POLICY-BASED MIDDLEWARE FOR MOBILE CLOUD COMPUTING

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

Ashik Kazi

© Copyright ashik kazi, August, 2013. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfillment 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

Mobile devices are the dominant interface for interacting with online services as well as an efficient platform for cloud data consumption. Cloud computing allows the delivery of applications/functionalities as services over the internet and provides the software/hardware infrastructure to host these services in a scalable manner [1]. In mobile cloud computing, the apps running on the mobile device use cloud hosted services to overcome resource constraints of the host device. This approach allows mobile devices to outsource the resource-consuming tasks. Furthermore, as the number of devices owned by a single user increases, there is the growing demand for cross-platform application deployment to ensure a consistent user experience. However, the mobile devices communicate through unstable wireless networks, to access the data and services hosted in the cloud. The major challenges that mobile clients face when accessing services hosted in the cloud, are network latency and synchronization of data.

To address the above mentioned challenges, this research proposed an architecture which introduced a policy-based middleware that supports user to access cloud hosted digital assets and services via an application across multiple mobile devices in a seamless manner. The major contribution of this thesis is identifying different information, used to configure the behavior of the middleware towards reliable and consistent communication among mobile clients and the cloud hosted services. Finally, the advantages of the using policy-based middleware architecture are illustrated by experiments conducted on a proof-of-concept prototype.

ACKNOWLEDGMENTS

First of all I would like to express my sincere gratitude to my supervisor, Dr. Ralph Deters for his advice and assistance throughout my entire duration of study. Besides my supervisor, I would like to thank my advisory committee: Dr. Julita Vassileva, Dr. Chanchal Roy and Dr. A. Wahid Khan for their valuable advice and suggestions. Furthermore, I would like to appreciate the assistance of Ms. Jan Thompson and Ms. Gwen Lancaster, at the Department of Computer Science, University of Saskatchewan, for their support and generosity.

I am glad to be a part of the Multi-Agent Distributed Mobile and Ubiquitous Computing (MADMUC) Lab. Thanks to all students from the lab for their supports.

Finally, I would like extend special thanks to my parents (Kazi Lutfa Mouhla and Nasrin Jahan), entire family and friends for providing me confidence, encouragement and emotional support throughout this period.

TABLE OF CONTENTS

page

PERMISSION TO USE………………………………………………………………………i

ABSTRACT ii

ACKNOWLEDGEMENTS...iii

LIST OF TABLES vi

LIST OF FIGURES viii

LIST OF ABBREVIATIONS…... iv

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: PROBLEM DEFINITION...4

2.1 The Limitation of Mobile Clients to Consuming WS in the Cloud 4
2.2 The Idea of Policy-Based Middleware in Mobile Cloud Computing 6
2.3 Research Goals and Sub-Goals 7

CHAPTER 3: LITERATURE REVIEW 8

3.1 Web Services 8
3.2 Service Oriented Architecture (SOA) 9
3.3 Representational State Transfer (REST)... 11
3.4 Web Services for Mobile Clients 12
3.5 Cloud Computing and SOA 13
3.6 Approaches for Mobile Cloud Computing 15
3.6.1 Virtual Machine Migration 15
3.6.2 Application Partitioning...18
3.7 Local Cloud... 20
3.8 Model View Controller (MVC) architecture 21
3.8.1 Model-View-Presenter (MVP)..22
3.9 Middleware Proxies 24
3.9.1 Middleware for Mobile Web Services 24
3.9.2 Policy-Based Middleware for Web Service Consumption 25
3.10 Stub-Skeleton Pattern.. 27
3.11 Web Service Caching and Prefetching 28
3.12 Event Handling and State Change Propagation 29
3.13 Summary 30
3.14 The Open Issues 32

CHAPTER 4: ARCHITECTURE...33

4.1 Example Scenario 33

4.2 Overview ... 35
4.3 Middleware Architecture ... 37
4.4 The Information for the Policy-File ... 44
4.5 Adoption of MVP Design Pattern in the Policy-based Middleware Architecture .. 47
4.6 Mobile Client ... 49
4.7 The delivery of Cloud Services through Policy-based Middleware ... 53
4.8 Summary ... 57

CHAPTER 5: IMPLEMENTATION ... 58

5.1 Mobile Client Implementation ... 58
5.1.1 Client Cache ... 60
5.2 Middleware Implementation ... 62
5.2.1 Router ... 62
5.2.2 Master Presenter ... 63
5.2.3 Configuration Component ... 64
5.2.4 Middleware Stub ... 65
5.2.5 Middleware Cache ... 66
5.2.6 Monitor ... 67
5.3 Summary ... 68

CHAPTER 6: EXPERIMENTS ... 69

6.1 Experiment Goals ... 69
6.2 Experiment Setup ... 70
6.3 List of Experiments ... 72
6.3.1 Latency Experiments ... 72
6.3.2 Experiment for support multiple devices model synchronization. ... 87
6.3.3 Overhead Experiments ... 90
6.3.4 Experiments to evaluate middleware performance in the local cloud. ... 99
6.4 Summary ... 102

CHAPTER 7: SUMMARY AND CONTRIBUTION ... 105

CHAPTER 8: LIMITATIONS AND FUTURE WORKS ... 108

8.1 Middlewate Support for Concurrent Clients and Load Distribution ... 108
8.2 Explore Implication in Different Domain and More Info in the Policy-File ... 108
8.3 Policies for Data Security on the Cloud Platform ... 109
8.4 Embedding the Middleware with Different Existing Web-framework ... 110

LIST OF REFERENCES ... 111

LIST OF TABLES

Table page

Table 3.1 Represents REST verbs and corresponding CRUD operations 11

Table 3.2 Summarize the capabilities of RESTful WS..12

Table 3.3 Summary of problems and solutions from the reviewed literature...................31

Table 6.1 Experiment Goals 69

Table 6.2 Results comparing avergae request-response time of Test 1 and Test2 75

Table 6.3 Request-response time for consuming SOAP-based WS directly 77

Table 6.4 Request-response time consuming SOAP-based WS through the middleware.78

Table 6.5 Results comparing request-response time for SOAP services ………………...79

Table 6.6 Average time for state synchronization without the middleware 81

Table 6.7 Average time for state synchronization with the middleware 82

Table 6.8 Average time and amount of data receiver by the client for state synchronization without the state-transition information………………………………………...83

Table 6.9 Average time and amount of data receiver by the client for state synchronization with the state-transition information 84

Table 6.10 Results comparing aTeat1 and Test (State Synchronization) 84

Table 6.11 Results comparing Test 3 and Test 4 (STD info in policy-file).......................85

Table 6.12 Average time of synchronization for a model in Google Nexus7 87

Table 6.13 Average time for synchronization a model with in ASUS EEE pad 88

Table 6.14 The average time to synchronization a model (middleware vs. Web server)..88

Table 6.15 Average request-response time using the middleware without policy-file......91

Table 6.16 Average request-response time, using the middleware with the policy-file....92

Table 6.17 Results comparing middleware overhead caching and pre-fetching 93

Table 6.18 Request-response time to obtain resources from the DETS table 94

Table 6.19 Request-response time to obtain resources from the MongoDB 94

Table 6.20 Results comparing Test 1 and Test 2 (overhead MongoDB vs. DETS)..........94

Table 6.21 Request-response time for different percentage of read-write mix 96

Table 6.22 Results comparing request-response time for read and write..........................96

Table 6.23 Overhead of state synchronization by the monitor component 98

Table 6.24 Overhead of state synchronization (State transition information)...................98

Table 6.25 Avergae request-response time to get resources from the local cloud 99

Table 6.26 Average request-response time to get resources from the distant cloud........100

Table 6.27 Average latency to delivery each resource local cloud vs. distant cloud 101

LIST OF FIGURES

Figure page

Figure 1.1 Policy-based middleware connecting consumer devices to cloud services........2

Figure 2.1 Consuming web services by a mobile client from the cloud.............................4

Figure 2.2 Latency introduced by number of hops server 5

Figure 3.1 Service Oriented Architecture 10

Figure 3.2 Virtual Machine Migration Approach 15

Figure 3.3 Application Partitioning Approach...18

Figure 3.4 Cloudlet 20

Figure 3.5 Model-View-Controller Architecture 21

Figure 3.6 Stub-Skeleton Pattern 27

Figure 4.1 A mobile client consuming cloud services 33

Figure 4.2 Policy-based middleware for consuming services from the cloud……….........35

Figure 4.3 Architecture for policy-based middleware 37

Figure 4.4 Keep active certain part of the model required by the client's.........................38

Figure 4.5 Configuration behavior of different middleware components 39

Figure 4.6 Active smart stub interaction with resources in database................................40

Figure 4.7 Sate synchronization by the monitor 41

Figure 4.8 Middleware supports N-devices of a single user...43

Figure 4.9 Service delivery in the disconnected environment..43

Figure 4.10 An example of the state-transition-diagram 44

Figure 4.11 JSON representation of State Transition Diagram 45

Figure 4.12 The adoption of MVP in distributed environment by the middleware...........48

Figure 4.13 Mobile Client Architecture 49

Figure 4.14 Client stubs interaction with resources in cache..51

Figure 4.15 Interaction between a mobile client and the middleware 52

Figure 4.16 Different state of an application interacting with different WS 53

Figure 4.17 An example of state transition info in the policy-files 54

Figure 4.18 An example of resource related info in the policy-files 55

Figure 4.19 Delivery of requested resource by the middleware 56

Figure 4.20 Pushing updates to the clients based on the state information 57

Figure 5.1 Adopted technologies and protocol towards developing hybrid app…………59

Figure 5.2 Code snippet to create table and store data in local databases 60

Figure 5.3 Screenshots of the Mobi-Crop app on an iPad 61

Figure 5.4 Code snippet for extracting client request from JSON....................................63

Figure 5.5 Code snippet of application for the mater presenter 63

Figure 5.6 Policy-file to the configuration component...64

Figure 5.7 Code snippet for processing policy-file by the configuration component 64

Figure 5.8 Code snippet for GET and POST by the middleware stub..............................65

Figure 5.9 Middleware response for GET 65

Figure 5.10 Code snippet for Read and Write operations by cache controller…………...66

Figure 5.11 Code snippet for monitor module...67

Figure 6.1 The Experimental Setup 70

Figure 6.2 Set-up for requesting resources directly from the WS. 72

Figure 6.3 Graph of the latency of resource delivery from the RESTful WS 73

Figure 6.4 Set-up for client requesting resources from the device cache 73

Figure 6.5 Graph of the latency of resource delivery from the client cache......................74

Figure 6.6 The Graph of the latency of resource delivery with and without the
Policy-based middleware 75

Figure 6.7 Request-response SOAP message 76

Figure 6.8 Set-up for consuming resources from the SOAP Web server directly............77

Figure 6.9 Set-up for consuming resources from SOAP WS through middleware...........78

Figure 6.10 Client request message and response message from the middleware 78

Figure 6.11 HTTP HEAD request and response from the server 80

Figure 6.12 HTTP GET request and response from the server...80

Figure 6.13 Set-up for update synchronization by the client pulling................................81

Figure 6.14 Set-up for update synchronization by the middleware push 82

Figure 6.15 Application with 5 screens, with each state having 100 resources updates ...82

Figure 6.16 Set-up for update synchronization with the state-transition information 83

Figure 6.17 Graph of latency of state synchronization client pull vs. middleware push...85

Figure 6.18 The latency of state synchronization for different refresh rates 86

Figure 6.19 Set-up for model synchronization with multiple devices of a single user......87

Figure 6.20 Graph of latency for the model synchronization...89

Figure 6.21 Set-up for client requests through the middleware without the policy-file....90

Figure 6.22 Set-up for client requests through the middleware with the policy-file.........91

Figure 6.23 Graph of the overhead of caching and pre-fetching information 92

Figure 6.24 Graph of performance of the read operation MongoDB vs. DETS table.......94

Figure 6.25 Overhead graph of Erlang interfacing tools for MongoDB............................95

Figure 6.26 Inconsistent behavior of Mongrel tool 95

Figure 6.27 Graph of performance of the middleware for percentage of read-write mix..97

Figure 6.28 Set-up to measure request-response time from the local cloud......................99

Figure 6.29 Set-up to measure request-response time for the distant cloud....................100

Figure 6.30 Measuring no of hops between mobile clients and middleware...................100

Figure 6.31 The latency of resource delivery local cloud vs. distant cloud....................101

LIST OF ABBREVIATIONS

EC2 .... Elastic Compute Cloud

GAE .... Google App Engine

HTTP .... The Hypertext Transfer Protocol

IaaS... Infrastructure as a Service

ISR .... Internet Suspend Resume

JS.. JavaScript

JSON... JavaScript Object Notation

MVC... Model View Controller

MVP... Model View Presenter

PaaS .... Platform as a Service

REST .... Representational State Transfer

SO .... Service Orientation

SOA .... Service Oriented Architecture

SOAP .... Simple Object Access Protocol

SaaS .... Software as a Service

UDDI .... Universal Description Discovery and Integration

URI .... Uniform Resource Identifier

VM .... Virtual Machine

WS .... Web Service

WSDL .... Web Service Description Language

XML .... Extensible Markup Language

XMPP.. Extensible Messaging and Presence Protocol

CHAPTER 1
INTRODUCTION

The improvements in mobile network infrastructure make mobile devices a popular client of web services (WS), which is a unified interface towards delivery of applications over the internet. Most of the popular mobile platforms like iOS; BlackBerry and Android include applications that consume WS from the popular websites, such as Google and Facebook [60]. Baccue [4] refers to *Mobile Cloud Computing (MCC)* as a paradigm to assist mobile devices with cloud-hosted software components, to offload resource consuming tasks from the mobile devices. The number of mobile cloud computing subscribers worldwide was 42.8 million in 2008, and this statistic is expected to increase to over 998 million by 2014 [7]. IBM [10] predicts that mobile cloud computing will reach a market value of US$9.5 billion by 2014.

Though there are improvements in terms of processing capabilities and memory for mobile devices, they still face multiple challenges to take full advantage of cloud hosted web services. Latency is a significant problem that mobile clients suffers when consuming resources from the Web. There are some other issues such as the loss of connectivity in the wireless environment and the resource state synchronization on the multiple devices of a single user with the backend service. As the number of mobile devices owned by a single user increases, the importance of cross–platform application development increases [28]. Instead of device-specific user experience, consistent user experience across multiple devices has become a key concern.

The main contribution of this research is providing an architecture for a policy-based middleware to address the above-mentioned challenges. The policy-based middleware supports state based services delivery and synchronization for an application across multiple mobile devices for a single user in a seamless manner. The research focuses on identifying different information to configure the behavior of the middleware towards reliable and consistent communication among

mobile clients and the cloud hosted services. This architecture adopts the CloneCloud [6] technique proposed in service-centric mobile computing. Instead of pushing or offloading by the mobile client, this architecture brings services closer to the mobile client. The middleware can be further aided by deploying on a local cloud [51] server in proximity to the mobile device. The architecture of the middleware combines the Model-View-Presenter (MVP) framework to ensure decoupling of the components of the system. The dual caching model by Liu et al. [38] adopted by the architecture reduces the latency of accessing cloud hosted services. The RESTful interface towards consuming resources from the cloud server is adopted by the middleware as REST is a lightweight protocol and more suitable for mobile clients [24]. Using stub with local cache [28] reduces the communication costs even further and provides offline availability of the application for intermittent connectivity. The model—collection of user–subscribed resources in the middleware—supports to access a single application with multiple devices with a very low latency. The middleware-push technology and content adaptation technology enable the middleware to shrink the client side overhead of processing unnecessary information from the server's responses to client requests.

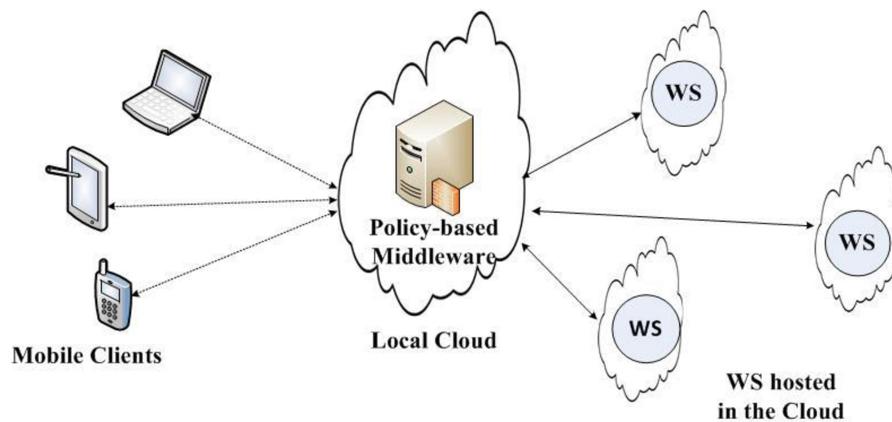

Figure 1.1: Policy-based middleware connecting consumer devices to cloud services

This thesis is organized as follows: Chapter 2 discusses the challenges current mobile clients have when consuming cloud hosted services. Chapter 3 presents an overview of the relevant literature in the areas of mobile cloud computing and web services. Chapter 4 describes the conceptual design and architecture of the proposed framework. A detail about the prototype implementation of the architecture is presented in Chapter 5. The experiments and evaluation are discussed in Chapter 6. Chapter 7 summarizes the research contributions, and Chapter 8 concludes with future work.

CHAPTER 2
PROBLEM DEFINITION

Cloud computing enables individuals and organizations to purchase different services such as virtualized hardware (IaaS), software platforms (PaaS) or applications (SaaS) in a pay-as-you-go manner, comparable to the metered purchase of public utilities (e.g., electricity, gas or water) [1]. This concept of “utility computing model” was envisioned by McCarthy in 1961 [24]. Recently, the popularity of mobile devices has led to an extension of cloud computing to the mobile platform, an area that is described by some researchers as mobile cloud computing [22]. Mobile cloud computing can reduce the workload on the resource poor mobile devices by providing additional resources to access web based services.

2.1 The Challenges of Cloud-hosted Web Services Consumption on Mobile Clients

In comparison to their desktop counterparts, mobile devices have limited connectivity to the network since the available means of connectivity over the wireless medium (e.g., Bluetooth, Wi-Fi) can experience intermittent disconnections. These wireless media also have bandwidth constraints and use of cell network data plan is very expensive. The following problems are identified in this research towards consuming cloud hosted services by mobile clients (Figure 2.1):

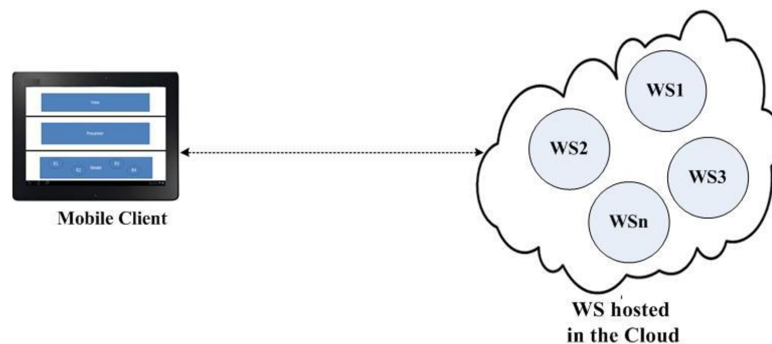

Figure 2.1: Consuming web services by a mobile client from the cloud

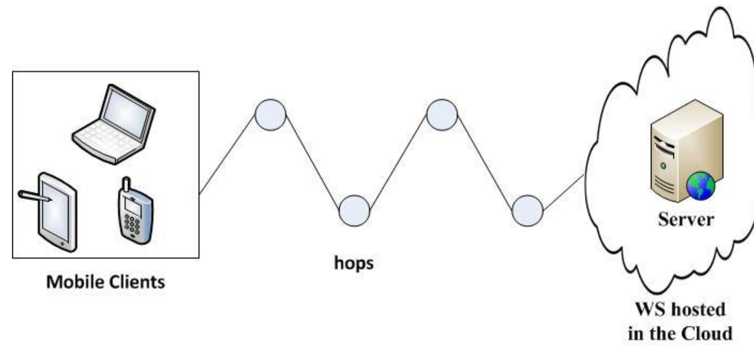

Figure 2.2: Latency introduced by number of hops

Problem 1 (P1): Network-Latency: The number of hops, which is a measurement of the distance between client and server, causes network related latency. Sometimes it is difficult to control traffic in WAN which introduces delay [51] (see Figure 2.2).

Problem 2 (P2): Bandwidth-Limitation: High bandwidth is required to transfer a large amount of data in client-server communication, to reduce the round trip time. The delay can be introduced due to limitation of bandwidth as mobile devices connected via wireless or cell network have restricted bandwidth, which has a negative impact the performance of an application [51].

Problem 3 (P3): State-Synchronization: As the number of mobile devices owned by a single user increases, it is very important to synchronize application state among multiple devices of a user to provide consistent user experience [28].

Problem 4 (P4): Cloud-Latency: The cloud has inherent latency. The overutilization of resources in the cloud requires high interaction within virtual machines, and also adds to the latency [64].

2.2 The Idea of Policy-Based Generic Middleware in Mobile Cloud Computing

This research aims to address the following question:

How can a web compliant generic middleware to support mobile clients consuming web services hosted in the cloud be built?

The following services are offered by the proposed policy-based middleware, to address above mentioned challenges towards consuming cloud hosted services by the mobile clients:

Middleware Service 1 (MS1): Caching: The proposed middleware adopts dual caching model proposed by Liu et al. [38]. This approach reduces the latency of accessing data in the cloud as the data is available in the local cache of the device.

Middleware Service 2 (MS2): Pre-fetching: The middleware pre-fetches resources in advance based on the application state. The middleware keeps updates of the resources in its own cache. Based on an application state on the client device the middleware pushes updates. This approach minimizes the processing of the data on the client end and the bandwidth required to deliver the resources.

Middleware Service 3 (MS3): Content adaptation: The middleware optimizes the service results to reduce the consumption of bandwidth to deliver services. It can convert the format of the service results from XML to JSON. It also removes unnecessary data from the original service response; the optimized service response, with unnecessary data removed, is more readily processed by mobile devices than the unprocessed response.

Middleware Service 4 (MS4): Support multiple devices of a single user: The middleware provides user-centric cloud computing experience, which helps the users to access their digital assets and services via an app across the multiple devices in a seamless manner.

Middleware Service 5 (MS5): Quick recovery from the intermittent loss of connectivity: The middleware caches all the subscribed resources for a client, which ensures quick recovery from the intermittent loss of connectivity.

2.3 Research Goals and Sub-Goals

Goal (G): The goal of this research is to define a generic standard or format which will formalize the behavior of the middleware towards efficient consumption of cloud hosted service by mobile devices. The research focuses on identifying different information for the policy-file of the middleware towards reliable and consistent communication among mobile clients and the cloud hosted services. To accomplish the goal, the following sub-goals must be addressed:

Goal 1 (G1): To identify the impact of different information in the policy-files of the middleware towards reducing the latency of resource delivery from cloud hosted services with minimal overhead.

Goal 2 (G2): To enable state synchronization of the resources in the client cache with backend services hosted in the cloud in real time without introducing much overhead.

