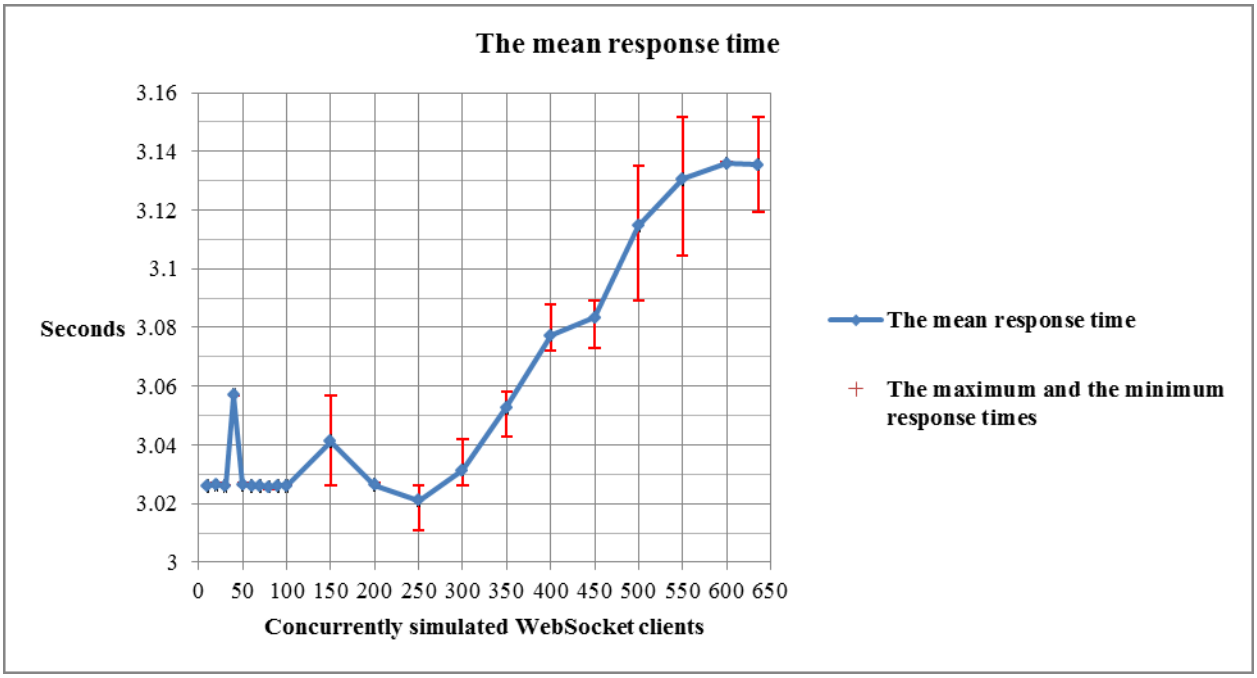

Figure 5-9 The mean response time for the simulated WebSocket client groups.

These results show that there is a relatively small difference in response time (±0.07 of a second) among the groups of the simulated clients. This small difference indicates that the server is scaling to the increasing number of simulated clients to a certain limit (in this case the limit is

636 clients). Also, as the number of clients increases from 10 to 300, the system maintains relatively stable response time (around 3.03 seconds). However, there are a couple of spikes and a drop in response time which can be caused by Microsoft Windows 7's background services because these services share the same resources such as the network bandwidth, the main memory and the secondary data storage with the implemented system. Also, as the number of clients increases from 400 until 636 clients, the response time increases which indicates that some of the clients' requests are pending at the web server's queue or the prototype is taking a longer time to register each client. The prototype is mainly memory bound because the ETS tables store their data on the main memory (Random Access Memory or RAM) and the DETS tables store their data on the secondary data storage. Adding faster main memory can reduce the time needed to store the ETS tables. Also, replacing the hard disk drive with a solid state drive can also reduce the time needed to store the DETS tables. The system is not bound to the processing power of the machine because it is an asynchronous system. However, increasing the size of the main memory is expected to increase the number of concurrent clients. In terms of bandwidth, currently available wired network interface controllers such as Ethernet are sufficient enough to serve large number of users.