Goal 3 (G3): To support access of an application from the multiple devices of a single user with minimal latency.

CHAPTER 3
LITERATURE REVIEW

Section 3.1 to 3.4 focuses on web services architecture and designs, Services Oriented Architecture (SOA) and Representational State Transfer (REST) web services (WS) for mobile clients respectively. Section 3.5 discusses how cloud and SOA together can deploy dynamically scaled, cost effective services. Section 3.6 reviews different approaches taken towards enabling mobile cloud computing. Section 3.7 reviews literature related to the local cloud, which can reduce the network and the bandwidth related latency of accessing services. Section 3.8 reviews the Model View Controller (MVC) and Model View Presenter (MVP) frameworks which make architecture transparent by decoupling the model, the view and the controller. Section 3.9 studies different middleware, proposed to support clients to consume the web services. Section 3.10 discusses the Stub-Skeleton pattern. Caching and prefetching are studied in section 3.11. Section 3.12 reviews event handling and state change propagation.

3.1 Web Services

The web service is defined as a software system which has the capability to support different machines interacting over the network [63]. The most common styles of web services are Service Oriented Architecture (SOA) and Representational state transfer (REST).

For web services, there are three basic elements: Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) [38]. SOAP is a simple XML based protocol (sort of a virtual envelope) which uses HTTP and XML as the mechanism for information exchange. The functionality offered by WS is represented by WSDL, which defines the rules for client (service consumer) interaction with the services. UDDI is used as a means of discovering WS. It has a registry

service, which is responsible for managing information about service providers and service implementations.

3.2 Service Oriented Architecture (SOA)

Liu et al. [38] defined services as autonomous technical functionality which has the capability of retrieving and processing information. Luthria et al. [39] defined a service as a business function implemented in software, which includes the formal documentation. The services enable consumers to access information and use it without knowing implementation specific details. The current organizational infrastructure is complex, which requires distribution of the components. The diversity of platforms, protocols and development environments, give rise to the concept of services and functional entities with the help of internet. This paradigm is called service oriented computing. Don Box specified the four most essential SOA principles [38]:

- **Boundaries are explicit:** In SOA services communicate through explicit message-passing over well-defined boundaries. This helps to reduce the cost of communication and provides abstraction to establish a secure environment.

- **Services are autonomous:** Services should be deployed independently. This eases the addition of new services without prior notification to existing components, and supports fault tolerant services.

- **Services share schema and contracts**: Communication among the services is via XML schema-based messages which provide interoperability without taking into consideration programming languages and platforms. The schema defines the content of the messages while the services contract provides the behavior of the service.

- **Service compatibility is determined based on policy:** The semantic compatibility of a service is specified by the policy which provides the behavior and expectations of a service.

Gartner [18] defines SOA as a software architecture where work is based on interaction of the server-side and the consumers. Independent business components in SOA provide accessibility to resources using a remote programming interface in a modular and distributed fashion. There are three components in SOA: service providers, service requesters (clients) and service discovery agencies (Figure 3.1).

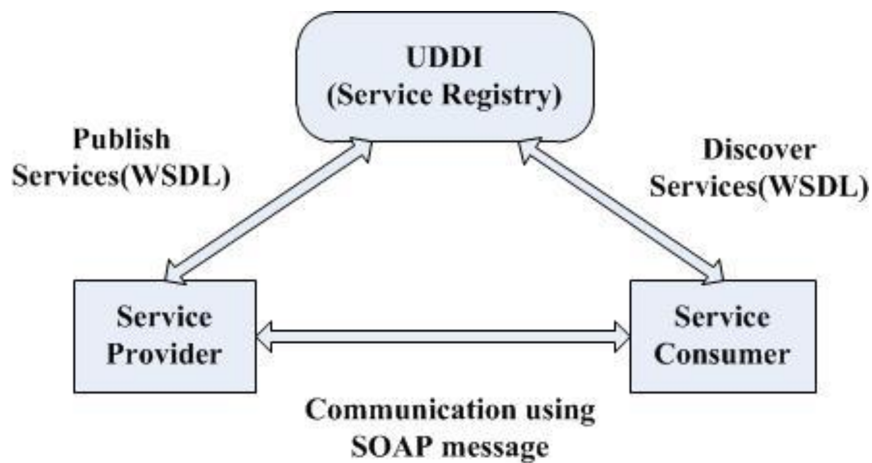

Figure 3.1: Service-Oriented Architecture

3.3 Representational State Transfer (REST)

Fielding [15], in his doctoral dissertation (2002), introduced RESTful web services. REST is a lightweight architecture style or concept for designing networked applications, which has the capability to substitute mechanism like SOAP. To implementation web services in the RESTful manner, one needs to follow four basic principles [49]:

- Using HTTP methods clearly
- Every resource must be exposed through a URI
- Provides a simple and uniform interface
- Interaction must be stateless

REST enables developers to use HTTP methods towards consistent protocol definition. REST is not bound to XML as SOAP services are. Other possible formats through which REST can communicate is JSON (JavaScript Object Notation), provided that the clients and the service providers agree upon it. A resource can be essentially any coherent and meaningful concept that may be addressed by a Uniform Resource Identifier (URI). The representation of a resource is considered to be a document that captures the state of the resource. REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations [49]:

Table 3.1: Represents REST verbs and corresponding CRUD operations

REST Verbs	Purpose
POST	Create a resource on the server
GET	Retrieve a resource
PUT	Update a resource
DELETE	Remove a resource from the server

The RESTful web services are popular because they use well-established web standards like HTTP, URI and MIME. The in-built support for building HTTP client and server application is available in most programming languages, which automatically support REST. The RESTful web services use URIs to represent resources which helps to eliminate the cost and complexity of publishing service description into a registry such as SOA. Table-3.2 summarizes the capabilities of RESTful WS from the literature review:

Table 3.2: Capabilities of RESTful WS

Capabilities of RESTful WS
Provides loose coupling [52][56]
Lightweight and cacheable [49] [60]
Scalable system [12] [52]
Interoperability among different platforms [52]
Memory friendly and requires less preprocessing of data and bandwidth [5] [52] [60]

3.4 Web Services for Mobile Clients

Selonen et al. [52] identified that adopting RESTful WS provides clients to access to all content in a uniform and simple manner and supports interoperability among different platforms. Moreover, this approach provides fine-grained control over the received content, and the statelessness constraint ensures scalability.

Christensen et al. [5] identified that RESTful web services are very suitable for the mobile environment. HTTP based RESTful responses minimize the network volatility, are easy to invoke, are memory friendly, are easier to consume and require less preprocessing of data on the mobile devices, than web services provided on other protocols.

Stirbu [56] et al. used RESTful web services in the mobile environment to provide adaptive and multi-device application sharing. Stirbu also observed that using the RESTful approach makes the system more scalable in term of the number of users, and provides loose coupling among the resources.

Farley et al. [12] defined Mobile Web Services (MWS) as delivering web services to the mobile environment. Mobile web services provide the advantage of increasing inter-operability; basic web services are provided by companies such as Amazon, eBay, Google and Microsoft that can be consumed by mobile clients.

Natchetoi et al. [44] proposed an architecture which can invoke and consume SOAP-services. The approach for mobile clients consuming web services is different from the traditional web-services consumption by the desktop devices. Mobile devices have limitations such as low processing power, low-bandwidth connection and small user interface.

3.5 Cloud Computing and SOA

Christensen et al. [5] argued that combining cloud computing and RESTful web services in the mobile applications has substantial benefits. Cloud computing can help mobile devices to run applications which require large storage capabilities and much processing power. RESTful WSs are quite suitable for providing a connectivity layer with optimized data throughput between cloud and mobile clients in real time through Wi-Fi or Bluetooth.

Feuerlicht et al. [14] defined cloud computing as a key driver towards network computing in which services can be delivered on demand based on virtualization technologies. This enables services to be rented on a subscription basis and removes the concern of a big infrastructure investment and its maintenance. There are three types of cloud computing [14]:

Infrastructure as a Service (IaaS): Provides infrastructure or raw hardware in term storage or computational units on demand e.g. Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2).

Platform as a Service (PaaS): Provides platform allows rapid development and deployment applications e.g. Google App Engine (GAE).

Software as a Service (SaaS): Provides a platform or environment to host services as well as provide an interface to access those services by the clients e.g. Google Apps services via Google docs. To be effective in enterprise computing environments, the scope of SOA needs to be extended to comprise different types of service models, including SaaS (Software as a Service), IaaS (Infrastructure as a Service), Platform as a Service (PaaS) and all other services that represent the web 2.0 environment [14].

Raines [48] defined cloud computing as an effective means for outsourcing IT services, ranging from hardware through application to the cloud providers. The infrastructure can be dynamically scaled and abstracts away the details from the clients. Raines also claimed that SOA and cloud computing are complementary to each other, because both of them have the same focus, namely to reduce the cost and standardization of the services. The service oriented approach for developing mobile applications is considered very suitable to overcome limitations of mobile devices. The computationally intensive services get executed in the resource rich service provider's side. Only lightweight interaction is required from the client side.

According to Tang et al. [59], with the advancement of cloud technology, the server side of a mobile application can be deployed in the cloud. Their proposed model promises easy maintenance of computational services as well as reduced total cost of ownership.

3.6 Approaches for Mobile Cloud Computing

Mobile cloud computing is an approach to increase the capabilities of mobile devices, by offloading computationally intensive operations to the resource rich cloud [32]. The mobile cloud is becoming popular due to cloud services that support remote data access, storage and applications, e.g. iCloud, and DropBox [28]. Different methodologies are proposed for enabling cloud computing in mobile environments. The majority of the research focused on offloading an application by the mobile client to the resource rich cloud. These approaches are mainly based on:

- Application Partitioning
- Virtual Machine Migration

3.6.1 Virtual Machine Migration

In a virtual machine migration approach, a VM is created which encapsulates resource intensive operations of an application in terms of processor, disk and memory status. The VM is transferred to the cloud environment, and the application gets executed from the same state as in the client end [51]. The virtual machine migration approach provides more RAM and CPU advantage as a cloud environment is rich in computational and storage capabilities.

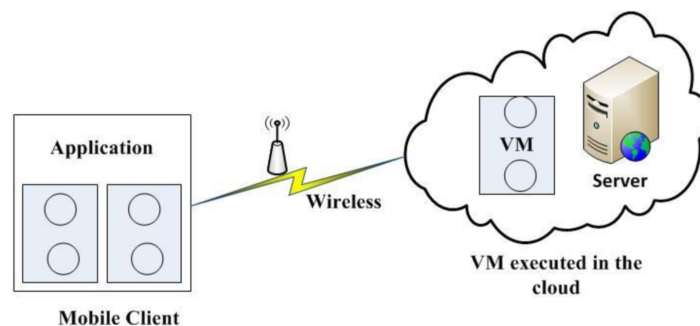

Figure 3.2: Virtual Machine Migration Approach

Satyanarayanan et al. [51] identified different technical obstacles which mobile devices face when consuming cloud hosted services due to network latency and bandwidth-limitations. The proposed approach is called Cloudlets, which reduces latency in the mobile cloud computing by utilizing nearby computing resources. The adoption of the dynamic virtual machine synthesis approach supports resuming execution of an application from a prior state from a previous environment. Cloudlets have advantages such as that performance is fully dependent on local resources and WAN delays or failures have reduced effect on the system, compared to non-cloudlet cloud-based environments.

Kozuch et al. [33] proposed the idea of the Internet Suspend/Resume (ISR) system, which provides a user consistent computing environment without carrying hardware. This approach allows users the flexibility to suspend work on one machine and resume work on another machine. The VM monitors proposed in the architecture encapsulate the entire state of the user's current computing in a parcel which can be kept on the internet and can be resumed by the user at his current place with flexible computing devices. The major limitation of the ISR system [54] specified is the delays experienced by the user due to suspend and resume operations.

The Horatio framework by Smaldone et al. [54] is an extension of ISR system, which treats the smart phones or mobile devices as trusted personal assistants with a self-cleaning portable cache for transferring the VM state. The state of the user's current computing environment is transferred to a personal mobile device from the ISR server. The transfer of VM state to personal mobile device is done when considered to be in a safe state. The device can clear its cache whenever there is an opportunity by transferring the state to the ISR server without further user interaction dependent on availability of networks.

Zhao et al. [68] proposed a framework, which combines cloud computing and virtualization techniques to shift the workload from the mobile devices to the cloud computation infrastructure. The framework keeps a copy of the application image for each mobile device on a computing infrastructure in the telecom network with the mirror (copy of the application environment). The proposed approach reduces the workload of resource constrained mobile devices significantly besides providing the capability of virtually expanding the required resources.

The ThinkAir [32] framework makes it simple for developers to migrate their Smartphone applications to the cloud. This approach takes advantage of the concept of smart phone virtualization on the cloud and provides method-level computational offloading. The architecture can enhance the power of mobile cloud computing by parallelizing method execution over multiple Virtual Machine (VM) images in the cloud platform.

CloneCloud [6] uses a technique which combines both virtual machine migration and application partitioning. The framework supports thread level migration which is essential for accessing device level features (e.g., functionality of a camera). A process, which is executing in the mobile device, is automatically partitioned into smaller processes, based on static analysis and dynamic profiling information. This smaller process is executed in a virtual machine located in the cloud. After the execution of the sub-processes, the framework supports integration in a fine-grained manner.

However, a challenge associated with the virtual machine migration is a high bandwidth requirement [32] for performance optimization. Additionally, the VM migration assumes a very tight coupling between mobile devices since both client and server must share the same VM environment.

3.6.2 Application Partitioning

Partitioning of the app is an alternative approach to the VM-centric offloading, in which computational intensive codes are sent over the internet to the cloud platform to be executed or replaced with cloud hosted services (Figure 3.3).

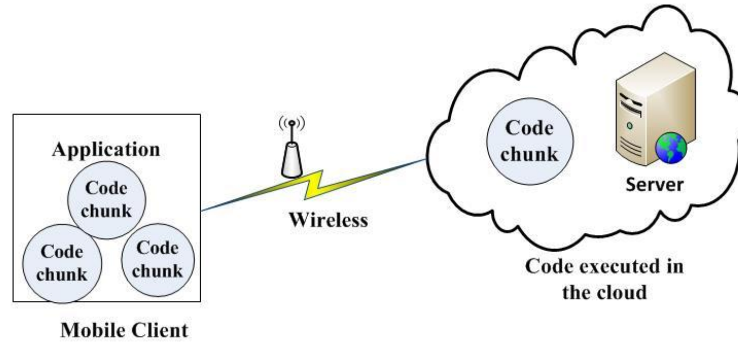

Figure 3.3: Application Partitioning Approach

The MAUI [11] system enables fine-grained energy conscious offloading of mobile code on to the cloud infrastructure. The architecture supports the application partitioning model, which provides a programming environment to the application developers to decide and mark which methods can be offloaded to be executed remotely. The MAUI solver [11] has the responsibility to determine whether it is beneficial in terms of energy consumption and performance, based on the wireless environment such as the bandwidth, the latency and the packet loss, to execute a code remotely. Additionally, the proposed architecture can reduce the latency and overcome the limitation of the smartphone application environment.

The Cuckoo framework [29], proposed by Kemp et al., simplifies the development of smartphone applications to be executed remotely. The programming model offered by Cuckoo splits services part of an application from the activities part. The architecture helps to generate

methods of remote services similar to local services, and can be re-implemented later by the developers according to application requirements. It also allows developers specify the level of parallelism for an application based on speed and performance requirements. Obviously, reliable communication with low latency is required since the local calls are now remote procedure calls (RPC).

The framework introduced by Mei et al. [40] has a proxy and a code repository to support mobile devices to dynamically outsource computation on the clouds. In this framework, mobile devices need to send the class name of code which is going to be executed to the cloud hosted proxy. XRay [46] uses a profiling tool that identifies methods in a smart phone application that can be offloaded to a remote server. Furthermore, it takes the responsibility of making optimized offloading decisions based on resource requirements in the device context.

Zhang et al. [67] provided an approach, which is based on defining RESTful services that can be executed on the mobile device or in the cloud. The architecture uses special SDK and C# codes that provide automatic generation of cloud-ready and device-ready components called weblets. Depending on the load, the mobile device can request the spawning of a weblets in a cloud service which will replace the internal component. The application developer has the responsibility to determine the way of organizing the weblets according to the functionality and runtime behavior of the devices.

While partitioning of apps avoids the tight coupling of VM- centric offloading, the partitioning is still based on a device-centric view which assumes a good interconnectivity and single device usage. Latency is introduced when the chunk of code to be executed in the cloud is very large due to bandwidth limitation. Additionally, the partitioning introduces development effort to specify the chunk of code required to be executed remotely.

3.7 Local Cloud

The concept of using resource rich computing facilities by resource-constrained mobile devices was introduced by Satyanarayanan in his paper "Pervasive Computing: Vision and Challenges" [50]. The concept known as "Cyber forging" supports resource constrained mobile devices to offload some of their heavy work to strong nearby machines.

Balan et al. [3] provided an approach for simplifying cyber forging that is based on a "tactics" file. The file contains information specific to different applications which makes the application adaptive to different environments during runtime. In the paper entitled "Execution plans for Cyber Foraging" [34], Kristensen suggested a framework called Locusts. The Locusts framework breaks down a task into small subtasks and arranges them into a graph. This approach allows a scheduler to distribute subtasks across multiple peers to provide parallelism in execution.

The local computing infrastructure introduced as Cloudlets by Satyanarayanan et al. [51], has the capabilities to support the mobile devices to consume cloud hosted services. The Cloudlet is defined as a trusted resource-rich local computer or a cluster of computers which can support mobile devices to obtain the resource advantages of cloud computing without experiencing delay and jitter (see Figure 3.4).

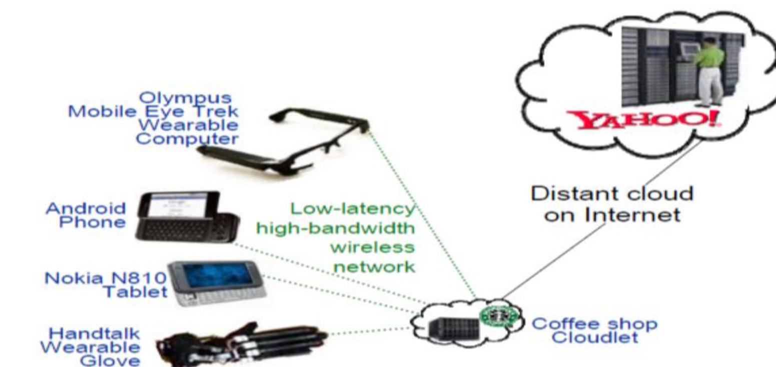

Figure 3.4: Cloudlet by Satyanarayanan et al. [51]

3.8 Model View Controller (MVC) architecture

MVC is an architecture pattern which stands for Model View Controller, invented at Xerox in the 1970s [43]. MVC separates the application logic from the user interface and supports transparency in the architecture. In MVC architecture, an application is decoupled into three parts: the model, the view and the controller (see Figure 3.5).

Model: A model represents the data of an application domain and contains rules to manage access and update those data. The model encapsulates the application states and delivers to view the required data when the view needs it. Additionally, it takes action to change the application state when requested by the view through the controller and on success notifies the view about the new state.

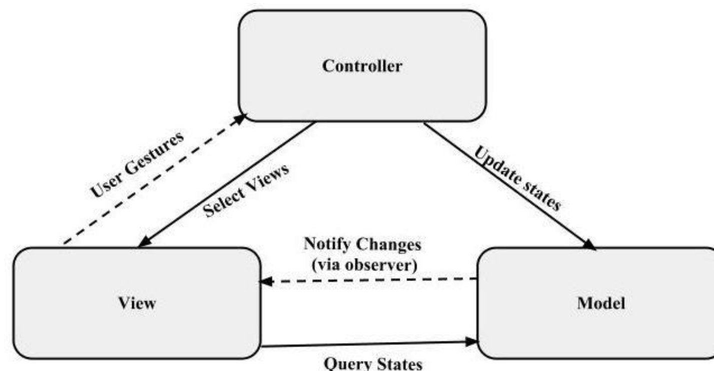

Figure 3.5: Model-View-Controller Architecture [66]

View: The view has the responsibility of rendering UI elements. The view modifies itself as the model is changed and also forwards user request to the controller. It also takes the responsibility to maintain the consistency with respect to changes in the model, which can be achieved through push mechanism as the view registers itself with a model using an observer pattern for change notification [66].

Controller: A controller manages all communication between the view and a model on user actions such as keystrokes or mouse clicks. The controller determines which model components are required to communicate in response to events initiated by a user.

3.8.1 Model-View-Presenter (MVP)

According to Fowler [17], Model–view–presenter (MVP) is derived from the model–view–controller (MVC) software pattern. It improves the separation of concern in MVC, by introducing presentation objects [66]. In MVP, the different components perform the following functions:

- The model is an interface which takes the responsibility of providing data along with an interface to perform change the state of those data.
- In MVP, the view (GUI components) handles the user actions and routes user generated events to the corresponding presenter components. Besides, it also takes the responsibility of presenting a model provided by the presenter component.
- The presenter is listening to an event from the view. It takes action when an event arises from the view, retrieves data from the model, and formats it to be displayed in the view. Additionally, the presenter maps data from a view into the corresponding model.

Difference between MVP and MVC: Both patterns, MVC and MVP, focus on separating responsibility across the different components and promote loose coupling. Several variations make MVP different from the MVC. The relation between the model, the view and the controller is triangular in MVC. This allows the view to query the model component directly, while in MVP, all communication between the model and the view must be handled by the presenter

component. This approach makes the view component more passive and enables better decoupling in the distributed environment.

Stirbu [56] provided an approach which includes MVC compatible interactive applications for remote devices. The framework provides the capability to adapt the user interface among multiple remote devices. In the framework, the model and the controller both reside on the hosting device whereas user agent resides on user devices to render the view according.

Zhang et al. [66] adopted a web-driven MVC pattern and the channel-based published subscribed pattern to enable access to an application from multiple devices of a single user. This approach allows sharing of a model for user applications to access from the multiple devices.

Cortez et al. [9] proposed an E-Learning platform which uses Model-View-Controller design patterns to implement the reusable view and controller built on the distributed cloud platform. In this architecture, the view is decoupled from the model. The view cloud offers Graphical User Interface (GUI) as a service, while the controller cloud offers Task as a service (TaaS).

Balanced MVC Architecture [35] proposed by La et al. enables service-based applications for mobile devices. In their proposed mobile application architecture, they considered mobile devices as a thin client and used the MVC framework to make a layered architecture, which decouples data access from business logic and user interaction. They adopted the MVC framework on both the client and the server side.

Hornsby et al. [21] provided a framework which is capable of supporting interactive applications in the mobile environment with a message based MVC architecture. Their proposed architecture uses the XMPP protocol for exchanging messages among different components. Along with decoupling the components, the approach ensures the reuse of components with minimal efforts.

3.9 Middleware for Web Service Consumption

A middleware is a software component or intermediary which eases communication between client and server in a distributed environment. It can provide different functionalities, for example, providing interoperability for applications, different ways of accessing data according the client's requirements. This section presents different middleware and frameworks proposed by other researchers for mobile cloud computing and policy-based web service consumption.

3.9.1 Middleware for Mobile Web Services

Wang proposed a middleware architecture in his thesis [60], which has a platform independent design for mobile service clients. The architecture supports mobile clients to consume both SOAP and RESTful WSs. The proposed middleware enhances the interaction between mobile clients and web services by content adaption and also provides a personal service mashup platform. Additionally, the middleware is deployed in cloud platforms like Google App Engine and Amazon EC2 to enhance the scalability and reliability of the architecture. One of the major limitations is that the proposed middleware architecture does not provide any information how it will deal with state synchronization on the client devices.

Jamal et al. [24] introduced a cloud-hosted proxy, towards RESTful service consumption by the mobile clients. The middleware framework helps resource constrained mobile devices to efficiently consuming cloud hosted services. The architecture combines a dual caching approach [38] which provides caching resources on the mobile devices while keeping a copy of the resource in the proxy. The adoption of that dual caching model achieves significant performance benefits for mobile clients. However, the work does not specify how the services on the device cache will be synchronized with the backend servers.

The middleware by Kovachev et al. [31] has the capability to take the advantages of nearby computing facilities or local cloud to reduce latency to execute an application. The architecture is based on the XMPP protocol, and provides real time, flexible and scalable software architecture which takes advantage of the cloud by the mobile devices to execute resource intensive applications. The XMPP protocol used in this middleware architecture delivers a pure XML foundation for real time messaging with the services hosted in the cloud.

A generic middleware for Mobile Cloud Computing proposed by Flores et al. [16] has the capability to support hybrid cloud services for mobile devices. The term hybrid cloud service refers to working with various services and APIs offered by different cloud vendors.

3.9.2 Policy-Based Middleware for Web Service Consumption

The policies defined by Zulkernine [69], as a set of rules, which can automate processes to achieve definite goals in a specific context. The middleware framework proposed by Zulkernine [69] provides automated negotiation of service level agreements in the Service Oriented Architecture. The middleware helps to define high level business goals, contexts, preferences, and values as policy. Additionally, the Negotiation Broker middleware supports local execution of policy-based negotiation, which reduces the communication cost in terms of bandwidth, along with overhead of processing messages exchanged between negotiating parties. Furthermore, the collection of information available in the knowledgebase allows both parties to make informed decisions along with providing scope for discussion to update policy specifications.

Joshi et al. proposed an approach [25], which enables service consumers to define different policies such as service's constraints, security policies and compliances towards automatic service discovery, negotiation and consumption of cloud hosted services. A tool developed by Joshi et al., for implementing their framework, allows users to specify their requirements of

service consumption in natural languages. This tool converts the user policies in the natural languages to services specific languages such as RDF, which significantly reduces the time and complexity of consuming services hosted in the cloud.

Takabi et al. [57] identified security and privacy issues as major complications towards quick adoption of cloud computing and presented a Policy Management Framework [58], which provides users with a single control point to define policies to control access to the resources distributed in heterogeneous cloud platforms. Each of the cloud services has its own security mechanism policy, which is very critical for a user to understand and deal with when working with multiple cloud services. The diverse access control policies for the different cloud platforms introduce problems to cloud adoption by different enterprises. The proposed middleware contains a component called policy management service provider which takes the responsibility of providing an interface to define, edit and manage their access policies for distributed resources hosted in different cloud platforms. In addition, the component supports detection of errors and inconsistencies in the access policies.

Lasilla [36] identified the importance of policy-aware service delivery and benefits such as access to device-specific data, and better information representation. The above proposed middleware for policy-based service consumption is not suitable for mobile environments, since mobile devices are resource constrained, have limited processing power, network connectivity and memory.

3.10 Stub-Skeleton Pattern

A stub is a proxy object of a server, residing in client space. A client proxy object residing in the local space of the server is called skeleton. This architecture gives the clients an illusion of talking to the server directly. The stub and the skeleton communicate over a socket connection. Cook et al. [8] and Lee et al. [37] used the Stub-Skeleton pattern to improve the performance of the distributed systems (Figure 3.6).

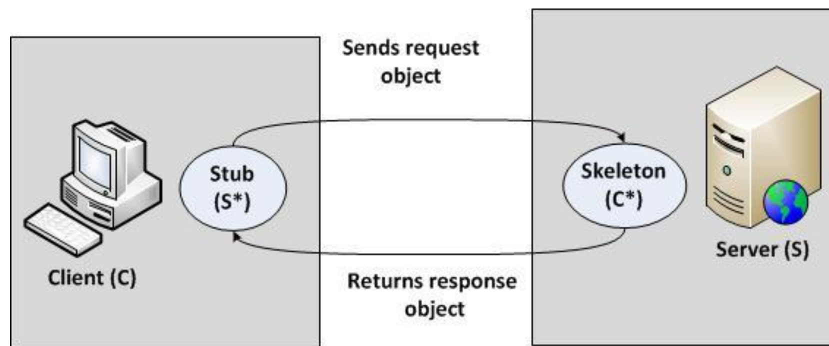

Figure 3.6: Stub-Skeleton Pattern

The responsibilities of a stub and skeleton are as following [45]:

- A stub provides a remote interface of the server to the client. Through stubs clients can invoke a remote method and the call is forwarded to a stub representing the client in the server side.
- The stub has the further responsibilities to carry semantics of communication such as opening socket, marshaling parameters, and data stream, and to dispatch remote objects at the end of the communication.
- The skeleton takes an incoming method, invokes the method on the actual remote object with appropriate parameters and returns the result to its caller which is a stub.

3.11 Web-Service Caching and Pre-fetching

Liu et al. [38] specified caching to be an effective way to reduce access latency, balance network traffic as well as improve availability of resources in the distributed environment. The approach includes caching both client side and proxy side to overcome problems arising from temporary loss of connectivity and fluctuation of bandwidth. The work also used a pre-fetching approach which is based on BPEL files describing user workflow. This approach reduces the latency of consuming web services by mobile devices. Pre-fetching is a technology which supports clients by preloading cacheable resources based on the prediction of user's future requests.

Fernandez et al. [13] proposed a cache based extension to the web services architecture which introduces two level caching architecture (server, client side caching) and three-level caching architecture (server, client and proxy side caching). This is one of the influential contributions towards making current web service architecture efficient. They claimed caching WS improves performance of the system by reducing latency of service consumption.

Kungas et al. [27] proposed a proxy cache solution for SOAP traffic, which supports multiple cache replacement policies and can be configured to tune performance. The proposed solution includes a novel caching scheme which provides minimal cache overhead and high hit ratio. The work also reflected that the Least Recently Used (LRU) replacement policy is better than other policies for cache replacement when traffic is high.

Jamal et al. [24] introduced proxy, adopted the Dual caching approach towards RESTful service consumption by mobile clients. The approach showed client caching can reduce the latency of accessing services when a service will be accessed repetitively.

3.12 Event Handling and State Change Propagation

Stirbu [56] extended the classical MVC pattern, to replicate the view and part of the controller residing in a remote device so as to share multiple-device applications. This architecture adapts resources according to device requirements to ensure a device specific user experience. This includes an event based change-propagation mechanism which ensures synchronization of a model within views of an application residing multiple platform and devices.

The Web Based Application Program Management (wAPM) framework [55] is a combination of two components: an application manager in devices and an application management server. The application management server ensures consistency of an application running multiple devices. The application manager on a device takes the responsibility of communication with the application management server (AMS) and ensures consistency of the state of the application running on user devices. The push based communication is adopted by the architecture to provide loose coupling between a proxy and its devices.

The framework proposed by Zhang et al. [66] is designed as an event driven system, which uses channel-based publish–subscribe. The use of event messaging allows asynchronous communication and provides loose coupling among components in a distributed environment. The framework propagates state changes across multiple devices using the push mechanism to ensure real-time propagation of updates on the connected devices. For disconnected devices, the architecture keeps track of updated resources related to the application and pushes updates automatically when device are re-connected.

For state synchronization, most of the research adopts a push based approach to ensure synchronization of services in real time.

3.13 Summary

The consumption of web services via mobile devices is becoming popular as the mobile technology continues to grow. SOAP based web services are not suitable for mobile devices because they are verbose and involve high performance cost in term of bandwidth, storage and processing [60]. In contrast RESTful web services provide lightweight and flexible interactions, which are very suitable for the mobile clients [5] [12] [60].

Cloud computing is a key driver towards network computing, which can deliver services on demand in a scalable manner [14] [48]. The service oriented approach for developing mobile application is considered very suitable [5] [21]. Middleware can enable mobile clients to consume WS efficiently [24] [60]. The middleware can provide features such as caching [24] [38] [60], prefetching [38] and content adaption [60]. Further, the architecture can use the local cloud [51] to reduce the latency of service consumption. A policy-based middleware can automate services consumption, towards reducing client overhead. Hence, the above policy-based middleware proposals [25] [58] [69] are not suitable for mobile devices as they did not consider the resource constraints that mobile devices have in terms of CPU, memory and network access. The software architectural pattern like MVC/MVP [56] [66] enable better decoupling among components in distributed environments; including a pattern like stub-skeleton [8] [37] increases the performance of a distributed system. One major concern is how the application state will be synchronized among multiple devices of a single user, which was addressed by Stirbu [56] and Zhang et al. [66] by using the event-driven mechanism.

The client centric approach for mobile cloud computing, which is based on offloading or pushing computational intensive operations to resource rich cloud, is an active area of research. However, my research focuses on bringing services closer to the client, which is called “Service centric approach” for mobile cloud computing [24] [60].

Solutions of the problems mentioned in the problem statement (Section 2.1) found in reviewed research are summarized in Table 3.3:

Table 3.3: Summary of problems and solutions from the reviewed literature:

Issues	Findings
Network-Latency & Bandwidth-Limitation	• Consuming RESTful WS is very attractive for a mobile client since it is lightweight in comparison to other protocols such as SOAP [5] [24] [56]. • Web Services caching can reduce the network latency [13] [24] [27] [38]. • Content adaptation reduces bandwidth consumption while consuming cloud services by the mobile clients [60]. • Pre-fetching can reduce the network latency of consuming WS [38]. • Hosting a middleware on the local cloud brings benefits in terms of saving bandwidth and removing network latency [3] [6] [51]. • Using a policy-based middleware for service consumption can reduce client overhead [25] [57] [69].
State-Synchronization	• Event driven mechanism supports real time state synchronization among multiple devices of a single user [56] [66].
Cloud-Latency	• Dual caching model reduces the number of calls to the cloud server [24] [38].
Other requirements to application support in the distributed environment	• Cloud computing provides scalability to hosting services [52] [56] [60]. • MVC/MVP technique provides better decoupling among components in a distributed environment [35] [66]. • Client-stub ensures availability of services in a disconnected environment [28].

3.14 The Open Issues

In this thesis, the following unresolved issues in the literature review have been identified:

- How can different information on the middleware ensure better support for mobile clients towards delivery and synchronization of cloud hosted services?
- How can the state of the resources in the client cache be synchronized with the cloud hosted services without introducing significant overhead to the mobile client?
- How can the model (collection of subscribed services by a user) be synchronized between the multiple devices of a single user with minimal overhead?
- How can the number of requests to the cloud server be reduced to overcome latency?

CHAPTER 4
DESIGN & ARCHITECTURE

In this chapter, a framework is proposed which includes policy-based middleware to minimize the latency of delivery and to synchronize cloud hosted services on user mobile devices. The following sections of this chapter include different approaches and techniques adopted by the proposed architecture to efficiently deliver and synchronize cloud hosted resources on multiple mobile devices of a user. This chapter includes possible solutions of the problems (network-latency, bandwidth-limitation, cloud-latency, and state-synchronization) listed in Chapter-2.

4.1 Example scenario

To understand the functionalities of a policy-based middleware, in this section I introduce an example mobile application which will be used throughout this chapter.

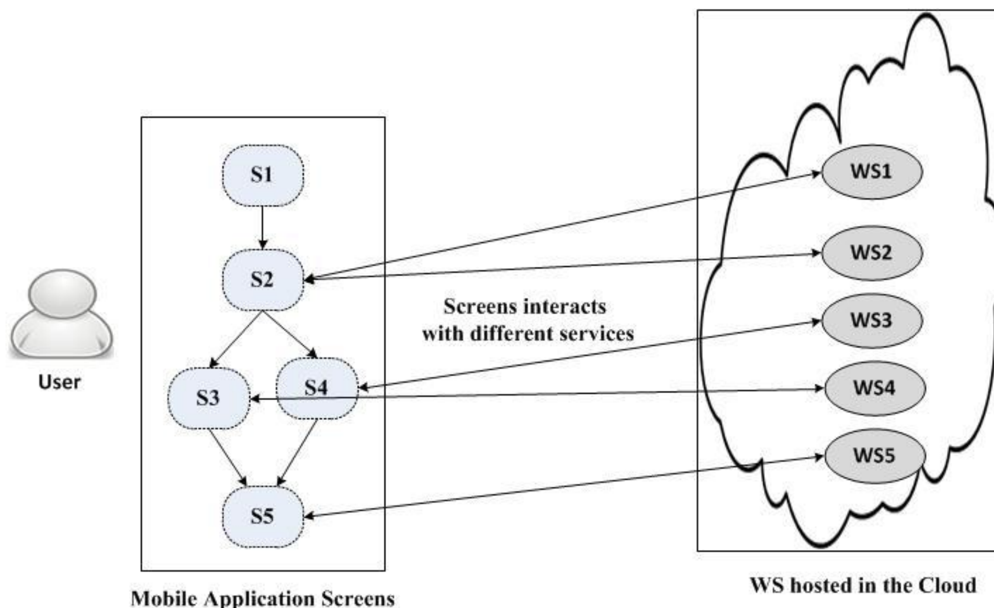

Figure 4.1: A mobile client consuming cloud services

Problem Description: Let us assume one example of a mobile application (see Figure 4.1) which consists of 5 screens, S1 to S5. Each of the screens is a user interface through which the mobile clients interact with the web services. Different screens interact with different web services hosted in the cloud, namely WS1, WS2, WS3, WS4 and WS5. The screen S1 is the starting screen for the application. The user can move to screen S2 in which the user can perform read–write operation to WS1 and WS2. The read operation provides user data from a service while, by issuing a write operation, a user can add new resources in the service. From S2 the user can move to S3 or S4. On screen S3 the user can interact with WS3 which only allows read operations. The screens S4 and S5 interact with WS4 and WS5 respectively. Suppose in S2, a user issues a get request to a service to read some data from the backend server. The consumption of cloud hosted services by mobile clients has challenges such as high latency, loss of connectivity and state synchronization of the resources in the client device. Mobile clients have to face these challenges each time it moves from one screen to another screen.

The proposed middleware takes a different approach towards dealing with the above mentioned scenario. This middleware knows in advance, about different states of the application and resources required by the client; the middleware's knowledge is derived from the policy-file given by an application developer. The policy-based middleware ensures that the resources, with which the user may soon interact, are synchronized in the device cache, in advance. Additionally, the middleware performs content adaptation to reduce overhead and latency of resource delivery and synchronization on the mobile clients. Furthermore, the middleware deals with the short-lived loss of connectivity and can support multiple devices of a single user.

4.2 Overview

This research focuses on a novel middleware architecture, to enable user-centric cloud computing [28] experiences, that supports users' access to their digital assets and services via an app across multiple devices in a seamless manner. The most important contribution of this research is identifying the information which will be used to configure the behavior of the middleware to synchronize only the relevant resources based on the application state. This synchronization of only relevant resources reduces network latency of service delivery and reduces client side overhead of data processing. The architecture consists of three components: the mobile clients, the middleware and the services hosted in the cloud (see Figure 4.2). The architecture adopts the Model View Presenter (MVP) framework to enable loose coupling within its components in the distributed environment. The middleware supports mobile clients to consume both SOAP and RESTful web services hosted in the cloud. Moreover, the architecture delivers the optimized service results to the mobile client. The approach further aids the mobile client to offload the computational workload to the middleware which can be deployed on a local cloud [51] server within proximity to the client.

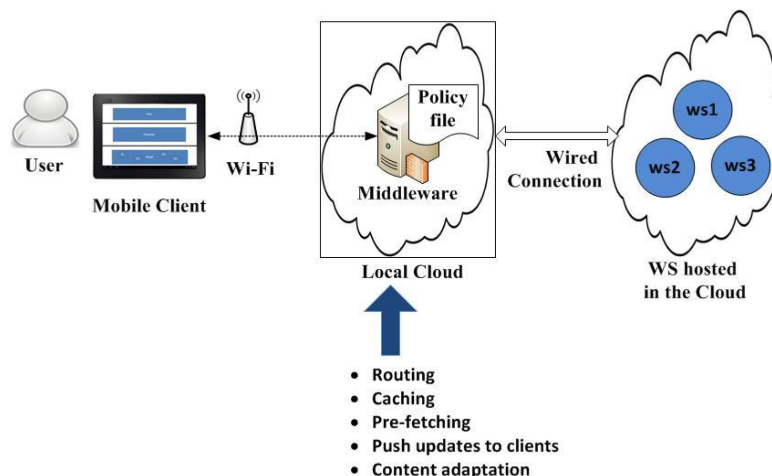

Figure 4.2: Policy-based middleware for consuming services from the cloud

The following section addresses how proposed architecture reduces the severity of different problems (e.g., Network-latency (P1), Bandwidth-limitation (P2), State-synchronization (P3), and Cloud-latency (P4)) which mobile clients face when consuming cloud hosted services:

- The architecture adapts the CloneCloud technique proposed by Chun et al. [6] in service-centric mobile computing. The focus is to bring services closer to the mobile clients in the local cloud [51] instead of keeping the services in a distant cloud, to reduce the latency (P1) introduced by the number of hops, and saves the bandwidth related cost (P2).

- In addition, the architecture replicates the cloud hosted services on both the middleware and the mobile as per the dual caching model proposed by Liu et al. [38]. The focus of the architecture is not keeping the entire services from the backend cloud on the middleware; rather, the middleware caches the services on the user devices in advance. This allows mobile clients to make a local call [24] [28] to access resources in its own space without making a remote call, thus reducing the latency of resource delivery (P1). Furthermore, the middleware reduces the number of HTTP requests that the mobile client issues to the cloud services to solve the cloud initiated latency problem (P4).

- The architecture includes a technique of application view or state based resource delivery and updates synchronization according to the policy-file. This approach ensures advance synchronization of the related resources based on an application's state on the client device, which minimizes the latency of accessing resources (P1) along with bandwidth consumption (P2). Furthermore, the middleware stores a copy of all the resources in its cache. This allows middleware to support state synchronization of an application within multiple devices of a single user with minimal overhead and latency (P3).

4.3 Middleware Architecture

My proposed policy-based middleware works as an intermediary between the mobile clients and the cloud services. The middleware works like a shop assistant who has advance information about a user's requirements to efficiently deliver services. The information in the policy-file given by the application developers ensures delivery and synchronization of required resources on the client devices in advance. The more declarative the policy-file is, the easier it is for the middleware to support mobile clients with less latency. The middleware keeps a copy the model (collection of resources in client devices cache) from the backend services. The model in the middleware provides the capability of a single user interaction with an application through multiple devices without experiencing much interruption. Moreover, the middleware helps users to start from the state it left in a previous session in case of application crash, as the middleware already knows information about user's previous state. The architecture is shown in Figure 4.3.

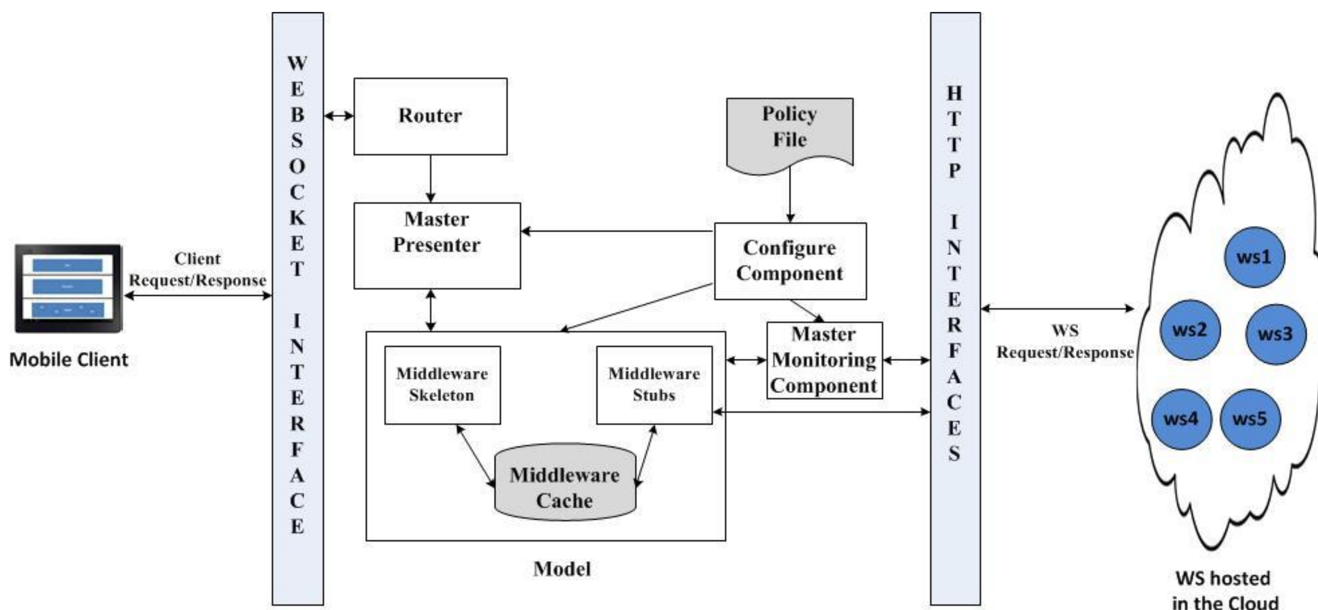

Figure 4.3: Architecture for policy-based middleware

The following section focuses different component of the middleware and interaction among those components:

Router: The router exchanges the data between mobile clients and a master presenter in the middleware. For each of the clients, at first time registration for the middleware services, a master presenter is initialized by the router.

Master presenter: The master presenter delivers required resources to the clients and ensures application consistency. A policy-file received by the master presenter from developers contains all the meta information about different resources associated with the application states. The master presenter starts the configuration component which tunes the behavior of the middleware according to the policy-file. Whenever the master presenter receives state transition information from the client, it checks whether the next possible states are already visited or not. If the next states are not visited, the master presenter component starts the middleware stubs and the middleware skeletons to perform all the allowed operations for each of the resources based on the information in the policy-file. The presenter component saves CPU power by putting idle skeletons and stubs in sleep mode (see Figure 4.4).