5.3.3 The Overhead Test

This test measures the latency generated from using the webserver. This test includes two phases:

1. WebSocket messages are generated using the load generator. The load generator is running on a remote physical machine to simulate WebSocket clients. The load generator passes these messages to YAWS through a WebSocket connection. The load generator records the starting time and the respond time for each passed WebSocket message. This load generator

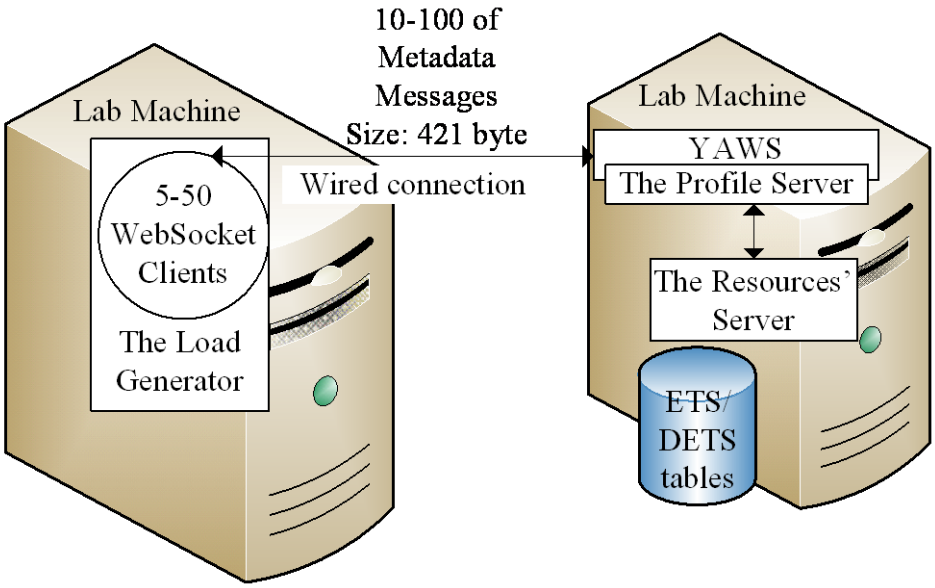

Figure 5-10 The load generator sending WebSocket messages to YAWS.

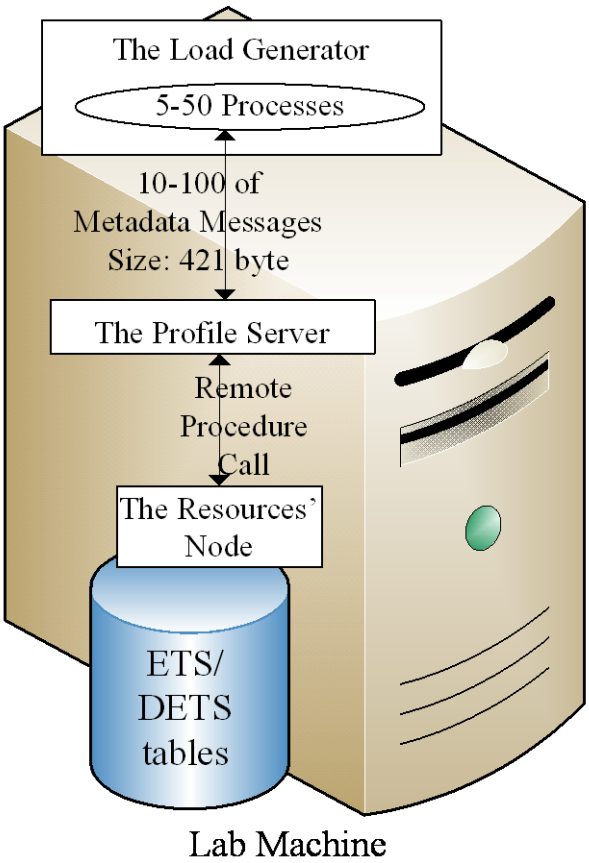

Figure 5-11 The load generator sending CRUD messages to the registration unit.

required 243 lines of Erlang code. Figure 5-10 shows the load generator sending WebSocket messages to YAWS.

2. CRUD messages are generated using the load generator. Figure 5-11 shows the load generator sending CRUD messages to the registration unit. The load generator is running on the same physical machine. The load generator passes these messages to the registration unit's process directly.The load generator records the starting time and the respond time for each passed CRUD message. This load generator required 240 lines of Erlang code.

5.3.3.1 Results and Discussion

- In phase 1, the number of simulated WebSocket clients sending WebSocket messages to the profile server started from 5 concurrent clients and incremented by 5 per group of test cycles until they reached 50 clients. The number of WebSocket messages sent during each group of test cycles started from 10 and incremented by 10 until they reached 100 WebSocket messages per client. Each WebSocket message contained a 421 bytes CRUD message. These

```
CREATE aladhami1@gmail.com/mediaStore/
        [{"fileName":"9w52nrnu58rxj001.3gp",
          "filePath":"file:///storage/emulated/0/mediaShare/9w52nrnu58rxj001.3gp",
          "title":"test",
          "sync":"1",
          "share":"1",
          "family":"1",
          "friends":"1",
          "public":"1",
          "emailList":"[]",
          "getURL":"HTTP://hashimpc.usask.ca:9090/17110730/mediaStore/9w52nrnu5
              8rxj001.3gp",
          "deleteURL":"HTTP://hashimpc.usask.ca:9090/deletefile/17110730/mediaStor
              e/9w52nrnu58rxj001.3gp"
        }]
```

Figure 5-12 An example of a CRUD message.

CRUD messages simulate the multimedia files' metadata. Figure 5-12 shows an example of a CRUD message used during the experiment. The resources' server recorded the amount of time needed to process all the WebSocket messages sent by each group of concurrent clients during each test cycle.

- In phase 2, the number of Erlang processes sending CRUD messages to the resources' server started from 5 concurrent processes and incremented by 5 per group of test cycles until they reached 50 processes. The number of CRUD messages sent during each group of test cycles started from 10 and incremented by 10 until they reached 100 CRUD messages per Erlang process. The resources' server recorded the amount of time needed to process all the CRUD messages sent by each group of processes for each test cycle.

  Figure 5-13 shows a graphical representation of the mean processing time collected in phase 1 and 2.