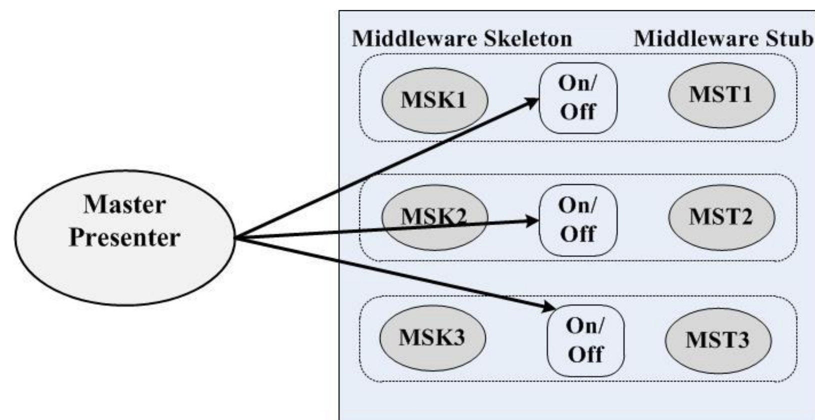

Figure 4.4: Keep active certain part of the model required by the client

For each of the resources which allow get operation, the master presenter delivers the resource to the client besides storing it in the middleware cache. For the state already visited, the master presenter push updates on the client devices.

Configuration: The configuration component is responsible for the formation of the behavior of the different components of the middleware according to the policy-file (see Figure 4.5). The policy-file is a declarative document given by the application developers, which instructs the middleware to reduce bandwidth and latency of cloud-service delivery to the mobile clients.

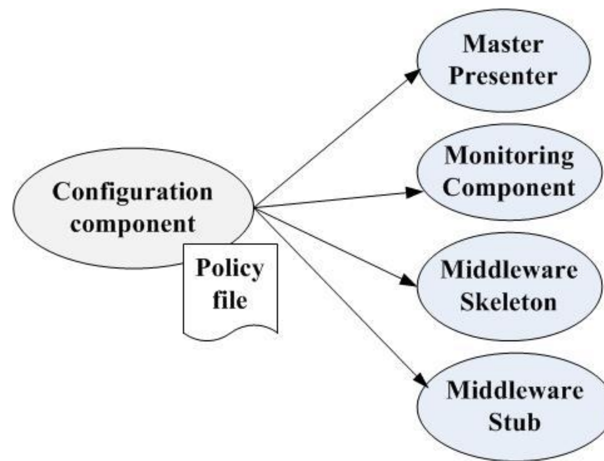

Figure 4.5: Configuration behavior of different middleware components

Model: The middleware model is the collection of the middleware stubs and the middleware skeletons which provide a client the RESTful interface to consume web services hosted in the cloud. The model caches the resources which are cacheable according to the policy-file, and keep cached resources synchronized with the backend services for further client requests.

Middleware stubs: The middleware stub represents the resources from the backend services (see Figure 4.6). Each of the middleware stubs includes methods to perform different operations on a resource. For each request for operation on a resource, the middleware stub gets the

resource URL and other information about operations on the resource from the configuration component. The middleware stub can performs requested operations using the standard HTTP verbs GET (read), PUT (update), POST (create) and DELETE (destroy). The middleware stub also pushes cacheable resources into the middleware cache.

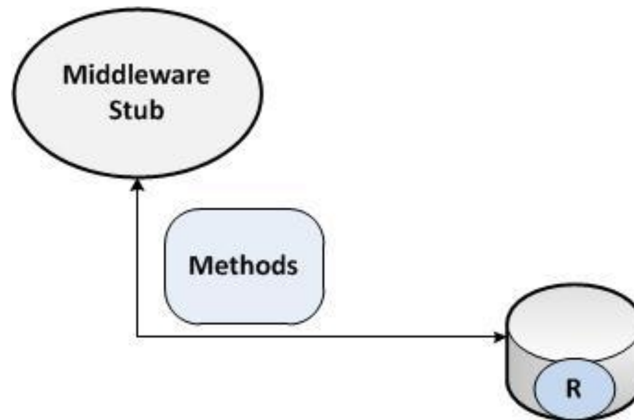

Figure 4.6: Active smart stub interaction with resources in database

Middleware skeletons: The middleware skeleton has the responsibility of content adaptation. It converts initial service responses from the web services into a format which is easily consumable by the mobile clients, such as format conversion (XML to JSON) and image compression.

Middleware cache controller: The middleware stub calls the corresponding cache controller to store and update resources in the middleware cache. The cache controller component starts a monitoring component which is responsible to check for upgrades on a resource with the backend services.

Monitor: A monitor component is maintained for each of the resources which support HTTP GET operation. The responsibility of the monitoring component is to periodically check for updates for the cached resources. The update frequency is specified in the policy-file. The monitoring component checks for updates in backend services by sending a HEAD request. Then the monitor checks the ETAG from the backend server included in the response of the HEAD request; this backend ETAG is compared with the middleware ETAG for that resource. If the ETAG is different, the monitoring component calls the corresponding stub to update the resource in the middleware cache by sending a GET request to keep the cached data fresh (see Figure 4.7).

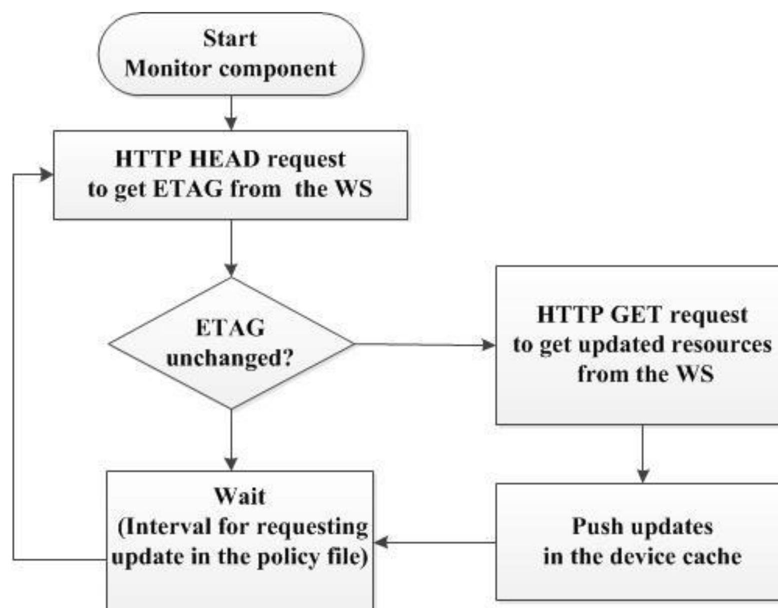

Figure 4.7: Sate synchronization by the monitor

This middleware provides the following features to improve the interaction between the mobile clients and the services hosted in the cloud:

- **Routing:** All communications between the web services hosted in the cloud and the mobile clients are routed through the middleware. The mobile clients can perform different operations on the resources hosted in the cloud through the middleware.

- **Application view based resource delivery and state synchronization:** The middleware has the responsibility to push all the related resources with which the user may interact soon on next states on the device cache, based on state information provided by the client. In addition, the middleware keeps track of services on client devices with the backend services. The middleware pushes updates of resources in the client devices as soon as a client is closer to interact with those resources.

- **Content adaptation:** The content adaptation is a very essential feature to deliver services on the mobile client as they have diverse form-factors. Additionally, mobile devices have limitations (e.g., in terms of energy, CPU, memory and network) in comparison to desktop counterparts. The response a middleware receives from the web services may not be optimized. The middleware adapts unprocessed web service responses to make them easily consumable by the mobile clients. The middleware can perform device specific content adaption such as resource size compression or reduction of the resolution of an image. Besides, the middleware can also perform service specific content adaptation according to the policy file, for example converting XML responses from a SOAP-based server into JSON format. The optimization of content by the middleware saves both bandwidth cost and network introduced latency.

- **Quick recovery from intermittent loss of connectivity and support for multiple devices:** The middleware consumes services from the cloud and stores cacheable resources into its cache besides delivering them on the client devices. This model of middleware cache (collection of user subscribed resources) allows the middleware to support mobile clients to quickly recover from crashes along with support access of an application through multiple devices with minimal overhead (see Figure 4.8).

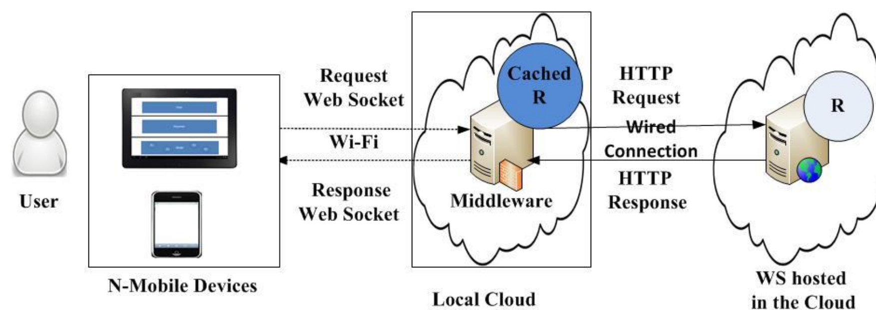

Figure 4.8: Middleware supports N-devices of a single user

- **Delivery of services in a disconnected environment:** The data available in the client cache make the application available to users during the intermittent loss of connectivity. Once the connectivity is restored, the updates on the resources are pushed by middleware on the device cache (see Figure 4.9).

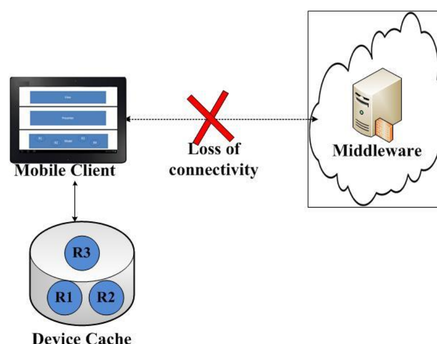

Figure 4.9: Service delivery in the disconnected environment

4.4 The Information for the Policy-File

A policy-file is a document which contains information to configure different components of the middleware support to mobile clients. The motivation of this research is to provide a generic approach instead of providing a device-specific solution. Lasilla [36] identified the importance of policy-aware service delivery and benefits such as optimized data delivery according to mobile device specifications. For example, if a user has limited connectivity it is more important to get priority updates rather than receiving updates for all the resources the application has. This research identified some of the meta information as described below, that the middleware needs to deliver services to the mobile clients:

The information to support application state based delivery and synchronization of cloud hosted services on user devices:

- State transition diagram (see Figure 4.10), includes information about all the states a user can transition between in a certain application.

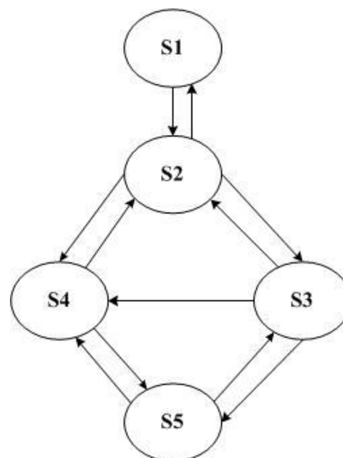

Figure 4.10: An example of the state-transition-diagram

Information about all the resources associated with each state, operations available on those resources and number of states which should be pre-fetched are also included in state transition information. Figure 4.11 shows an example of a State Transition Diagram and other related information in JSON format. This information allows the middleware to deliver and synchronize resources based on an application state in advance.


```
"statetransitdiagram":
[
{  "currentstate": "S1",
"listofpossiblestate":
["S2", "S3"]
},
{"currentstate": "S2",
"listofpossiblestate":
["S3", "S4"]
}]

"resourcelist":
[{ "currentstate": "S2",
   "listofresource":
            ["R1", "R2"]       },
{"currentstate": "S3“,"listofresource": "R3"]
 }]

"op_constrains":

[{"resource":"R1", “operation": ["read“,“write”],
"refreshrate": 5ms

  },

  {"resource": "R2", "operation": ["read"],

   "refreshrate": 10ms

  }]
```


Figure 4.11: JSON representation of State Transition Diagram

The information to support application to support effective caching:

- Information about the result of specific operations on a resource can be cached by the middleware.

- Data on how long a resource can be cached?

- A refreshment policy for cached resources.

The information to support device specific and service specific content adaptation:

- The policy-file has information which guides content adaptation according to device requirements. This information allows the middleware to deliver resources to different devices, in device-adapted form.
- The middleware can also have information to provide service specific content adaptation. Such as middleware can get XML responses from SOAP-based services and convert them into JSON format which is very lightweight and convenient for mobile devices to consume. The optimization of the content by the middleware saves bandwidth cost and reduces latency for accessing services.

The middleware can also include information to provide a single point of authentication [58] towards accessing different services hosted in diverse cloud platforms. Additionally, different policies such as information for priority or context based services delivery and information for service composition can be added in the policy-file. The proposed middleware accepts the policy-file in XML or JSON format to keep the middleware configuration simple.

4.5 Adoption of MVP Design Pattern in the Policy-based Middleware Architecture

This architecture adopts the Model View Presenter (MVP) architecture to provide decoupling of components in the distributed environment. Additionally, including the MVP in the architecture supports application state-based resource delivery and update synchronization. The model, view and presenter are distributed over different machines. This research takes MVP instead of MVC [66] to ensure separation of concern between the view and the model as they are distributed on different machines; details are discussed in the literature review section. The MVP design pattern simplifies the delivery of the model among different mobile devices.

In this architecture, the view is an application screen or state on the client device. The local presenter residing on the client mobile devices is responsible to deliver appropriate model to view. The local presenter also takes the responsibility to communicate with the master presenter residing on the middleware to ensure that all the resources or updates on the resources are available on the client cache in advance. The master presenter treats each of the client devices as a view and takes the responsibility to deliver and push updates for appropriate models to the view. A middleware model provides a RESTful interface [24] for communication with the services hosted on the cloud server. In addition, the presenter pre-processes the resources to make it easily consumable by the mobile clients. The master presenter knows which model to deliver to which view according to the information available on the policy-files. Figure 4.12 shows the adoption of the MVP architecture pattern by the policy-based middleware in the distributed environment. The architecture considers each of the client devices as a view to the master presenter resident in the local cloud.

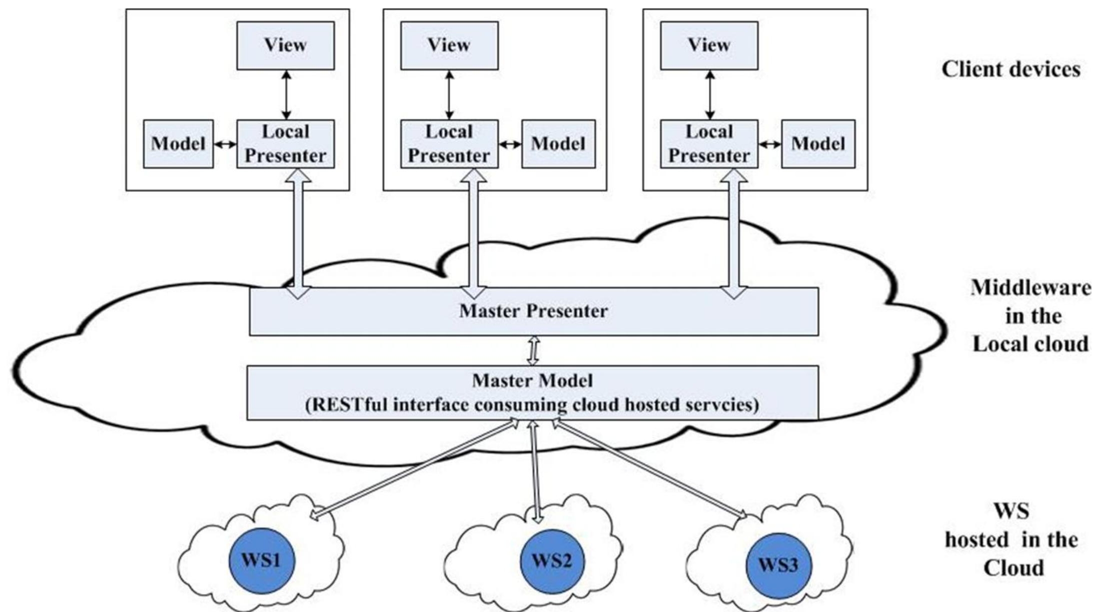

Figure 4.12: The adoption of MVP in distributed environment by the middleware

The local presenter in the client device detects changes in the application state when a client is moving from one screen to another screen. The local presenter informs the presenter in the middleware through messages. The presenter in the middleware gets information from the message about client next possible states. Using that information, the middleware fetches detailed information of all the resources in next states with which a client can interact. Further, the middleware communicates with the model responsible for those resources. The model gets those resources, or updates of those resources, from the cloud hosted server. In addition, the models process the service results to make them suitable to consume by mobile clients. The middleware presenter makes sure the resources or updates related to those resources are available in a client cache in advance. The middleware model contains all the resources that are in the client device cache. This technique provides richer user experiences as middleware can synchronize application state on multiple devices of a single user with minimal latency.

4.6 Mobile Client

In my proposed architecture, the mobile client is the view, which considered as a thin client with fat caching capabilities. A thin client supports computing architecture, in which application processing is possible to be surrogate to a remote server instead of running the application locally. Additionally, the client side architecture adopts the Mobile View Presenter (MVP) architecture to enable loose coupling within its components (see Figure 4.13).

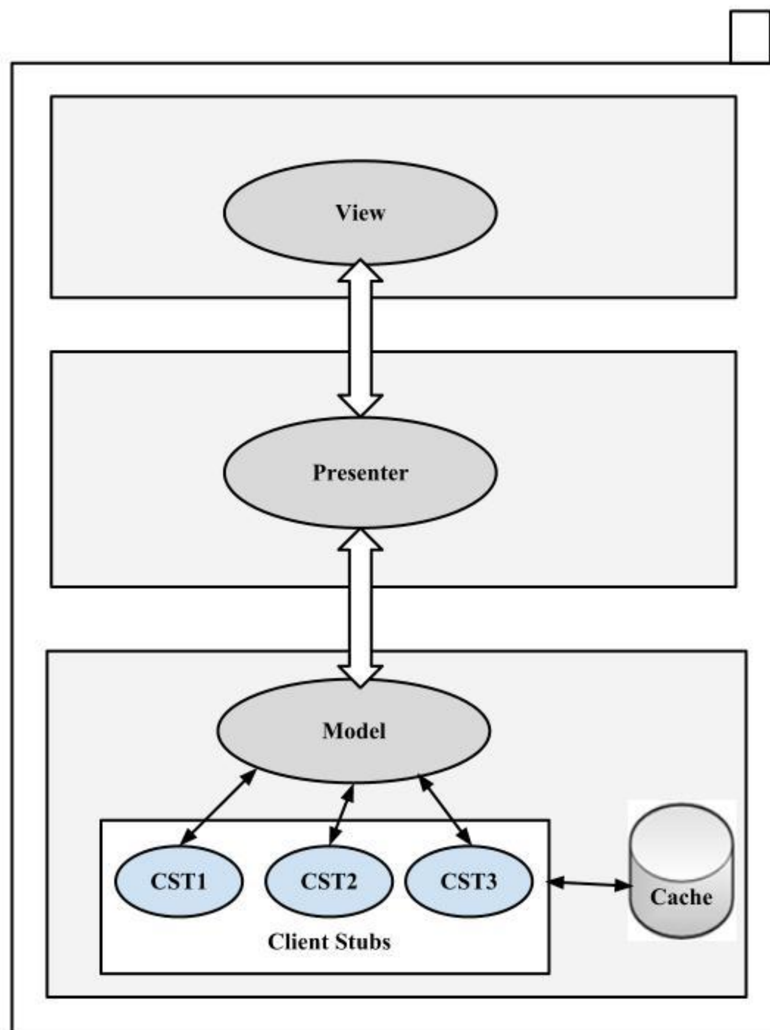

Figure 4.13: Mobile Client Architecture

View: The view component is responsible for issuing events based on the button clicked by the client. It communicates with the local presenter and provides the model view needs for user.

Presenter: The presenter on the mobile devices communicates with the master model component to get or change the model requested by the view. On the temporary loss of connectivity, the presenter is responsible to provide a consistent user experience by using cached data on the device. The presenter in this architecture also takes the additional responsibility to communicate with the master presenter, which resides in the middleware, through a connection component. Whenever the local presenter detects changes in user view, it informs the master presenter. A local presenter receives resources from the master presenter, and pushes those resources onto the device cache by calling appropriate stubs.

Model: The model component in the mobile client is the combination of three components, namely the passive smart stubs, the master model and the connector. Each of the components described below in detail:

- **Master model:** The master model provides all the interactions between different components of the model and the client side presenter. The responsibility of the master model is to communicate with the corresponding smart stubs to satisfy a local presenter request. The pre-fetched resources or updates on them are also pushed by the local presenter on the client cache via the master model.

- **Client stub:** The client stub works as a bridge between cached resources in the mobile client and the master model. Each of the client stubs is a JavaScript object which provides methods to interact with resources in the client cache (see Figure 4.14).

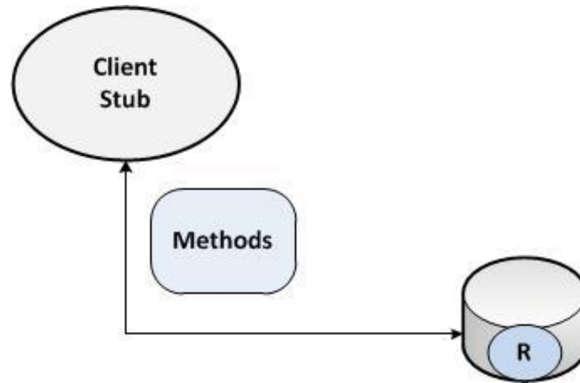

Figure 4.14: Client stub interaction with resources in cache

A client stub represents a single resource or a collection of resources that are associated with a screen. The client stub interprets the JavaScript call from the presenter and returns information requested by the view as a list of JavaScript objects from the model. In this architecture, the client stubs are dumb stubs as they never require communication with the middleware.

- **Client cache:** The client cache is responsible for caching resources required by an application in the client side. Upon the starting of an application, the local presenter caches the resources given by the middleware and pushes the updates accordingly. The architecture uses Web SQL Database [62] technology to cache data on the mobile device.

Connector: The connector establishes the WebSocket connection between a mobile client and the middleware. This connection carries out all communication between the mobile devices and the middleware.

All the interactions between a mobile client and the middleware (Figure-4.15) are as follows:

1. **Register/ De-register for services:** The mobile clients can register or de-register for services to the middleware. At the beginning of an interaction, a client establishes a WebSocket connection with the middleware. The message includes client identity, device identity and security information. If the same client wants to access the same application with other devices, the middleware allows this access with client credentials.

2. **Send state information to the middleware to make sure the required resources are in cache in advance**: The mobile client inform the middleware about client next states, on every occasion the client is moving from one screen to another screen of an application. The information helps the middleware to deliver and update the resources on the device cache in advance, which may be used by the mobile client soon. Furthermore, the middleware saves computational power and the bandwidth cost of dealing with resources that are not presently required.

3. **Request for other operations:** The mobile client can send requests to the middleware to perform operations like create, update, and delete on the resources or services hosted in the cloud.