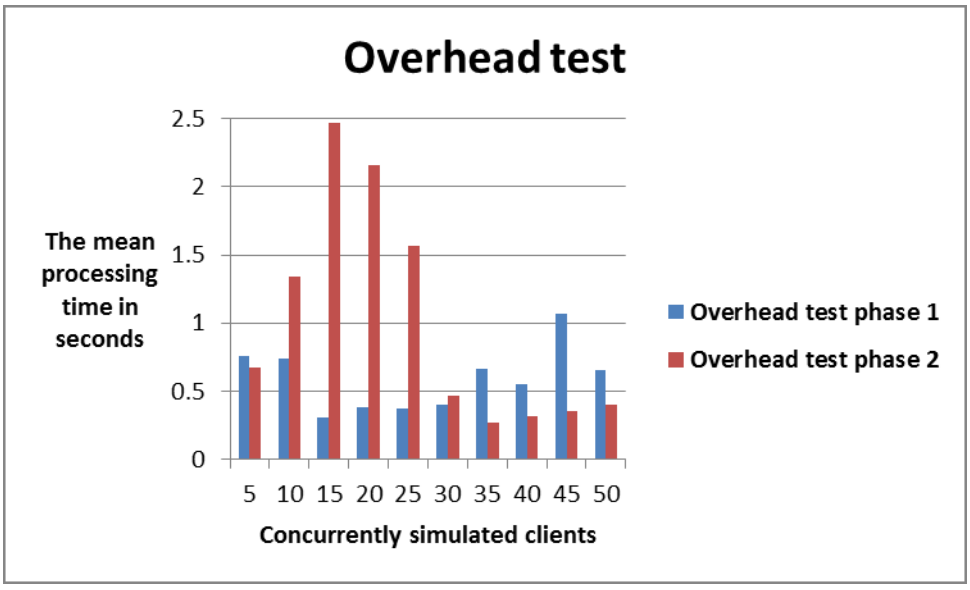

Figure 5-13 Phase 1 and 2 mean processing time.

The results of the two phases show that the number of concurrent clients has minimum effect on the mean processing time. This behavior is expected because of the scalability of the system.

Also, the mean processing time of all the test cycles is 0.58985933 and 1.00277975 seconds in phase 1 and 2 respectively. These readings indicate that YAWS webserver is more efficient than the load generator in terms of WebSocket message delivery; therefore a different approach for testing the overhead of YAWS's implementation of the WebSocket protocol is required.

5.4 Summery

This chapter explained the different types of experiments to measure the system's performance in terms of synchronization, scalability and overhead. This chapter tried to answer the following questions:

- Which data storage should be used to implement the global model and the local model?

  The synchronization test: A prototype application was implemented that utilized Erlang's ETS and DETS NoSQL tables as the global model. Also, this prototype used WebSQL database hosted in an Android tablet as the local model. This WebSQL database was used to simulate a NoSQL database. This prototype was exposed to a case of local data loss. This prototype was able to recover from this case by resynchronizing these databases using WebSocket messages. This experiment concluded that NoSQL databases can be synchronized using lightweight WebSocket messages. However, further experiments are required to test other synchronization cases such as the synchronization using the publish/subscribe-based notifications and the synchronization through an intermittent connection.

- Which webservers should be used to implement the global presenter?

  The scalability test: The prototype used YAWS as RESTful-webserver. However, the prototype is expected to handle large number of concurrent connections. To measure

YAWS's scalability in the prototype, a load generator was implemented to simulate multiple groups of concurrent WebSocket clients connecting to YAWS and registering in the system. This load generator and the prototype were hosted on two separate machines. This load generator measured the amount of time needed to register these WebSocket clients. The prototype was able to successfully serve a group of 636 clients in 3.136 seconds as a maximum limit. When the number of clients passed the maximum limit, the number of registered clients was less than the number of simulated clients. Figure 5-8 showed that the throughput (the number of registered clients per second) was increasing as the number of registering clients increased. Also, there was a relatively low response time variance (0.002) among the groups of the simulated clients. These two observations indicate that YAWS and the prototype are scaling to the number of connections. However, further experiments are required to test other implementations of this thesis' architecture that include other RESTful webservers such as Apache Tomcat[3] and XAMPP[38].

- Which protocol should be used to exchange CRUD messages between the local presenter and the profile server?

  The overhead test: The prototype used the WebSocket protocol and the WebSocket messages to move the CRUD messages between YAWS and the Android tablet. To study the impact of using YAWS's implementation of the WebSocket protocol, this test was applied in two phases:

  1. A load generator was implemented to simulate groups of 5 to 50 concurrent WebSocket clients. Each client sent 10 to 100 WebSocket messages. Each WebSocket message contained 421 bytes of metadata as a CRUD message (Figure 5-12). The load generator and the prototype were hosted on two separate

machines. The prototype measured the amount of time needed to process all the WebSocket messages for each group.

2. The load generator was reconfigured to bypass YAWS and generate groups of 5-50 concurrent Erlang processes. Each Erlang process sent 10-100 CRUD messages to the resources' server directly. The prototype measured the amount of time needed to process all the CRUD messages for each group.

Figure 5-13 showed that the number of concurrent clients have minimal impact on YAWS's processing time because the measured processing time for YAWS did not follow any recognizable pattern. Also, Figure 5-13 showed that YAWS's WebSocket message processing is more efficient than the load generator's CRUD message generation among the cycles from 10 to 30 concurrent clients; therefore a more efficient load generator is required to test the overhead of the prototype.

CHAPTER 6
SUMMARY AND CONTRIBUTION

This thesis introduces a novel approach for synchronizing cloud-hosted NoSQL databases with tablet/smartphone-hosted NoSQL databases using publish/subscribe-based RESTful web services. The growing popularity of smartphone and tablet mobile devices motivated popular social media websites that host user generated multimedia content to enter the mobile application market. The success of these applications encouraged small businesses and communities to invest in similar mobile applications. One of these investors was the community of comedians in Saskatoon. This community was searching for a mobile application that distributes their multimedia content to their fans and other members of the community directly from their mobile devices. Also, they required that these content to be organized according to the content's topics (such as animal jokes and food jokes). The fans should be able to register to their favorite topics and receive notification when the comedians add new content to these topics. This thesis suggests a possible design to the background software system operations and components that will lead to a full implementation of this mobile application in the future.

This thesis's architecture is based on the Model-View-Presenter architecture. This thesis's architecture consists of 5 components, they are: the profile server (represents a middleware between the resources' server and the mobile application), the resources' server (maintains a the message broker, a user profile, and a NoSQL database that contains multimedia file's metadata records for each user), the channel server (maintains a NoSQL database contains the topics' records), the media file's server (maintains the user's shared multimedia files) and the mobile application. The mobile application is subdivided into 3 components: the local model (maintains the locally generated and

downloaded multimedia files and a locally accessible NoSQL database that contains the multimedia files' metadata), the local presenter (distribute the synchronization operations among the local model, the servers and the view, collects the multimedia file's metadata and generates the user's profile) and the view (captures the user's events and collects the user's input data).

When the user starts the mobile application for the first time, the local presenter generates a unique record called the user's profile. The user's profile contains information about the user, the mobile device and the structure of this device's metadata. The resource's server uses the user's profile to search through the metadata, which is based on the content-based publish/subscribe approach. The profile server monitors mobile device's connection status and passes the NoSQL databases' synchronization operations from the mobile device to the resource's server. When a user generates a multimedia file, the file is uploaded to the media file's server. The media file's server generates a new hyperlink for each newly uploaded file and pass it back to the mobile device, which is based on the REST communication style. When a user shares the new hyperlink using one of the topics at the channel server, this server passes notifications to all the message brokers that relates to the users that have been registered to this topic. The message broker either forwards the notification to the user's mobile device or maintains this notification until the user's mobile device reestablishes the connection. This notification style is based on the topic-based publish/subscribe approach. The NoSQL databases' synchronization operations are based on the CRUD data manipulation operations.

To test the architecture, a prototype of the mobile application was implemented using Android's WebView and PhoneGap. Also, a prototype of the profile and the media file's servers were implemented using Erlang-based modules hosted in YAWS. Furthermore, a prototype of the resource's server was implemented entirely in Erlang. The servers used the Erlang-based ETS and DETS tables as the shared NoSQL databases. The mobile application used WebSQL database to simulate a mobile-hosted NoSQL database.

The contribution and the findings of this work are summarized below:

- A novel approach for synchronizing NoSQL databases using a RESTful web service.
- Combining the content-based with the topic-based publish/subscribe notification mechanism can provide user-specific topic-based notifications.
- Demonstrating the benefits of implementing a mobile-hosted NoSQL database.

Although this thesis's architecture achieved relatively high scalability, the system's scalability is still limited to the capacity of the machines used to host the system's servers. Also, the push communication style is not as reliable as the pull communication style because there is no guaranty that the "pushed" message has actually arrived to its destination especially when the message is sent through an intermittent connection.

The tests results indicates that this architecture is applicable for background services that requires an indirect interaction between the user and the server because of the 3 second average in respond time and because of the intermitted connectivity of mobile devices. Also, this architecture is not suitable for real-time applications because of the relatively long processing time caused by the increasing number of message brokers and interactions among these message brokers.