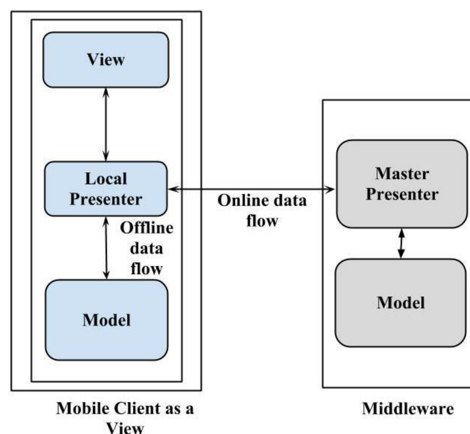

Figure 4.15: Interaction between a mobile client and the middleware

4.7 The delivery of Cloud Services through Policy-based Middleware

In this section, I will continue the example shown at the beginning (section 4.1). The focus is to clarify details on how different components of the proposed architecture interact with each other to satisfy mobile client requests. Figure 4.16 shows the interaction of different screens with different WSs.

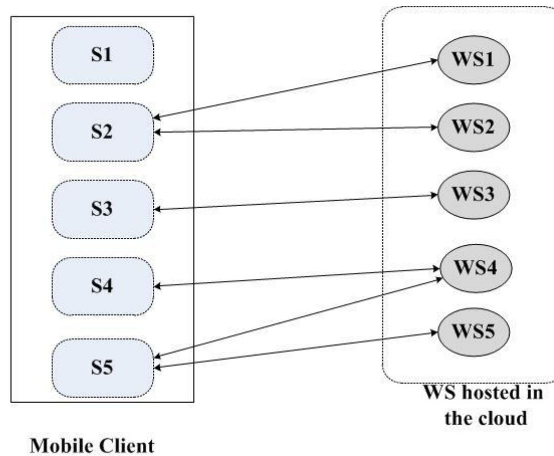

Figure 4.16: Different states of an application interacting with different WSs

Registration and consumption of WS by mobile client through the middleware: In the first step of the communication, the mobile client and the middleware establish the WebSocket connection through which all communication between them is carried. Next the mobile client sends a register command to the middleware with the client id and device information. Whenever the middleware receives a register command, it starts a presenter component with the client id. The master presenter component starts a configuration component, which is responsible for configuring the behavior of the middleware with the information available in the policy-file for the application, given by developers. The configuration component reads the information from the policy-file and configures different components of the middleware accordingly.

For example, from the start screen S1, according to the policy-file, a user can go to screen S2, where it may interact with web service WS1, WS2 or both. The WS1 and WS2 deliver resources R1 and R2 accordingly. Figure 4.17 shows information about different states of the application that a client can visit, and the cloud hosted web services associated with those states with which a client may interact.

```
<?xml version="1.0"?>
<statetransitiondiagram xmlns="http://www.example.org">
    <currentstate>
        <pageid>S1</pageid>
        <listofpossiblestate>
            <nextpossiblestate>S2</nextpossiblestate>
                <listofresources>
                    <resource>R1</resource>
                    <resource>R2</resource>
                </listofresources>
        </listofpossiblestate>
    </currentstate>
    <currentstate>
        <pageid>S2</pageid>
        <listofpossiblestate>
            <nextpossiblestate>S3</nextpossiblestate>
                <listofresources>
                    <resource>R3</resource>
                </listofresources>
            <nextpossiblestate>S4</nextpossiblestate>
                <listofresources>
                    <resource>R4</resource>
        </listofpossiblestate>
    </currentstate>
    <currentstate>
        <pageid>S3</pageid>
        <listofpossiblestate>
            <nextpossiblestate>S5</nextpossiblestate>
                <listofresources>
                    <resource>R4</resource>
                    <resource>R5</resource>
                </listofresources>
        </listofpossiblestate>
    </currentstate>
    <currentstate>
        <pageid>S4</pageid>
        <listofpossiblestate>
            <nextpossiblestate>S5</nextpossiblestate>
                <listofresources>
                    <resource>R4</resource>
                    <resource>R5</resource>
                </listofresources>
        </listofpossiblestate>
    </currentstate>
</statetransitiondiagram>
```

Figure 4.17: An example of state transition info in the policy-file

This middleware stub offers HTTP operation (GET, PUT, POST and DELETE) on the resources hosted in the cloud according to the policy-file. In addition, the policy-file contains information about all the operations that are available on a resource, the URI of the resource, refreshing frequency of the resource. For example, in Figure 4.18, on the resources R1 and R2, GET and POST operations are allowed, while the resource R3 allows only a GET operation.

```xml
<?xml version="1.0"?>
<resources xmlns="http://www.example.org">
    <resource>
        <resourceid>R1</resourceid>
        <resourceuri>www.example.com/R1</resourceuri>
        <listofoperations>
            <method>
            <methodname>GET</methodname>
                <cacheability>cacheable</cacheability>
                <updatefrequency>5 min<updatefrequency>
                <priority>high<priority>
            </method>
            <method>
            <methodname>POST</methodname>
                <cacheability>non-cacheable</cacheability>
            </method>
        </listofoperations>
    </resource>
        <resource>
        <resourceid>R2</resourceid>
        <resourceuri>www.example.com/R2</resourceuri>
        <listofoperations>
            <method>
            <methodname>GET</methodname>
                <cacheability>cacheable</cacheability>
                <updatefrequency>8 min<updatefrequency>
                <priority>high<priority>
            </method>
            <method>
            <methodname>POST</methodname>
                <cacheability>non-cacheable</cacheability>
            </method>
        </listofoperations>
    </resource>
        <resource>
        <resourceid>R3</resourceid>
        <resourceuri>www.example.com/R2</resourceuri>
        <listofoperations>
            <method>
            <methodname>GET</methodname>
                <cacheability>cacheable</cacheability>
                <updatefrequency>10 min<updatefrequency>
            </method>
        </listofoperations>
    </resource>
</resources>
```

Figure 4.18: An example of resource related info in the policy-file

The master presenter delivers all the resources with which the client may interact in their next states, in the client cache in advance. This approach enables mobile clients to make a local call to the resources in its cache without experiencing much delay. The delivery of resources based on application state saves cost of bandwidth along with reducing overhead of the client processing. The presenter starts a cache controller component, to make each cacheable resource available in the middleware cache (see Figure 4.19).

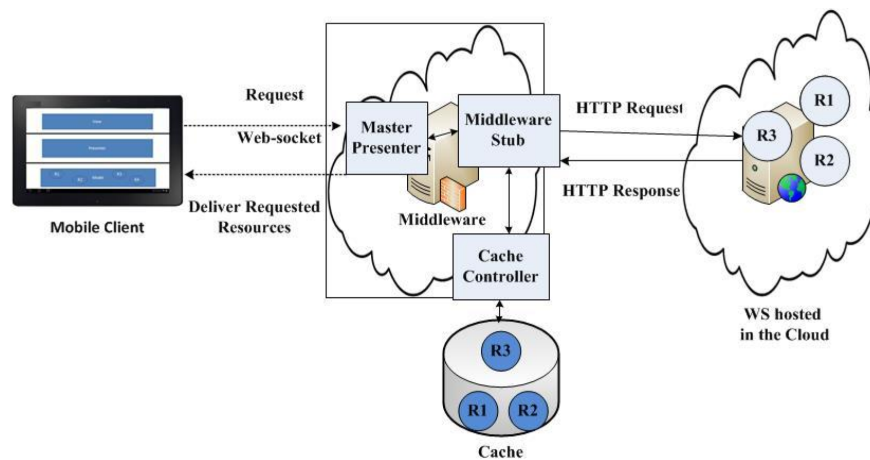

Figure 4.19: Delivery of requested resource by the middleware

Delivery updates to the client by the middleware: The monitor component keeps resources in the middleware cache synchronized with the backend services. A client sends a state update message to the middleware. The master presenter component in the middleware gets information about all next possible states for the client from the configuration component. For already visited states, the master presenter component pushes updates on the client devices. Application state based resource synchronization on the devices saves the bandwidth of sending unrelated data and overhead of client processing (see Figure 4.20).

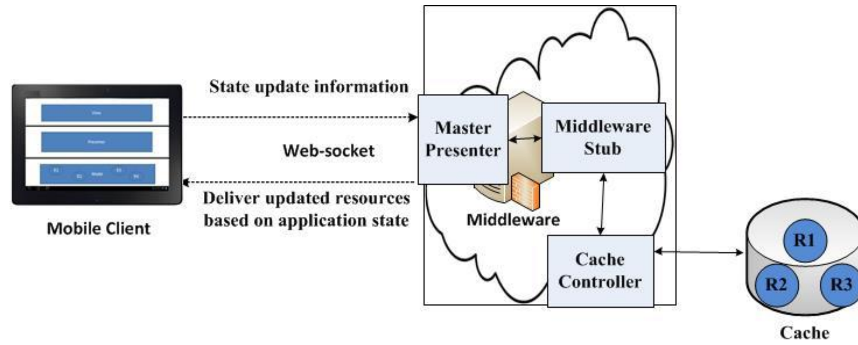

Figure 4.20: Pushing updates to the clients based on the state information

4.8 Summary

An architecture has been proposed in this chapter, which addresses the challenges this research identified of delivering and synchronizing cloud hosted services on mobile devices. The major component of this proposed architecture is the policy-based middleware which supports application state based optimized resources delivery and synchronization on the client devices in advance. The replicated data in the client cache [28] ensure offline availability of the application in the case of a user experiencing intermittent connectivity. The middleware provides a RESTful interface [24] to consume cloud hosted services, which is very suitable for mobile clients. The MVP [66] architecture is adopted for providing better decoupling among the components in distributed environments. Furthermore, the middleware preserves a similar model for each client to ensure consistent user experience of accessing apps with multiple devices of a single user with minimal overhead. The concept of using the local cloud [51] to host the middleware also helps to reduce the latency of accessing services as it brings the web services closer to the user. In the next chapter, details are provided about the implementation which was done to evaluate the performance of the proposed architecture.

CHAPTER 5
IMPLEMENTATION

This chapter details the implementation of a prototypical policy-based middleware. The policy-based middleware is integrated with an agriculture related application, called "*MobiCrop App*", which supports farmers with information for decision making such as pesticide solutions. The forthcoming discussions focus on the justification of using various programming languages and technologies for the implementation.

5.1 Mobile Client Implementation

The implementation of the mobile client application comprises three layers: the view, the presenter, and the model. The view or user interface layer is implemented as an embedded browser using HTML5, CSS and jQuery mobile [26] web technologies. The local presenter is implemented using JavaScript, which responsible for delivering the appropriate model to the view components. Additionally, the local presenter provides means of communication between the stubs residing on the client device and the master presenter on the middleware to get and synchronize resources. The model component is also implemented using JavaScript. Each of the stubs represents a single resource, or a collection of resources from the model component, and each stub is a JavaScript method. Using libraries offered by PhoneGap [47] to implement the architecture with HTML5, grants embedded bowsers full access to the device level features in most of the popular mobile platforms, e.g. BlackBerry, Android, iOS and Windows Phone. The development of the mobile application using platforms such as PhoneGap, enables compiling the mobile Web app as a native application. Additionally, PhoneGap–like libraries provide the benefit of deploying the same code in various mobile platforms, save the effort of building a native application for each different mobile platform in different programming languages.

The mobile clients use the WebSocket [61] interface to communicate with the middleware. The WebSocket is a communication protocol that facilitates bi-directional communication, over a single Transmission Control Protocol (TCP) socket. Though WebSocket has an initial overhead of establishing a connection, it has a benefit when the data flow is repetitive between a client and a server [28]. The WebSocket removes the limitation of HTTP, which requires sending request and response headers for each connection. Figure 5.1 represents different technologies and protocols, adopted in different layers, for the mobile client implementation.

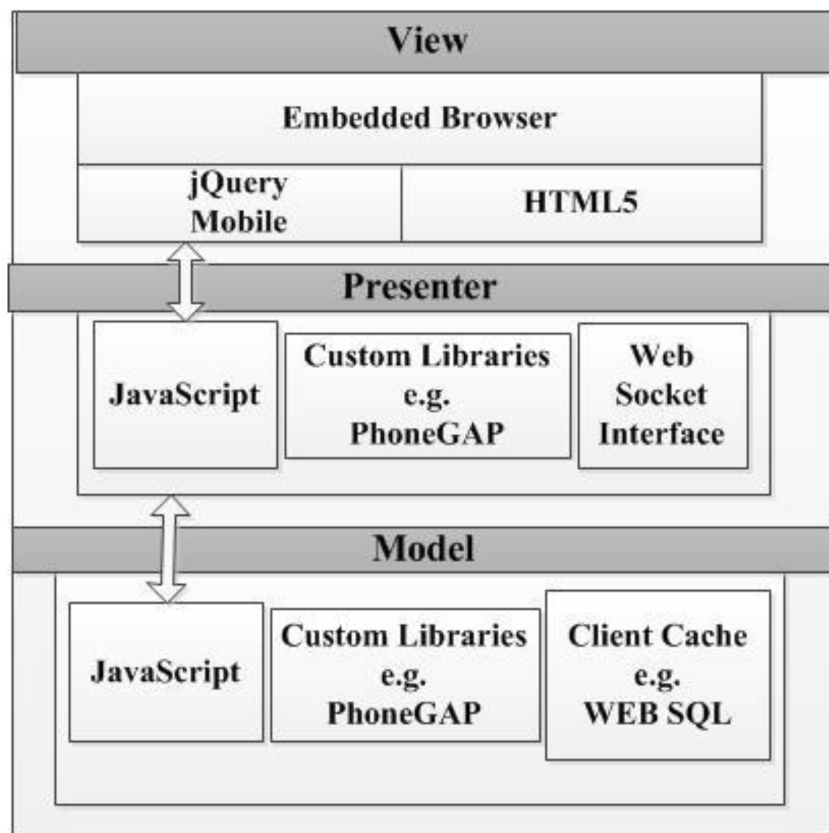

Figure 5.1: Adopted technologies and protocol towards developing hybrid app

5.1.1Client Cache

The client cache is implemented using the Web SQL Database API [62], which allows the user to store data in a structured manner on the client devices. The API provides a simple client-side storage mechanism for relatively small amounts of data using HTML5 and JavaScript. The Web SQL database supports SQL syntax, which is easy to adopt by application developers. The cache in a client device works as performance booster and reduces the latency of requesting data from a server. The API also ensures supports for major mobile browsers (Android Browser, Safari). Figure 5.2 shows the creation of the database, table and insertion of data using Web SQL Database API.

```
//Create and open a database
func.webdb.openDB = function() {
       console.log("creating database");
       var dbSize = 5 * 1024 * 1024; // 5MB
       func.webdb.db = openDatabase("agricultureDB", "1.0", "agricultureDB", dbSize);
       if (func.webdb.db != null){
      console.log("database created");
       }
}
//Create table and fill in data
func.webdb.createTables = function() {
       var db = func.webdb.db;
       db.transaction(function(tx) {

       tx.executeSql("CREATE TABLE IF NOT EXISTS crops(crops_id INTEGER, crops_name TEXT)");

//Inserting data in a table
       for ( var i = 0; i < dataLen; i++) {
       var temp_id = i;
       var temp_name = "barley";
       tx.executeSql("INSERT INTO crops(crops_id, crops_name) VALUES (?, ?)",
                    [temp_id, temp_name], null, func.webdb.onError);
       }
});
```

Figure 5.2: Code snippet to create table and store data in local databases

Figure 5.3 represents the screen shots of the client side application. The application enables farmers to access cloud hosted services that provide pesticides information. The different technologies and protocols discussed above are adopted to implement the application. The architecture ensures service availability during intermittent loss of connectivity by pushing services in the cache of the devices.

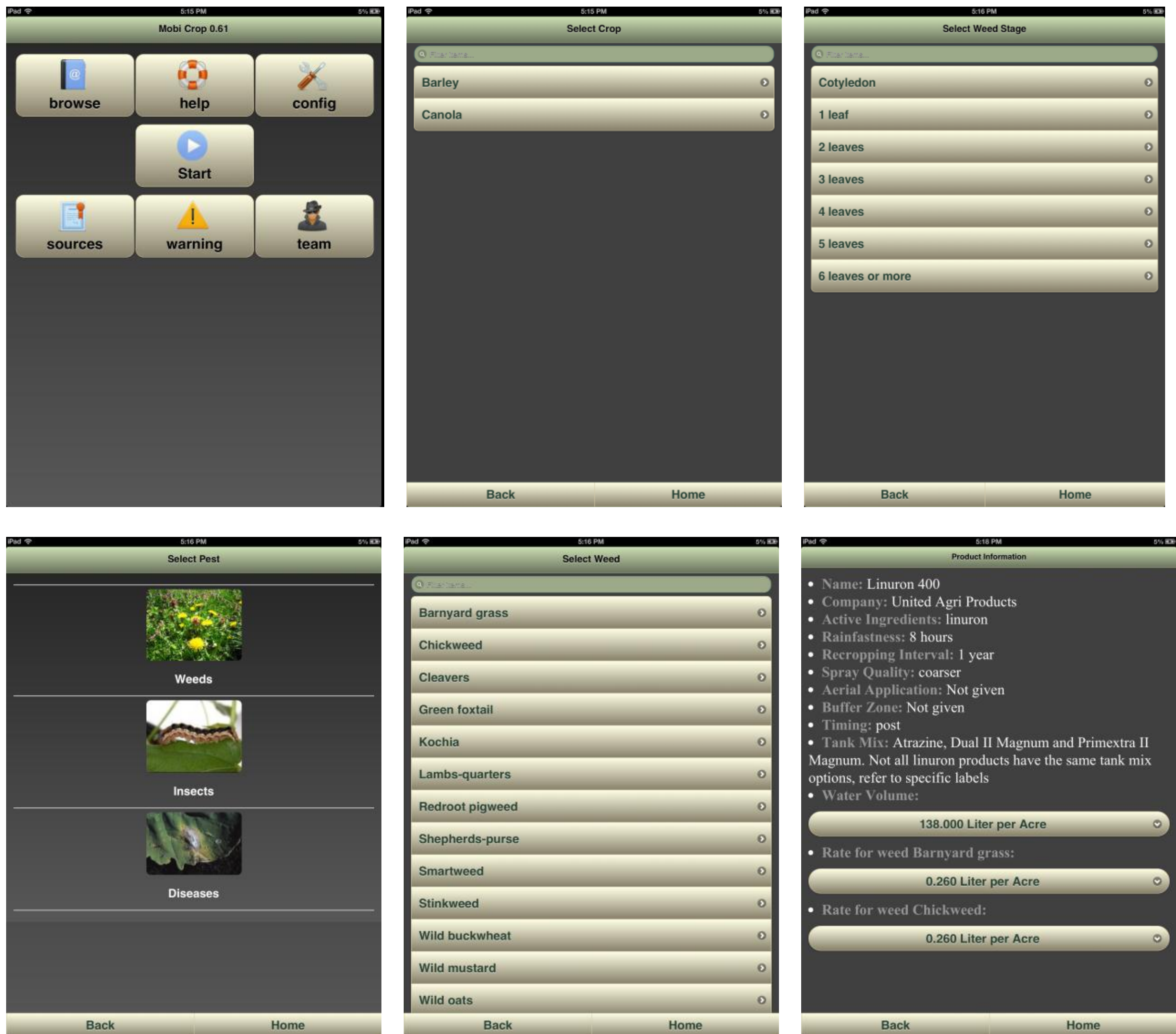

Figure 5.3: Screenshots of the MobiCrop App on an iPad

5.2 Middleware Implementation

The middleware is implemented in the Erlang [2] programming environment since the language supports concurrency in distributed systems. Each of the components of the middleware is built using the Generic Server Behavior (gen_server) process (module). This process supports asynchronous communication between the mobile participants and the middleware, which is an application server. This module also includes functionalities for error reporting and debugging. The middleware is hosted on YAWS 1.92 server [65], which is an HTTP server written in Erlang, and supports WebSocket communication. Additionally, YAWS supports a large number of concurrent connections as it is designed to handle each connection via a separate Erlang process. The following sections provide details about the implementation and communication protocol for the different major components of the policy-based middleware.

5.2.1 Router

All communications from the mobile clients are transmitted through the router, which is a component of the middleware. The integration of rfc4627 Erlang module allows the router to encode and decode JSON objects into Erlang code for processing by the middleware. The mobile client can send a register command to the router. The router creates a master presenter component and configures the behavior of the middleware for the application using information in the policy-file given by an application developer. For each of the update messages from the client, the router sends state information to the corresponding master presenter to deliver the resources that the clients may require urgently. Additionally, the router forwards requests for different operations on resources, for instance add, update and delete, to the middleware, from the mobile clients. The following code snippet (Figure 5.4) represents the extraction of data by the router module for further processing by the middleware.

```
%% Router module
%% Get client request from the JSON message
Message="{"action": "update","newstate": "S3"}"

{ok,ErlangText,[]}=rfc4627:decode(Message),

{ok,Action}=rfc4627:get_field(ErlangText, "action"),
```

Figure 5.4: Code snippet for extracting client request from JSON

5.2.2 Master Presenter

The master presenter component starts a configuration component, at the beginning of communication with each mobile client. The configuration component configures the behavior of the middleware according to the given policy-file provided by an application developer. The master presenter component retrieves different information about user requirements from the configuration component. According to the state information provided from a mobile client, the master presenter component calls different stubs to perform the user requested operations. When the master presenter receives the response from the middleware stub for a read operation, it both pushes the data in the middleware's model, and sends the resources, to the device (Figure 5.5). Furthermore, the master presenter keeps resources in the client cache synchronized with the backend services.

```
%%Master Presenter Module
%%Push resources to client cache in advance according to applicationstate
handle_call({client_stateInfo, Client_state}, _From, Tab) ->
      %%Get next possible sates from the configuration component.
      [{_,Next_possible_states}]= gen_server:call({global, 'my_configure_component'},
      {getdata, 'state_transition_information', "S3"}),
      List_of_allresources = getallresourcelist(Next_possible_states),
      Resources_List=getresourcesfromserver(["crops", "weeds"]),
      {ok,JsonOb, []}=rfc4627:decode("{}"),
      Client_Operation= rfc4627:set_field(JsonOb,"Resources",Resources_List),
      Client_Obj=rfc4627:set_field(Client_Operation,"Operation",'Insert'),
```

Figure 5.5: Code snippet of application for the mater presenter

5.2.3 Configuration Component

The configuration component configures the behavior of different components of the middleware according to available information in the policy-file. The current implementation accepts a policy-file in the JSON format as shown in Figure 5.6.