CHAPTER 7
FUTURE WORKS

7.1 Scaling Horizontally

This thesis's architecture is designed to include 4 RESTful servers. This architecture's scalability is limited to the hardware capacity of the hosting machine and the system's implementation. If the number of connected mobile devices is within this limit, the system is expected to perform normally. However, the system will start refusing incoming messages if this limit was exceeded. To overcome this issue from the server side, this system needs a load balancing mechanism. This mechanism should decide when and where to route the registration and CRUD messages among multiple resources' server-hosting machines. Using multiple resources' server-hosting machines can increase the maximum number of concurrent clients. Also, this mechanism should route the CRUD messages among multiple media files' server-hosting machines because these files are expected to be large and the system's storage requirement is expected to increase rapidly. To overcome the profile server's scalability issue, the local presenter at client side should maintain a list of multiple profile servers or request the list of the available profile servers from a central server and decide the best server to connect to depending on the server's criteria such as destination and response time.

7.2 User Modeling

The user's profile describes the data structure of the multimedia files' metadata. The message broker uses The user's profile to search through the content of this metadata because the content and the structure of the metadata are expected to vary among different mobile devices. However, the user's profile can also be used to send customized content and on-demand content to specific

users. The variety of delivered content enables service providers to apply content access policies such as age-related, country-specific, security and privacy policies. Also, the user's profile can be used to collect statistical data about the user's behavior that could be used by User Modeling-related systems such as the recommender systems. These systems can enhance the user's multimedia experiences.

7.3 Data Replication

The user's data are hosted on 3 servers. Any external or internal problems can cause the system to fail. Also, any data loss or database failure is currently irreversible. To overcome this problem, data replication mechanisms can be included in the architecture to redirect the user's messages to a replica of the original database until the original database is back online. The system's servers are responsible for maintaining these replicas and the client side should not be concerned about these resource-intensive operations.

LIST OF REFERENCES

[1] Aijaz, F. et al. 2009. Enabling High Performance Mobile Web Services Provisioning. *2009 IEEE 70th Vehicular Technology Conference Fall* (Sep. 2009), 1–6.

[2] Android 4.2 APIs: *http://developer.android.com/about/versions/android-4.2.html#MultipleUsers*. Accessed: 2013-07-22.

[3] Apache Tomcat: 2013. *http://tomcat.apache.org/*. Accessed: 2013-08-20.

[4] Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. 13, 6 (Jun. 1970), 377–387.

[5] Dean, J. and Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*. 51, 1 (Jan. 2008), 107–113.

[6] Erlang programming language: *http://www.erlang.org/*. Accessed: 2013-04-29.

[7] Eugster, P. 2007. Type-Based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems*. 29, 1 (Jan. 2007), Article 6.

[8] Eugster, P.T. et al. 2003. The many faces of publish/subscribe. *ACM Computing Surveys*. 35, 2 (Jun. 2003), 114–131.

[9] Fielding, R.T. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. University of California.

[10] Gottschalk, K. et al. 2002. Introduction to Web services architecture. *IBM Systems Journal*. 41, 2 (2002), 170–177.

[11] Hypertext Transfer Protocol -- HTTP/1.1: 1999. *http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4*. Accessed: 2013-02-04.

[12] Indexed Database API: *http://www.w3.org/TR/IndexedDB/#introduction*. Accessed: 2012-10-12.

[13] Indrawan-Santiago, M. 2012. Database Research: Are We at a Crossroad? Reflection on NoSQL. *2012 15th International Conference on Network-Based Information Systems* (Melbourne, Sep. 2012), 45–51.

[14] Ireland, C. et al. 2009. A Classification of Object-Relational Impedance Mismatch. *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications* (2009), 36–43.

[15] Jayathilake, D. et al. 2012. A Study Into the Capabilities of NoSQL Databases in Handling a Highly Heterogeneous Tree. *2012 IEEE 6th International Conference on Information and Automation for Sustainability(ICIAfS)* (Sep. 2012), 106–111.