Policy_file= "[{"listofstates": ["S1","S2","S3","S4"]},{"statetransitiondiagram": [{"currentstate": "S1","listofpossiblestate": ["S2","S3"]},{"currentstate": "S2", "listofpossiblestate": ["S3","S4"]}]},{"resourcelist":[{"currentstate": "S1","listofresource":["R1","R2"]},{"currentstate": "S2","listofresource":["R3"]}]},{"operational_constrains":[{"resource": "R1","operation": "read", "refreshrate": 5ms},{"resource": "R2","operation": "read","refreshrate": 10ms},{"resource": "R3","operation": "read","refreshrate":10ms}]}]"

Figure 5.6: Policy-file given to the configuration component

Figure 5.7 shows a code snippet for the configure component which is responsible for configuring behaviour of different components, based on the policy-file.

```
%%Configure Component module
%%Get state transition diagram from policy file
      {ok,Statetransitiondiagram}=rfc4627:get_field(Statetransition_diagram,
      "statetransitiondiagram"),

%%Get current state and next possible states from the state transition diagram
insert_std_table(Std_info)->
      {ok,Current_state}=rfc4627:get_field(Std_info, "currentstate"),
      Currentstate=binary_to_list(Current_state),
      {ok,Nextpossible_states}=rfc4627:get_field(Std_info, "listofpossiblestate"),
      Nextpossiblestates = convert_tolist(Nextpossible_states),
%%Push in local DETS table for use by the master presenter component
      insertdata('my_configure_component', 'state_transition_information',
      Currentstate, Nextpossiblestates).
```

Figure 5.7: Code snippet for processing policy-file by the configuration component

5.2.4 Middleware Stub

The middleware stub provides HTTP interface to consume resources hosted in the cloud. The middleware uses the HTTPC module provided by Erlang, to perform different operations on the resources. The module supports HTTP/1.1 compatible clients and allows the middleware to use the following HTTP methods: HEAD, GET, PUT, and POST. The code snippet (Figure 5.8) shows HTTP GET and POST operation to the server by the middleware stub.

```
%%Middleware Stub Module
 Url = {"http://xoxo.usask.ca:8080/agri_db_service/crops"
%%Handle for GET request to server
handle_call({getdata, Url}, _From, State) ->

%% HTTPC GET request to server.
       {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =
       httpc:request(get, {Url, []}, [], []),
       [_, {_, Etag }, _, _, _] = Headers,
       NewState = #state {etag = Etag},
       {reply, Body, NewState};


%%Handle for POST request to server
JsonOb="{\"id\":3, \"name\": \"wheat\"}",

handle_cast({postdata, Url}, State) ->

       %% HTTPC POST request to server.
       {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =

       httpc:request(post, {Url, [],"text/html", JsonOb },[], []),
       {noreply, State};
```

Figure 5.8: Code snippet for GET and POST by the middleware stub

Figure 5.9 shows the JSON response from the RESTful Web server.

```
[{"crops": [{id: 1,"name": "wheat"}, {id: 2,"name": "oats"}]
```

Figure 5.9: Middleware response for GET

5.2.5 Middleware Cache

The cache controller module provides different methods to cache data in the middleware and retrieve or update them (Figure 5.10). The middleware uses the DETS storage facility of Erlang to cache the consumed data from the cloud hosted services. In order to provide faster response time, data in a DETS table is pushed to ETS (Erlang Term Storage) tables. The ETS tables are stored in the RAM of the computer of the middleware, and thereby have the capability of providing faster lookup times. The middleware maintains a separate cache that is the combination of a DETS and an ETS table for each of the registered users. The middleware caching makes it possible for the architecture to provide quick recovery to the users when an application crashes. Besides, middleware caching enables accessing the same application from multiple devices of a single user, with low latency.

```
%%Cache Controller Module
%%Insert data in to middleware cache
handle_cast({insertdata, TableName, ResourceName, Data}, State) ->
       dets:insert(TableName, {ResourceName, Data}),
       dets:to_ets(TableName, TableName),
       Reply ="Data Inserted",
       {noreply, State};


%%Gives cache data back to the client
handle_call({getdatacache, TableName, ResourceName}, From, State) ->
       Reply = ets:lookup(TableName, ResourceName),
       io:write("Data Retrieved"),
       {reply, Reply, State};
```

Figure 5.10: Code snippet for Read and Write operations by cache controller

5.2.6 Monitor

The middleware stub starts a monitor component for each of the cacheable resources. The monitor component synchronizes data in the client cache with the cloud hosted services based on an application state. The handle_info function (code snippet in Figure 5.11) is called after a fixed interval, to check for updates in the backend services. The monitor component issues an HTTP HEAD request and gets the "ETAG" from the server for a resource. An "ETAG" or entity tag is a mechanism provided by HTTP, to validate cache. The use of "ETAG" allows caches to be more efficient, thus saving bandwidth as Web server does not need to send the full response if the resource's state has not changed. If the "ETAG" of the resource is different from the available "ETAG" in the middleware, the middleware updates the resources, sending an HTTP GET request to the cloud hosted server.

```
%%Monitor Module
%%The handle_info function is repetitively called on a fixed interval specified policy-file.
handle_info(interval, State)->
%%Head request to get ETAG from the Web server.
   {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =
    httpc:request(head, {"http://xoxo.usask.ca:8080/agri_db_service/crops", []}, [], []),
    [_,{_, Etag}, _, _] = Headers,
       Etag_server= Etag,
%% Provides ETAG for the resources available in the middleware.
       Etag_middleware=middleware_stub:getetag(Resource_name),
%%Compares ETAG on the middleware for the resources with the EATG from the Web server.
if Etag_server==Etag_middleware->
       io:write("No updates"),
       {noreply, State};
   Etag_server/=Etag_middleware->
      io:write("update required"),
%%Get updated resource form Web server.
     Updated_Resource=
     middleware_stub:getdata(Stub_name,"http://xoxo.usask.ca:8080/agri_db_service//crops"),
%%Push updates in the middleware cache.
     cache_controller:insertdatacache('middlewarecache', 'middlewarecache',
     "Resource_name",Updated_Resource),
     {noreply, State}
end.
```

Figure 5.11: Code snippet for monitor module

5.3 Summary

This chapter provides details about different technologies, programming languages and protocols adopted towards prototype implementation of the proposed framework.

The client side application is implemented as a hybrid application (i.e., same functionalities as a native app) using HTML5 and other Web frameworks (e.g. PhoneGap). The adoption of this mechanism provides application interoperability to different mobile platforms (e.g. BlackBerry, Android, iOS and Windows Phone). Furthermore, the chapter includes details about adoption of technology to enable offline availability of the application and a communication mechanism to reduce the latency of consuming cloud hosted services.

The middleware is implemented using the Erlang programming environment, since the language supports concurrency in distributed systems. Each of the components of the middleware is built using the Generic Server Behavior (gen_server) process (module), to support asynchronous communication between the mobile participants and the middleware. The implementation adopts YAWS server (HTTP server written in Erlang), to support a large number of concurrent processes. Besides, YAWS provides support for HTTP protocols like WebSocket which enables delivery of resources with low latency and reduced bandwidth consumption. The middleware cache is implemented using two leading version of NOSQL databases, namely Erlang DETS and MongoDB.

The next chapter provides details of different experiments to evaluate performance and overhead of using the policy-based middleware.

CHAPTER 6
EXPERIMENTS

This chapter describes the experiments that were performed to evaluate the performance of a policy-based middleware. The goal of the experiments is to investigate how the architecture affects the latency of accessing cloud hosted services by mobile clients. In addition, the experiment investigates the overhead introduced by the middleware to the system. This chapter also includes details of the experimental setups, tools and the workloads used for different experiments. Furthermore, a summary and evaluation of the results are presented in the last section.

6.1 Experiment Goals

The following is the lists of experiments (Table 6.1) used to evaluate the design of the middleware in terms of the problems (network-latency, bandwidth-limitation, and state-synchronization) mentioned in Section 2.1:

Table 6.1: Experiment Goals

Experiments Name	Experiment Goals
1. Latency Experiments	To evaluate the importance of different information in the policy-file, to reduce the latency of resource delivery and state synchronization.
2. Experiment to support multiple devices model synchronization	To evaluate middleware support model synchronization on the multiple devices of a single user and latency of synchronizing the model.
3. Overhead Experiments	To evaluate the overhead, introduced by the policy-based middleware.
4. Experiments to evaluate middleware performance in local cloud	To evaluate the advantages of hosting in the local cloud, in comparison to hosting it on a distant cloud.

6.2 Experimental Setup

The experiments are simulated in order to provide the observer (i.e. the person conducting the experiments) to have more control over the environment. The RESTful Web server is hosted in an environment identical to the Amazon EC2 (see Figure 6.1). The mobile clients are outside of the cloud infrastructure.

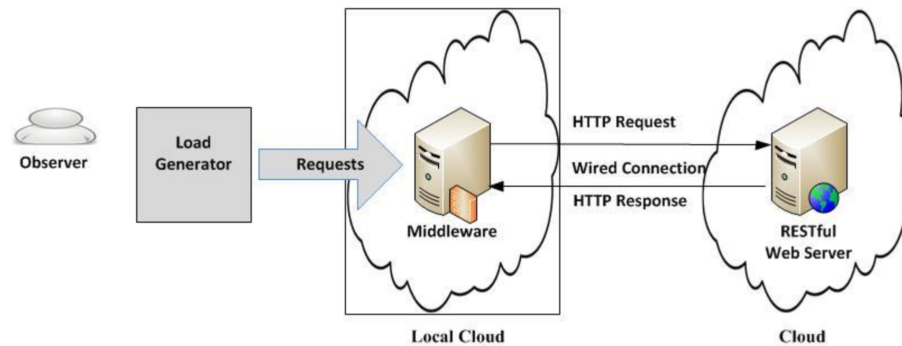

Figure 6.1: Experimental setup

An application developed for the mobile client is used to generate requests to evaluate the latency over the Wi-Fi networks towards consuming cloud hosted services. The empirical data related to these experiments are recorded by an observer for analysis of the system behavior.

Mobile client specification:

- Asus EEE Pad Transformer TF101

Processor: NVIDIA Tegra 3

1.2GHz

RAM: 1 GB

Operating System: Android 4.1, Jelly Bean

- Google Nexus 7 Tablet:

Processor: Quad-core Cortex-A9 processor

1.3 GHz

RAM: 1 GB

Operating System: Android 4.0, Ice Cream Sandwich

- Desktop client:

Processor: Intel (R) Core(TM) 2 Quad CPU

Q9400@ 2.66 GHz

RAM: 4 GB

Operating System: Windows 8 Enterprise version, 64-bit operating system

Middleware machine specification:

The Middleware was hosted on the local cloud with following specifications:

Processor: Intel (R) Core(TM) 2 Quad CPU

Q9400@ 2.66 GHz

RAM: 8 GB

Operating System: Windows 8 Enterprise version, 64-bit operating system

Web-Server machine specification:

Processor: Intel (R) Xeon (R)

CPU E5410@ 2.33 GHz

RAM: 16GB

Operating System: Windows 8 Enterprise version, 64-bit operating system

6.3 List of Experiments

This section describes the list of experiments that were conducted to measure the performance of the proposed architecture:

Experiment 6.3.1: Latency Experiments

In these experiments, the latency of consuming cloud hosted services is measured with and without the policy-based middleware to determine the importance of different information (caching, pre-fetching and content adaptation) in the policy-file, as it affects the efficiency of consumption of cloud hosted services. The experiments include the following phases:

Experiment 6.3.1.1: The importance of caching and pre-fetching information in the policy-file of the middleware.

For these experiments, firstly the request-response time is measured for requesting resources directly from the Web server. The size of each resource is of 24 bytes. The HTTP GET requests are sent in a closed JavaScript loop. Each of the HTTP request is sent sequentially after the preceding request-response interaction is complete. Then the request-response time is measured for requesting the same resources pre-fetched and cached in the client device by the policy-based middleware.

Test 1: For this test, the HTTP GET request is sent, using Google Nexus7, directly to the RESTful Web server hosted in the cloud (Figure 6.2).

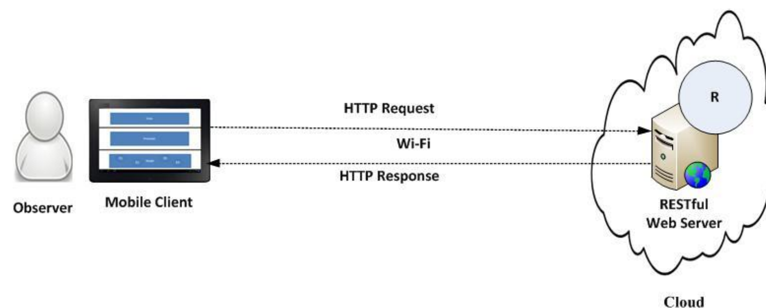

Figure 6.2: Set-up for requesting resources directly from the WS

The test was repeated ten (10) times on each round, with the number of requests varying from 1000 to 10000. The result of the average, minimum and maximum request-response times between the mobile client and the cloud hosted Web server are presented as a graph in Figure 6.3.

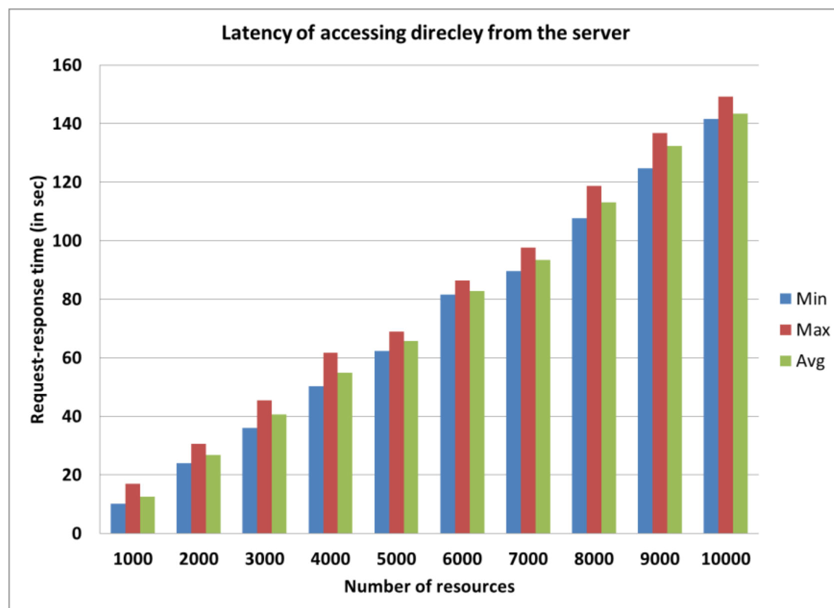

Figure 6.3: Graph of the latency of resource delivery from the RESTful WS

Test 2: In the second test, the same resources are requested from the client cache of Google Nexus7 and the request-response time is measured (Figure 6.4).

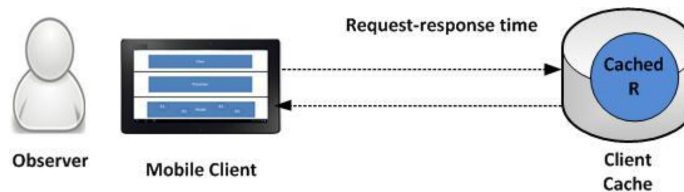

Figure 6.4: Set-up for client requesting resources from the device cache

The test was also repeated ten (10) times on each round, with the number of requests varying from 1000 to 10000. The result of the average, minimum and maximum request-response times are shown in Figure 6.5.

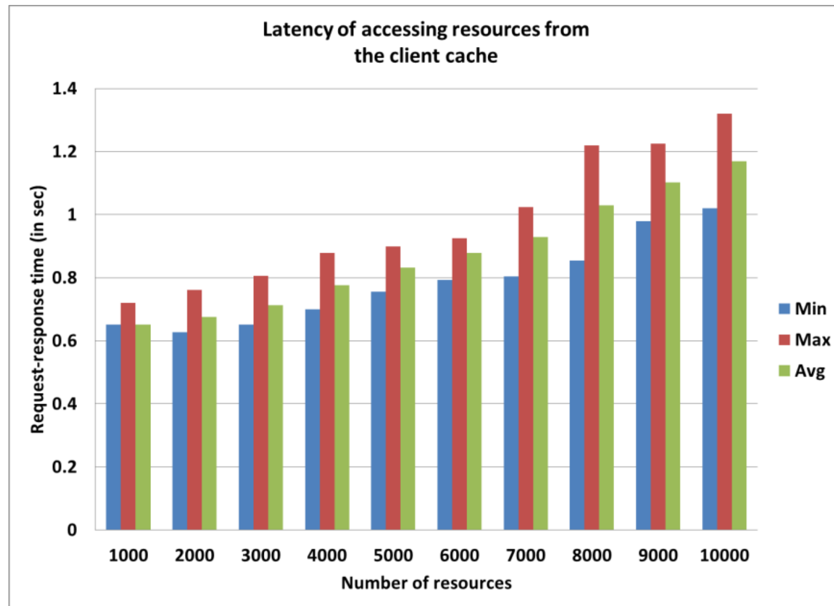

Figure 6.5: Graph of the latency of resource delivery from the client cache

6.3.1.1 Results and discussion

The results from Test 1 and Test 2 reflect that caching and pre-fetching information in the policy-based middleware reduces the latency of resources delivery to the mobile clients. Figure 6.6 shows the graph of the average latency for delivering resources for each of the request with and without the policy-based middleware.

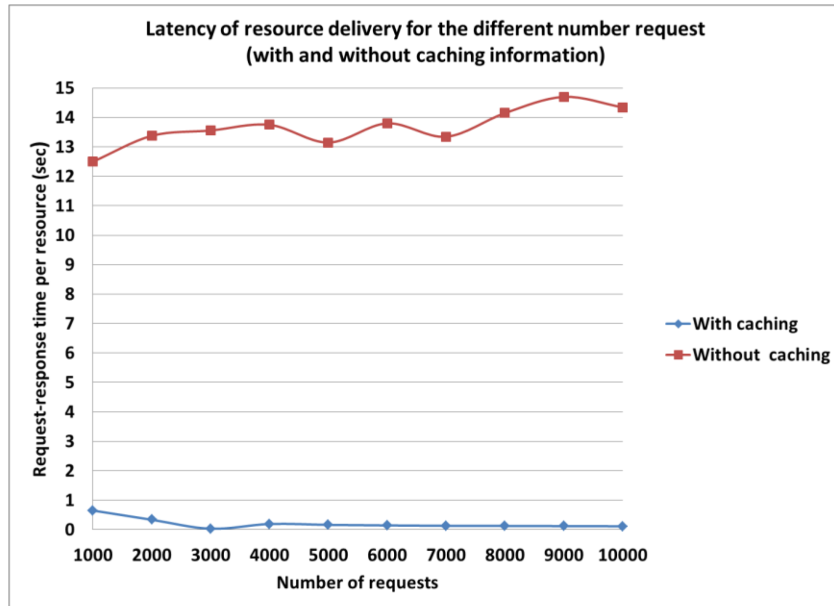

Figure 6.6: The Graph of the latency of resource delivery with and without the policy-based middleware

Table 6.2: Results comparing average request-response time of Test 1 and Test 2

	Average Request-response time for each resources (sec)
Direct request to the WS	12.50
Client cache	0.652

The results in Table 6.2 show that it takes an average of 12.50 seconds to requests each resource directly from the Web server by the mobile client, while it takes only 0.652 seconds to request the same resource from the client cache. The caching and pre-fetching of information in the policy-file reduce the latency of resource delivery by 19.17 times.

Experiment 6.3.1.2: The importance of content adaptation information in the policy-file of the middleware.

These experiments evaluate the performance of resource delivery with and without content adaptation information on the policy-based middleware. The average latency and data delivered towards consuming SOAP-based web services by a mobile client with and without the content adaption information in the policy-file, are compared. The requests for resources are made by an Erlang client in a sequential loop. The SOAP-based Web server calculates the average of given five numbers. The size of each of the SOAP messages (Figure 6.7) sent to the server is 320 bytes and the response message from the server is 228 bytes for each request.

Request SOAP message:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Body>
<i:calcAverage xmlns:i="http://www.example.org">
<i:argument>3</i:argument>
<i:argument>7</i:argument>
<i:argument>1</i:argument>
<i:argument>123</i:argument>
<i:argument>17</i:argument>
</i:calcAverage>
</env:Body>
</env:Envelope>
```

Response message from the SOAP server:

```
"<sp:Envelope xmlns:sp=\"http://www.w3.org/2003/05/soap-envelope\">
<sp:Body><out:calcAverageResult xmlns:out=\"http://www.example.org/out\">
<out:okResult>30.200</out:okResult></out:calcAverageResult>
</sp:Body></sp:Envelope>"
```

Figure 6.7: Request-response SOAP message

Test 1: The request-response time and amount of data delivered to the client is measured to request the resources directly from the SOAP based Web server (Figure 6.8).

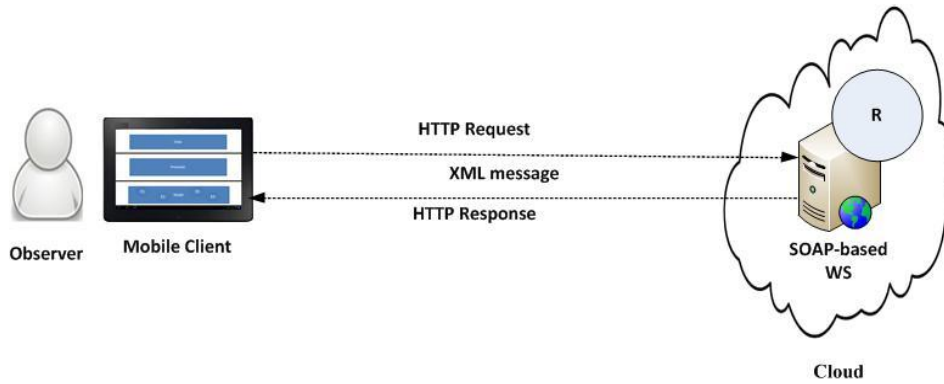

Figure 6.8: Set-up for consuming resources from the SOAP-based Web server directly

The average, minimum and maximum round-trip time to consume 100 resources (10 runs) is shown in the Table 6.3.

Table 6.3: Request-response time for consuming SOAP-based WS directly

Minimum request-response time (ms)	**Maximum request-response time (ms)**	**Average request-response time (ms)**
282	286	283.2

Test 2: For this test**,** the requests are made through the policy-based middleware to consume resources from the SOAP-based Web server (see Figure 6.9). The middleware converts the XML response messages from the SOAP server into a lightweight JSON message with required information, and is delivered to the mobile clients.

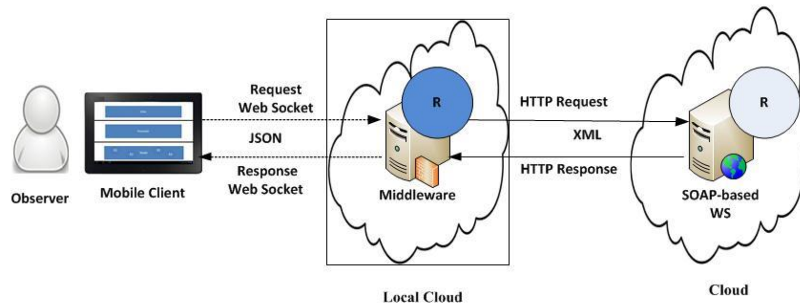

Figure 6.9: Set-up for client consumption of resources from SOAP-based WS through middleware

The size of the JSON message sent to the middleware for consuming same resource as above is 83 bytes, (Figure 6.10) and the response from the middleware which contains result is 28 bytes.

Request message to the middleware:

{calcAverage:[{argument1:3,argument2:7,argument3:1,argument4:123,argument5:17}]

Response message from the middleware:

{"calcAverageResult":30.200}

Figure 6.10: Client request message and response message from the middleware

The average, minimum and maximum round-trip time to consume 100 resources (10 runs) is shown in the Table 6.4.

Table 6.4: Request-response time of consuming SOAP-based WS through the middleware

Minimum request-response time (ms)	**Maximum request-response time (ms)**	**Average request-response time (ms)**
230	261	238.9

6.3.1.2 Results and discussion

Table 6.5: Results comparing request-response time for SOAP services

Consuming SOAP services	Average request-response time (ms)	Data send size (bytes)	Data received size (bytes)
Without the middleware	283.2	32000	22800
With the middleware	238.9	8300	2800

The results in Table 6.5 show that it takes an average of 283.2 milliseconds to return 100 resources from the SOAP-based Web server without the middleware, while it takes 238.9 milliseconds to get the same number of resources through the middleware. The content adaptation information in the middleware decreases the latency by 18.54% in comparison to direct access to the services by mobile clients. Furthermore, the middleware decreases the data overhead for the mobile client by 3.86 times in terms of sending and 8.14 times in terms of receiving. The policy-based middleware reduces the bandwidth cost of the consuming services using mobile devices. Furthermore, the middleware deliver information in JSON format which is lightweight and easy to process by the mobile clients.

Experiment 6.3.1.3: Evaluation of the impact of state transition information to support application state based resource-state synchronization by the policy-based middleware.

These experiments evaluate the importance of information about application states (e.g. resources in different states, allowed operations on the resources and update frequency) and state transition information in the policy-file of the middleware for efficient resource state synchronization. The first tests verify the advantages of using the middleware push for state

synchronization. The second test is done to evaluate the importance of application state based resource synchronization by the policy-based middleware. For these experiments, the requests are made sequentially with an Erlang client in a loop. The response size for each HTTP HEAD request (Figure 6.11) is 153 bytes, while the response size for HTTP GET (Figure 6.12) is 234 bytes.

HTTP HEAD Request
Request URL:http://xoxo.usask.ca:8080/agri_db_service/cropsdb
Request Method:HEAD
Accept: Application/json
Erlang Web server Response
HTTP/1.1 200 OK
Content-Type:application/json
Date:Sun, 12 May 2013 19:53:51 GMT
Etag:StvoZZx626uxuAc6DIFB9w==
Server:Yaws 1.94

Figure 6.11: HTTP HEAD request and response from the server

HTTP GET Request
Request URL:http://xoxo.usask.ca:8080/agri_db_service/cropsdb
Request Method:GET
Accept: Application/json
Erlang Web server Response
Content-Length: 73
HTTP/1.1 200 OK
Content-Type:application/json
Date:Sun, 12 May 2013 19:53:51 GMT
Etag:StvoZZx626uxuAc6DIFB9w==
Server:Yaws 1.94
Resource
[{id: 2, name: wheat}, {id: 1, name: oats}, {id: 3, name: barley}]

Figure 6.12: HTTP GET request and response from the server.

Test 1: Initially, I measured the time to update resources directly from the server by the client. The total time for resource synchronization by client pulling includes the time for client sending a HTTP HEAD, the comparison time of the ETAG value from the HEAD and sending a HTTP GET (Figure 6.13) request to get the updated resource from the server.

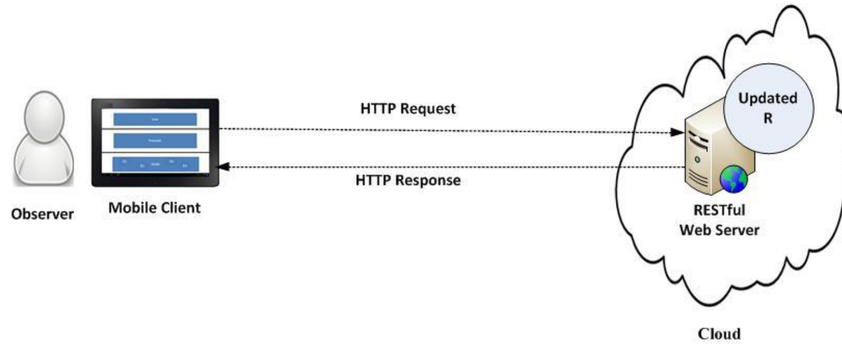

Figure 6.13: Set-up for update synchronization by the client pulling

Table 6.6 shows the average time for state synchronization and the amount of data received by the client from the server hosted in the cloud for 100 resources. Each test is repeated 5 times.

Table 6.6: Average time and amount of data for state synchronization without the middleware

No of Resources	**Average time for state synchronization (ms)**	**Data (bytes)**
100	1702.6	38800

Test 2: In this test, I measured the time for resource state synchronization and amount of data delivered to the client for the middleware push technique (Figure 6.14). Table 6.7 shows the average time for state synchronization and amount of data received by the client from the server for 100 resources through the middleware push, also repeated 5 times.

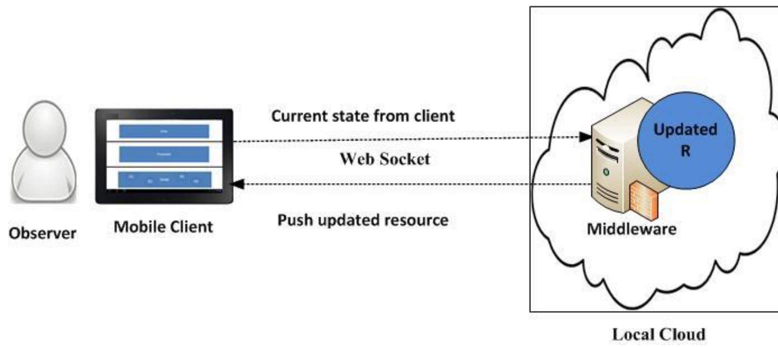

Figure 6.14: Set-up for update synchronization by the middleware push

Table 6.7: Average time and amount of data for state synchronization with the middleware

No of Resources	Average time for state synchronization (ms)	Data (bytes)
100	664.1	7300

Test 3: In this test, I measured the latency and the amount of data delivered to the client without information about application states. The experimental setup remained the same as with Figure 6.14. Here, I consider a demo application with five screens, receives 100 updates for each of the screen (6.15).

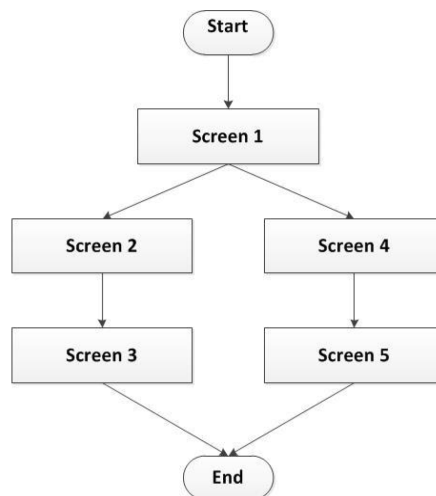

Figure 6.15: Application with 5 screens, with each state having 100 resources updates

Table 6.8 shows the average time for state synchronization and amount of data received by the client from the middleware for 100 resources in the each screen or state of the application (5 runs) through middleware push.

Table 6.8: Average time and amount of data receiver by the client for state synchronization without the state-transition information

Application states	**Average time for resources state synchronization (ms)**	**Data (bytes)**
S1, S2, S3, S4, S5	3320.5	36500

Test4: Now, suppose for the application shown above (Figure 6.15), users are only going to visit three screen or states (S1, S2, and S3). The middleware, with state-transition information in the policy-file, (Figure 6.16) only synchronizes updates for those resources the client is going to use. Table 6.9 shows the average time for state synchronization and the amount of data received by the client from the middleware for 100 resources in each screen or state of the application (10 runs) through the middleware push.

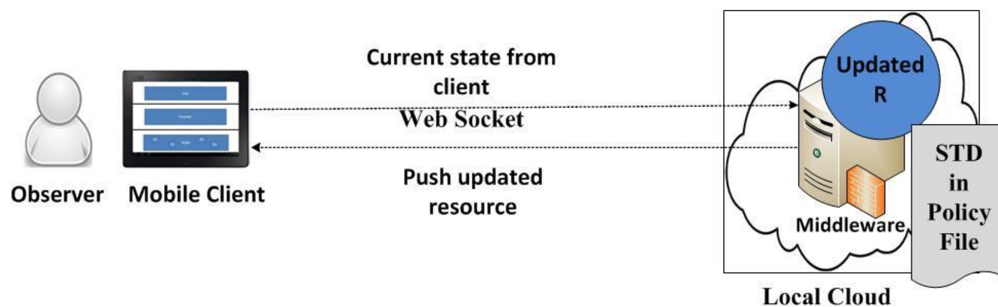

Figure 6.16: Set-up for update synchronization with the state-transition information

Table 6.9: Average time and amount of data receiver by the client for state synchronization with the state-transition information

Application states	**Average time for resources state synchronization (ms)**	**Data (bytes)**
S1, S2, S3	1992.3	21900

6.3.1.3 Results and discussion

Table 6.10: Results comparing Test 1 and Test 2 (State synchronization)

	Average time for state synchronization each resources (ms)	**Data (bytes)**
Client Pull	17.02	388
Middleware Push	6.64	73

The results in Table 6.10 show that it takes on an average 17.02 milliseconds for state synchronization for each of the resources using client pull, while it takes on average 6.64 milliseconds for the middleware push. The client pull increases the latency of state synchronization by 156.32% compared to middleware push. Furthermore, the client pull approach for state synchronization has around 5.31 times data overhead to the mobile client compared to middleware push, which increases bandwidth cost for service consumption from the mobile devices.

Figure 6.17 shows the graph which compares the average latency of state synchronization for each resource for Test1 and Test2, with the number of resources increased from 10 to 100 (each of tests is repeated 5 times).

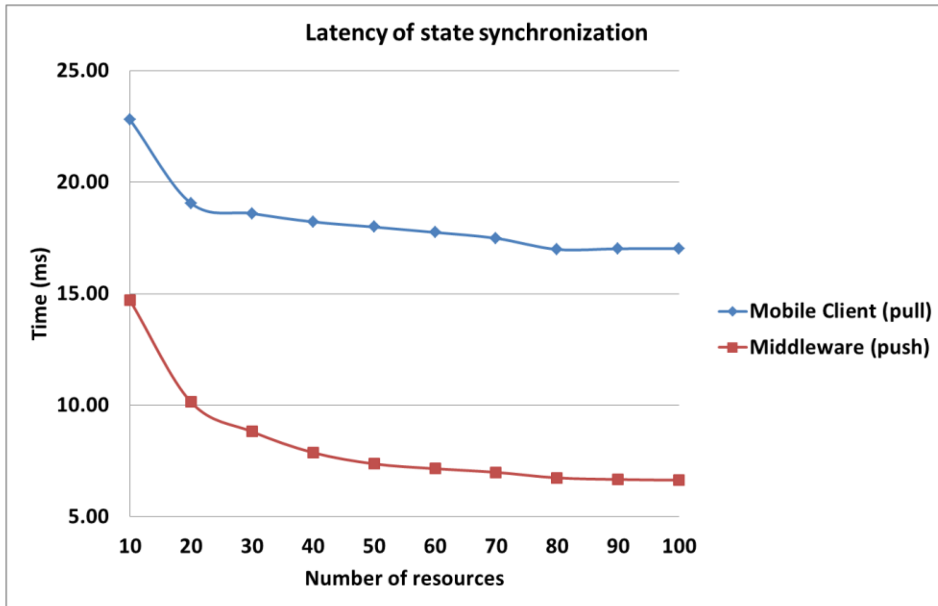

Figure 6.17: Graph of latency of state synchronization client pull vs. middleware push

Table 6.11: Results comparing Test 3 and Test 4 (STD info in policy-file)

Middleware Push	**Average time for state synchronization (ms)**	**Data (bytes)**
With STD information in the policy-file	3320.5	36500
Without STD information in the policy-file	1992.3	21900

The results in Table 6.11 reflect that application state transition information in the policy-file of the middleware reduces data overhead for the state synchronization by 1.67 times; the performance improvement of the system is a reduction of latency of state synchronization by 66.67%.

The graph in Figure 6.18 shows the latency of state synchronization for different refresh-rates in the policy-file of the middleware. The number of resources in each state of the application was increased from 10 to 100 (each of tests repeated 5 times).

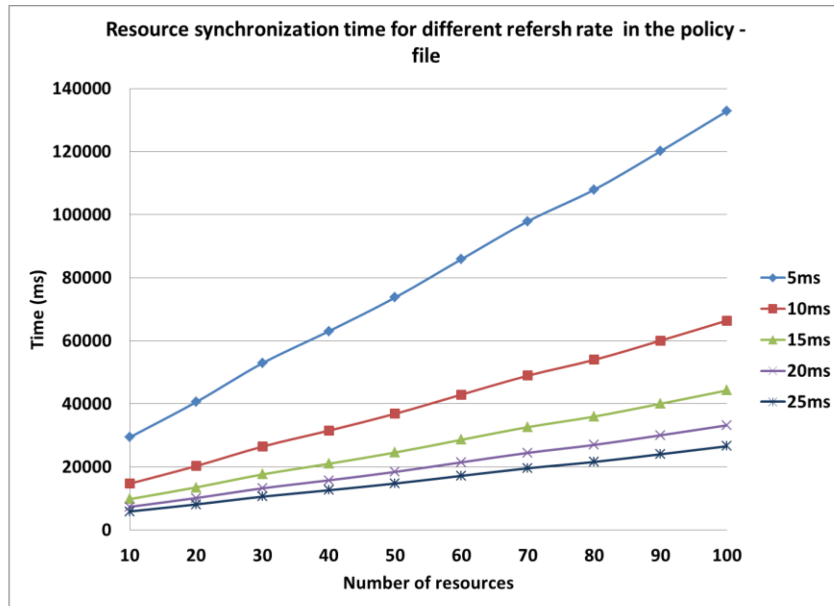

Figure 6.18: The latency of state synchronization for different refresh rates in the policy-file

Experiment 6.3.2: Experiment for support multiple devices model synchronization.

These experiments verify the support of the middleware to synchronize the model (collection of resources subscribed by a user) to different devices of the user (Figure 6.19). The experiments also measure latency of model synchronization among different devices having different operating systems. For these experiments, a model in the middleware is a collection of 1000 resources and each resource's size is 73 bytes.

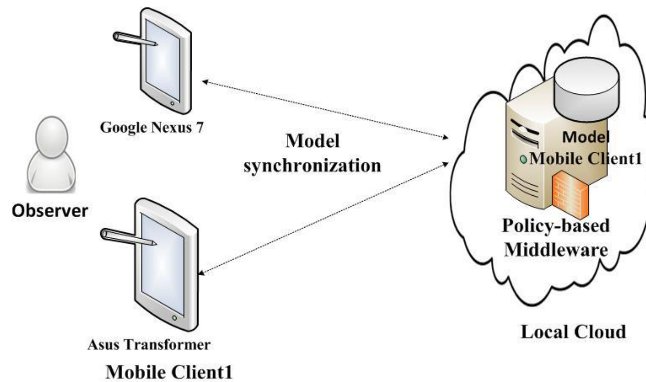

Figure 6.19: Set-up for model synchronization with multiple devices of a single user

Test 1: In the first test, the requests are sent to get a model for a user that is available in the middleware, with a Google Nexus7 tablet that has Android 4.0, Ice Cream Sandwich as the operating system. Table 6.12 shows the average time to synchronization for a model with 1000 resources from the middleware (10 runs).

Table 6.12: Average time of synchronization for a model in Google Nexus7

Model	Average time for model synchronization (ms)
1000 resources	7381.6

Test 2: I repeated the same experiment to get the model from the middleware with 1000 resources using ASUS EEE pad transformer (TF101) tablet with Android 4.1, Jelly Bean as operating system. Table 6.13 shows the average time to synchronization for a model with 1000 resources from the middleware (10 runs).

Table 6.13: Average time for synchronization of a model with in ASUS EEE pad

Model	**Average time for model synchronization (ms)**
1000 resources	10451.1

6.3.2 Results and discussion

Table 6.14: The average time for synchronization of a model (middleware vs. Web server)

Model **Google Nexus7**	**Average time for model** **synchronization (ms)**	**Data (bytes)**
1000 resources (middleware)	7381.6	73000
1000 resources (WS)	12497	234000

The results in Table 6.14 shows that it takes an average of 7381.6 milliseconds for the middleware to perform model synchronization (1000 resources) on the mobile clients, while the time for synchronizing the same model by the client from the Web server is 12497 milliseconds (Section 6.3.1, Test 1). The result shows the model synchronization from the middleware reduces the latency by 69.30% in comparison to getting resources directly from the Web server. Furthermore, the middleware reduces the data overhead of the model synchronization 3.20 times.

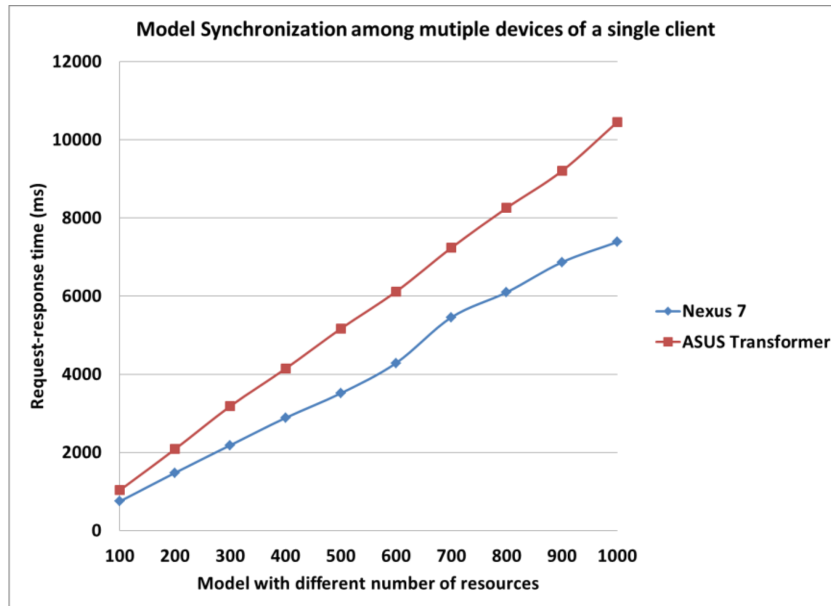

Figure 6.20: The average latency for the model synchronization

Test1 and Test2 above verify the support of model synchronization of the middleware to the multiple devices of a single user with different operating systems and configuration. Figure 6.20 shows the graph which compares the average latency of model synchronization for Test 1 and Test 2, where the number of resources in the model increased from 100 to 1000. According to the figure, average time for model (1000 resources) synchronization in Google Nexus7 is 7381.6 milliseconds, while the average time to synchronize is 10451.1 milliseconds. The Google Nexus7 has 1.41 times performance benefit in terms of latency in comparison to ASUS Transformer as Nexus7 has a more powerful CPU and smaller screen size.

Experiment 6.3.3: Overhead Experiments

The goals of these experiments are to evaluate the overhead introduced by the policy-based middleware in terms of latency. Different tests are carried out to evaluate the overhead introduced by caching, pre-fetching information in the policy-file, middleware caching, mixing Read-write operations and state-synchronization.

Experiment 6.3.3.1: The evaluation of the overhead of resource delivery by the policy-based middleware because of caching and pre-fetching information.

These experiments evaluate the overhead of introducing policy-file including caching and pre-fetching information. The latency is measured with and without using caching and pre-fetching information in the policy-files of the middleware. The size of each resource for these experiments is 24 bytes. An Erlang client is used to make sequential requests for the resources.

Test 1: Firstly, I measured request-response time to get resources from the Web server using middleware without a policy-file (Figure 6.21).

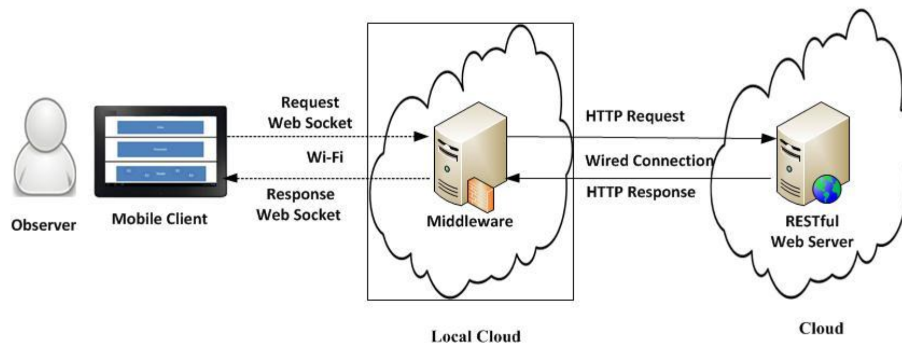

Figure 6.21: Set-up for client requests routed through the middleware without the policy-file

The tests are repeated five (5) times and the average round-trip time for getting 5000 resources from the Web server by the client are recorded as shown in Table 6.15.

Table 6.15: Average request-response time using the middleware without the policy-file

No of resources	Average request-response time (ms)
5000	26442.4

Test 2: For the second test, I introduced caching and pre-fetching information in the policy-file and configured the middleware accordingly. Afterwards, the request-response time is measured as above using policy-based middleware to evaluate overhead introduced by the information in the policy-file (Figure 6.22).

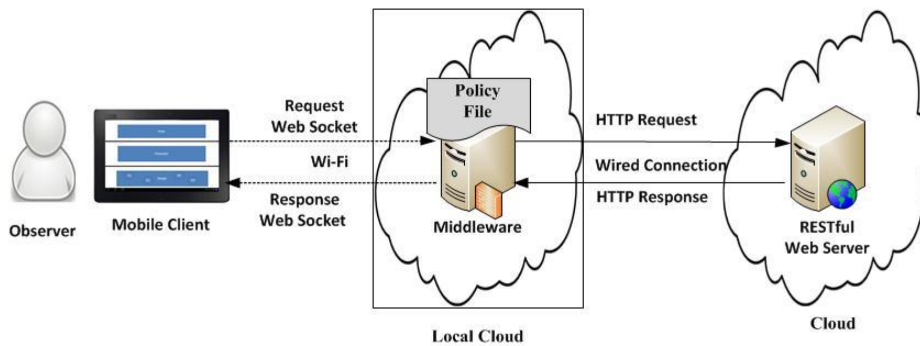

Figure 6.22: Set-up for client requests routed through the middleware with the policy file

The tests are repeated five (5) times and average request-response time for getting 5000 resources from the Web server by the client shown in Table 6.16.

Table 6.16: Average request-response time, using the middleware with the policy-file

No of resources	Average request-response time (ms)
5000	34461.4

6.3.3.1 Results and discussion

Figure 6.23 shows the graph of the average duration for a request-response interaction through the middleware in both tests.

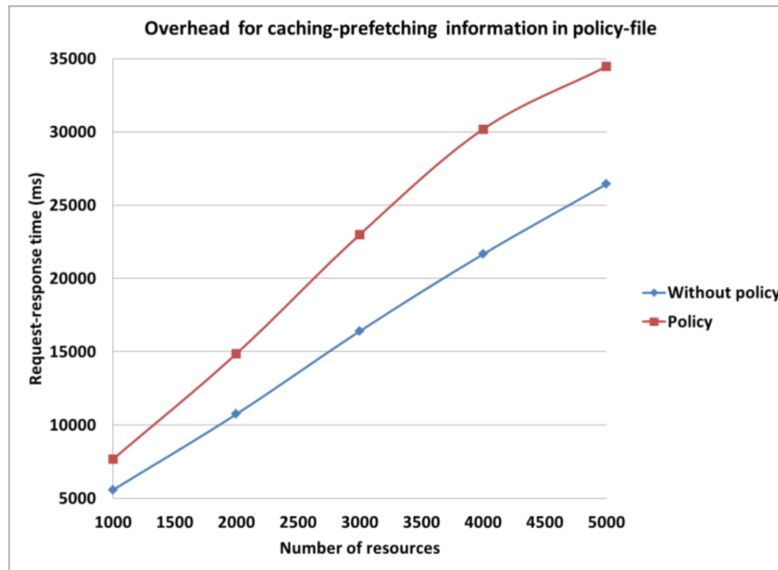

Figure 6.23: Graph of the overhead of caching and pre-fetching information in the middleware

Table 6.17: Results comparing middleware overhead for caching and pre-fetching

Middleware	Average request-response time (ms)
With policy-file	5.28
Without policy-file	6.89

The results in Table 6.17 shows that the request-response time for getting a resource without policy-file are 6.89 milliseconds, whereas the time requires to get the same resource using caching and pre-fetching information in the middleware is 5.28 milliseconds. The caching and pre-fetching of information in the policy-file add a latency overhead of 30.49% for each resource.

Experiment 6.3.3.2: The evaluation of the middleware caching overhead of using different No-SQL databases.

These experiments evaluate the overhead of using different No-SQL databases as proxy caches. I measured the latency of requesting resources using Erlang DETS table and MongoDB [42] as client caches. The size of each of the resources is 92 bytes. The requests for resources are made sequentially by clients written in Erlang and JavaScript.

Test 1: Firstly, request-response time is measured to retrieve resources from the Erlang DETS table. The tests are repeated five (5) times and the average request-response times for obtaining 5000 resources from the DETS table are shown in Table 6.18.

Table 6.18: Request-response time to obtain resources from the DETS table

Average request-response time (ms)	**Maximum request-response time (ms)**	**Minimum request-response time (ms)**
29.2	30	28

Test 2: In this test, I measured request-response time to get resources from the MongoDB. The tests are repeated five (5) times and the average request-response times for obtaining 5000 resources from the MongoDB are recorded, are shown in Table 6.19.

Table 6.19: Request-response time to obtain resources from the MongoDB

Average request-response time (ms)	**Maximum request-response time (ms)**	**Minimum request-response time (ms)**
143.6	146	140

6.3.3.2 Results and discussion

Figure 6.24 shows the graph which provides comparison of the performance of MongoDB and Erlang DETS table for read operation where the number of requests is increased from 1000 to 5000.

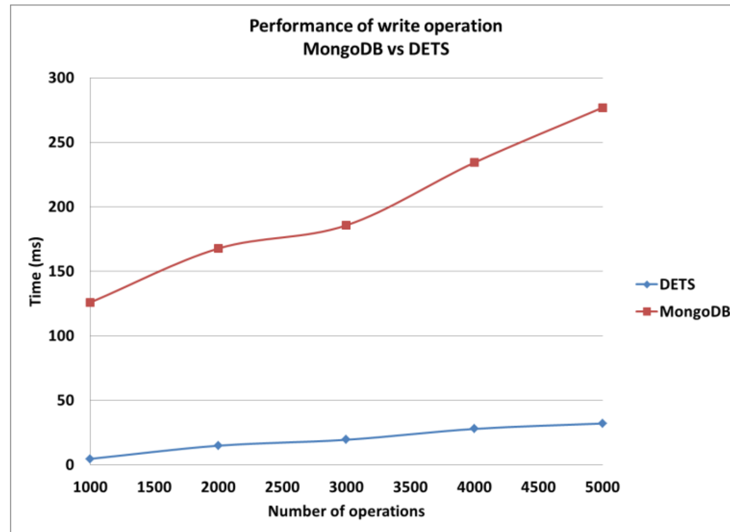

Figure 6.24: Graph of performance of the read operations of MongoDB vs. DETS table

Table 6.20: Results comparing Test 1 and Test 2(overhead of MongoDB vs. DETS)

Middleware platform	Average request-response time for 5000 resources (ms)	Maximum request rate (request/s)
DETS (Erlang)	29.2	171232.87
MongoDB	143.6	34818.94

The results (in Table 6.20) show that the use of DETS table as the middleware cache has the capability of serving around 4.92 times more requests/sec than the MongoDB under the same conditions. In addition, DETS shows a lower average request-response time.

One of the major disadvantages MongoDB has when used in the Erlang environment is that the interfacing tool has a very high overhead, around 41.80 times that of Erlang DETS, as shown in the graph of Figure 6.25. After 5000 requests the behavior of the tools is inconsistent and often crashes (Figure 6.26).

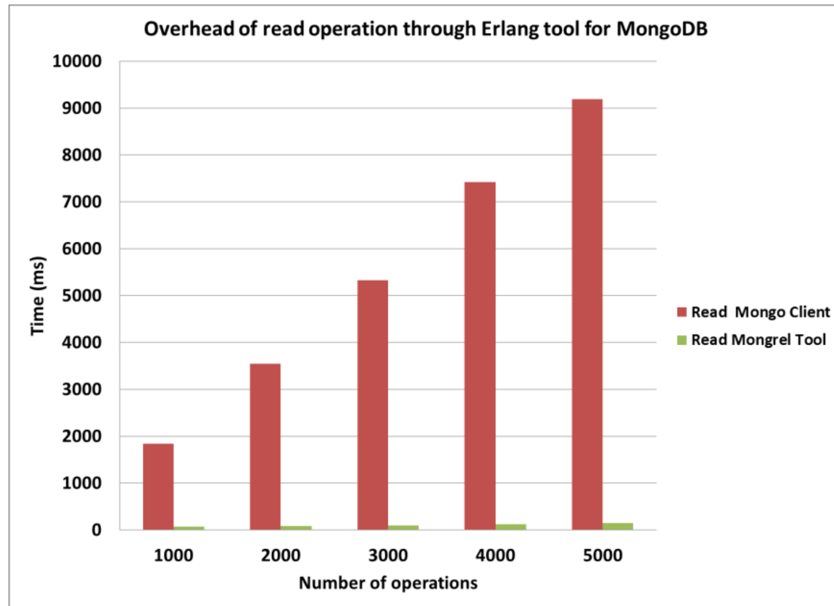

Figure 6.25: Overhead of Erlang interfacing tools for MongoDB