[16] jQuery Mobile: 2013. *http://jquerymobile.com/*. Accessed: 2013-04-30.

[17] Krasner, G. and Pope, S. 1988. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object Oriented Programming*. 1, 3 (1988), 26–49.

[18] Lakshman, A. and Malik, P. 2010. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*. 44, 2 (Apr. 2010), 35–40.

[19] Lomotey, R.K. et al. 2012. SOPHRA: A Mobile Web Services Hosting Infrastructure in mHealth. *2012 IEEE First International Conference on Mobile Services* (Saskatoon, 2012), 88–95.

[20] Lomotey, R.K. and Deters, R. 2013. Reliable Consumption of Web Services in a Mobile-Cloud Ecosystem Using REST. *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering (SOSE)* (Mar. 2013), 13–24.

[21] Martins, J. and Duarte, S. 2010. Routing Algorithms for Content-Based Publish/Subscribe Systems. *IEEE Communications Surveys & Tutorials*. 12, 1 (2010), 39–58.

[22] Mobile device market to reach 2.6 billion units by 2016: 2013. *http://www.canalys.com/static/press_release/2013/canalys-press-release-220213-mobile-device-market-reach-26-billion-units-2016_0.pdf*. Accessed: 2013-06-25.

[23] Model-View-Presenter Pattern: *http://msdn.microsoft.com/en-us/library/ff647543*. Accessed: 2013-02-27.

[24] OrientDB Graph-Document NoSQL dbms: *http://www.orientdb.org/*. Accessed: 2013-02-06.

[25] Overdick, H. 2007. The Resource-Oriented Architecture. *Proceedings of the 2007 IEEE Congress on Services - SERVICES 2007* (Salt Lake City, Utah, USA, 2007), 340–347.

[26] Pautasso, C. et al. 2008. Restful web services vs. "big"' web services: making the right architectural decision. *Proceedings of the 17th international conference on World Wide Web - WWW 08* (Beijing, P.R. China, 2008), 805–814.

[27] Phonegap FAQ: 2012. *http://phonegap.com/about/faq*. Accessed: 2012-09-04.

[28] Potel, M. 1996. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*.

[29] Reenskaug, T. 1979. *THING-MODEL-VIEW-EDITOR an Example from a planningsystem.*

[30] Richardson Maturity Model: steps toward the glory of REST: 2010. *http://martinfowler.com/articles/richardsonMaturityModel.html*. Accessed: 2011-03-30.

[31] Rodriguez, M.A. and Neubauer, P. 2010. The Graph Traversal Pattern. *CoRR*. abs/1004.1, (Apr. 2010).

[32] Sakr, S. et al. 2011. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials*. 13, 3 (2011), 311–336.

[33] Selinger, P.G. et al. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the 1979 ACM SIGMOD international conference on Management of data - SIGMOD '79* (New York, New York, USA, 1979), 23–34.

[34] Singhera, Z.U. 2008. A Workload Model for Topic-based Publish/Subscribe Systems. *Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA Companion '08* (New York, New York, USA, 2008), 703–711.

[35] The WebSocket protocol: *http://tools.ietf.org/html/rfc6455*. Accessed: 2013-07-01.

[36] Von der Weth, C. and Datta, A. 2012. Multiterm Keyword Search in NoSQL Systems. *IEEE Internet Computing*. 16, 1 (Jan. 2012), 34–42.

[37] Wickramarachchi, C. et al. 2012. Andes: A Highly Scalable Persistent Messaging System. *2012 IEEE 19th International Conference on Web Services* (Jun. 2012), 504–511.

[38] XAMPP: 2013. *http://www.apachefriends.org/en/xampp.html*. Accessed: 2013-08-20.

[39] Yaws - Yet Another Web Server: *http://yaws.hyber.org/yaws.pdf*. Accessed: 2013-06-03.