```
Erlang
File  Edit  Options  View  Help

Erlang R16B (erts-5.10.1) [64-bit] [smp:4:4] [async-threads:10]

Eshell V5.10.1  (abort with ^G)
1> mongo_timmer:time_post_cache(6000).
{1368,911960,718000}{1368,911976,940000}0
2> mongo_timmer:time_post_cache(6000).
{1368,912053,289000}** exception exit: {badmatch,{error,system_limit}}
     in function  book_database:populate/0 (c:/Users/ashik/workspace/MongoDB/src/
book_database.erl, line 36)
     in call from mongo_timmer:cache_timer_write/2 (c:/Users/ashik/workspace/Mong
oDB/src/mongo_timmer.erl, line 49)
3>
```

Figure 6.26: Inconsistent behavior of Mongrel tool

Experiment 6.3.3.3: The overhead of mixing read-write operation.

These experiments are to measure how much overhead is introduced to the system for different percentages of read-write operations. For these experiments, each of the resource's size is 24 bytes. An Erlang client is used to sequentially perform the read and writes operations and measures the request-response time. The experiments are conducted by varying the read-write request ratio between 50% and 100%. The tests are repeated five (5) times, and average, minimum and maximum request-response times for each combination are shown in Table 6.21.

Table 6.21: Request-response time for different percentage of read-write mix

% of Read	Maximum request-response time (ms)	Minimum request-response time (ms)	Average request-response time (ms)
100%	397	387	392.4
90%	388	376	384.18
80%	386	371	379.8
70%	381	375.4	368
60%	369	353	364
50%	359	347	352

6.3.3.3 Results and discussion

Table 6.22: Results comparing request-response time for read and write

Number of read/write	Average request-response time (ms)
50 read operations	200
50 write operations	152

The results in Table 6.22 shows that it takes on an average of 200 milliseconds to perform 50 read operations through the middleware, while 152 milliseconds are required for 50 write operations. The read operation through the middleware has 31.57% latency overhead in comparison to the write operation. The middleware performs better when the number of read requests is high in comparison to the number of write requests, since middleware pre-fetching and caching features are only applicable for read operations. Figure 6.27 show the graph that provides the performance of the middleware for the different percentages of read-write operations.

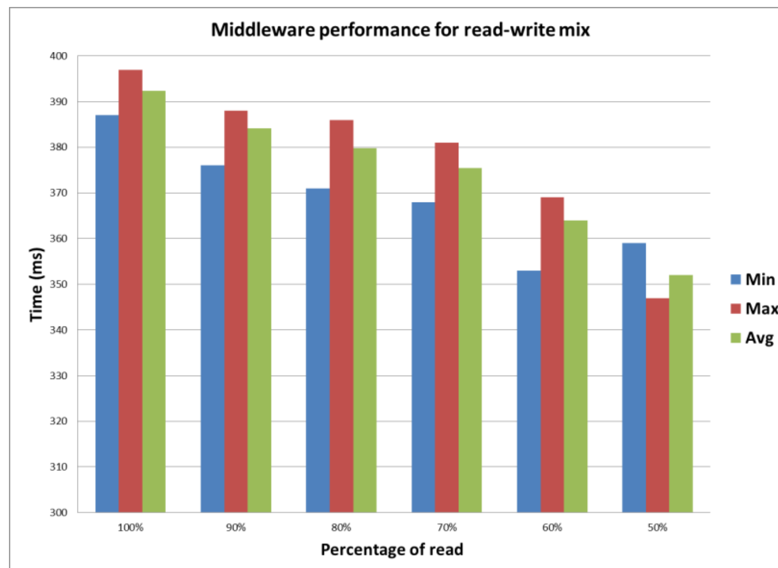

Figure 6.27: Graph performance of the middleware for different percentages of read-write mix

Experiment 6.3.3.4: The overhead of the resource state synchronization by monitoring component in the policy-based middleware.

These experiments are to measure how much overhead is introduced by the middleware to the system for synchronization of resources in the middleware cache. The Erlang client is used to perform the update synchronization of the resources in the middleware cache with the backend cloud hosted server. For these experiments, each resource's size is 73 bytes. The size of the head is 161 for the GET request and 153 bytes for the HEAD request.

6.3.3.4 Results and discussion

I measured the time for state synchronization of 100 resources in the middleware cache. The time includes making an HTTP HEAD request, comparing the ETAG from the request with that in the middleware, a HTTP GET request to the server and pushing the updates in the middleware cache. The tests are repeated five (5) times, and the average state synchronization time is shown in the Table 6.23.

Table 6.23: Overhead of state synchronization by the monitor component

No of resources	Average time for state synchronization (ms)
100	435

The result shows that to synchronize 100 resources, the monitor component introduces 435 milliseconds latency overhead to the middleware.

Experiment 6.3.4: Experiments to evaluate middleware performance in the local cloud.

These experiments evaluate the advantages of hosting the middleware in the local cloud with respect to the distant cloud. For these experiments using Google Nexus7, I measured the request-response time for different number of resources from both the local and the distance cloud. The requests were made sequentially with a JavaScript loop, and the resource size is 73 bytes.

Test 1: For the first test, I kept both the middleware and the mobile client in the same network (Figure 6.28). The middleware was hosted on a machine which belongs to the university network. The mobile was connected to the internet through the university Wi-Fi network.

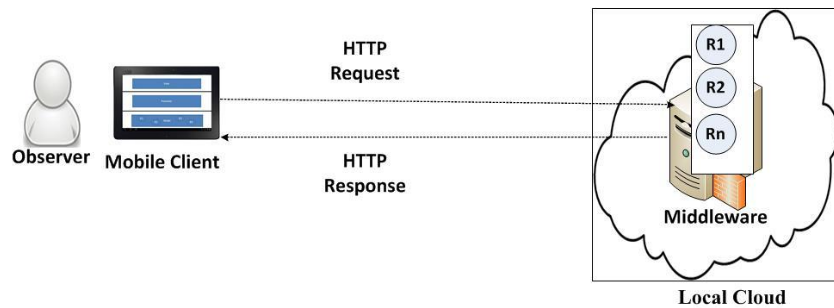

Figure 6.28: Set-up for measuring request-response time to get resources from the local cloud

Table 6.25 shows the average time for getting 1000 resources while both middleware and mobile client are in the same network (10 runs).

Table 6.25: Average request-response time to get resources from the local cloud

No of resources	Average request-response time(ms)
1000	8138.6

Test 2: For the second test, I kept the middleware in the university network (Figure 6.29), while I took the mobile client 15 hops away from the middleware. The Table 6.26 shows the average time for getting 1000 resources while middleware is 15 hubs away from the mobile client (10 runs).

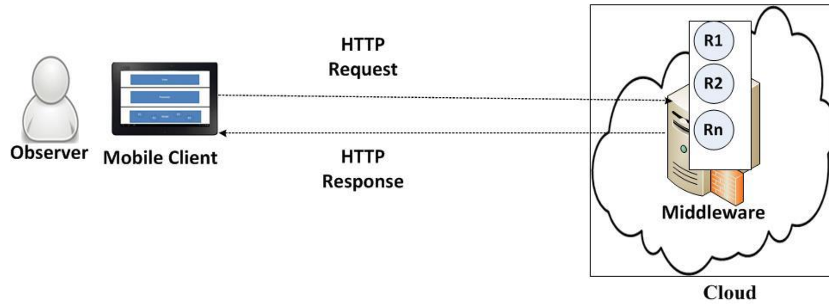

Figure 6.29: Set-up for measuring request-response time to get resources from the distant cloud

Table 6.26: Average request-response time for obtaining resources from the distant cloud

No of resources	Average request-response time(ms)
1000	58840.1

The distance is measured using trace route command in command prompt (Figure 6.30).

```
Tracing route to www.xoxo.usask.ca [67.63.50.58]
over a maximum of 30 hops:

  1     *        *        *     Request timed out.
  2    13 ms    12 ms    10 ms  64.59.177.83
  3   259 ms    12 ms    10 ms  rd1ha-tge2-1.ss.shawcable.net [66.163.73.5]
  4    23 ms    17 ms    20 ms  rc1sc-ge3-1-0.wp.shawcable.net [66.163.76.189]
  5    20 ms    20 ms    19 ms  rc1nr-tge0-0-1-0.wp.shawcable.net [66.163.73.181
]
  6    38 ms    45 ms    47 ms  rc1wh-ge6-0-0-1.vc.shawcable.net [66.163.77.206]

  7    33 ms    37 ms    34 ms  chi-eqx-i1-link.telia.net [62.115.9.77]
  8    33 ms    39 ms    34 ms  chi-bb1-link.telia.net [80.91.246.168]
  9    80 ms    63 ms    74 ms  nyk-bb2-link.telia.net [213.155.136.18]
 10    61 ms    63 ms    63 ms  ash-bb4-link.telia.net [213.155.130.75]
 11    62 ms    63 ms    65 ms  ash-b1-link.telia.net [213.155.130.59]
 12    72 ms    76 ms    71 ms  internap-ic-140171-ash-bb1.c.telia.net [213.248.
95.138]
 13    68 ms    68 ms    64 ms  border2.pc2-bbnet2.wdc002.pnap.net [216.52.127.8
3]
 14    77 ms    67 ms    66 ms  infospaceinc-3.border2.wdc002.pnap.net [64.94.31
.142]
 15    68 ms    70 ms    65 ms  shawcassist-c.infospace.com [67.63.50.58]

Trace complete.
```

Figure 6.30: Measuring number of hops between mobile clients and middleware

6.3.4 Results and discussion

Figure 6.31 show the graph that represents the average latency of resource delivery from the local cloud vs. the distant cloud. The number of requests for resources was increased from 100 to 1000. Each of the tests was repeated 10 times.

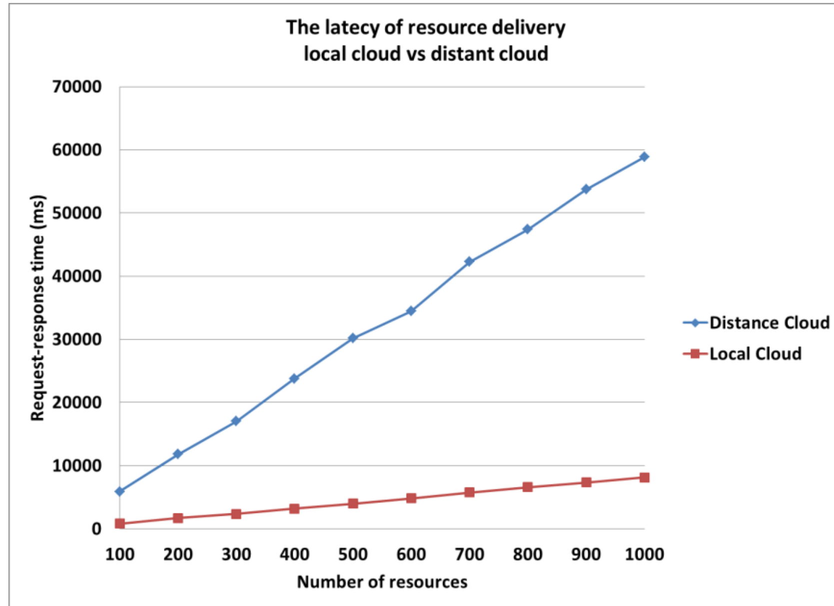

Figure 6.31: The latency graph of resource delivery of the local cloud vs. distant cloud

Table 6.27: Average latency to delivery each resource for the local cloud vs. distant cloud

	Average Request-response Time (ms)
Distance Cloud	59.02
Local Cloud	8.14

The result shows (Table 6.27) hosting the middleware in the local cloud reduces the latency of resource delivery by 7.25 times in comparison to the distant cloud.

6.4 Summary

This chapter describes various experiments that were conducted in order to evaluate the performance of the proposed policy-based middleware architecture and its overheads. Section 6.1 provides an overview of the experiments and their goals. Section 6.2 describes the experimental setup for conducting proposed experiments. Section 6.3 includes the list of experiments that were conducted to evaluate performance and overhead introduced by the proposed middleware architecture. All of these tests performed were employed in simulation in a laboratory environment. The summary of the experiments and outcomes are listed below:

Goal 1: To reduce the latency of the resource delivery and state synchronization on user mobile devices.

Experiment 6.3.1.1: The mobile-end caching and pre-fetching of information in the middleware provides high performance boost. The result shows latency of accessing resources is reduced by 19.17 times in comparison to direct access to cloud hosted services by the mobile devices.

Experiment 6.3.1.2: The user-specific information (i.e., content adaption) to the middleware decreases the amount of data transferred and further facilitates efficient client side service consumption. The content adaptation information in the middleware decreased the latency of resource delivery by 18.54% in contrast to getting data directly from the cloud server. Additionally, content adaptation reduces the data overhead of the mobile client by 3.86 times in terms of sending and 8.14 times in terms of receiving.

Experiment 6.3.1.3: The clients pull increases the latency of state synchronization by 156.32% in comparison to the middleware push. These tests reflect application state based resource synchronization, which reduces data overhead for the resource state synchronization in

the client cache by around 1.67 times, while the performance improvement of the system is 66.67% in terms of latency.

Goal 2: Middleware support for the model synchronization on the multiple devices of a single user.

Experiment 6.3.2: The tests in this experiment verify the support of model synchronization of the middleware to the multiple devices of a single user with different operating systems and configurations. The comparison shows that model (1000 resources) synchronization from the middleware can reduce the latency by 69.30%, in comparison to getting it directly from the Web server. The Google Nexus7 has 1.41 times performance benefit in terms of latency in contrast to ASUS Transformer as the Nexus7 has a more powerful CPU and smaller screen size.

Goal 3: To keep the overhead minimal for using different information in the policy-file of the middleware towards reduce latency of resource delivery and state synchronization.

Experiment 6.3.3.1: The experiments verify that the caching and pre-fetching information in the middleware added latency overhead by 30.49% in comparison to retrieving resources directly from the cloud server. The client will not experience this as the resources are already pre-fetched in the client cache.

Experiment 6.3.3.2: Using DETS table as the middleware cache provides the middleware capability of serving around 4.92 times more requests than the MongoDB. Furthermore, MongoDB interface tool in the Erlang environment has very high overhead.

Experiment 6.3.3.3: The read operation through the middleware has 31.57% latency overhead in comparison to the write operation. The middleware will perform better while the number of read request will be very high in comparison to write, since, the middleware pre-fetching and caching features are only applicable for read operations.

Experiment 6.3.3.4: The result shows that to synchronize 100 resources monitor components introduces 435 milliseconds latency overhead to the middleware.

Goal 4: Evaluate the advantages of hosting middleware in the local cloud.

Experiment 6.3.4: The result shows that hosting the middleware in the local cloud can reduce the latency of resource delivery in comparison to the distant cloud by 7.25 times.

The overall summary of the test results reflects the impact of different information (state transition, caching, pre-fetching and content adaption) in the policy-file of the middleware for efficient delivery and synchronization of cloud hosted services on multiple devices of a single user. In addition, using local cache reduces the cost of communication and compensates for intermittent connections. The policy-based middleware achieves its planned goal of reducing the processing overhead of the mobile devices. Though the proposed approach increases processing overheads of the middleware, this is not a major concern, as hosting the middleware in cloud platforms provides elasticity of using required computational resources.

CHAPTER 7
SUMMARY AND CONTRIBUTION

This research introduced a policy-based middleware, which supports users to access cloud hosted services via apps across multiple devices in a seamless manner. The main contribution of this work is to identify the impact of different information in the policy-file to configure the middleware behavior to address the challenges (e.g., latency, bandwidth limitation and data state synchronization) that mobile clients face to consume services in the wireless environment.

Additionally, the architecture can be made more efficient by bringing services closer to the mobile clients (in the local cloud) to minimize the latency of accessing the cloud hosted services. The Model-View-Presenter (MVP) framework was combined with the architecture for better decoupling among the components in the distributed environment. The middleware provides a RESTful interface for consuming cloud hosted services which are very suitable for mobile clients. Using stub with local cache reduces the communication costs even further and provides offline availability of the application for intermittent connectivity. The push technology allows the middleware to shrink the client side overheads of processing unnecessary information from the server responses.

The middleware is implemented using the Erlang programming environment as the language supports concurrency in distributed environments. All the communication within the components of the middleware takes place asynchronously through message passing which is one of the requirements of making distributed systems scalable. The middleware cache is employed using two leading version of NOSQL databases, namely Erlang DETS and MongoDB. The mobile clients and middleware communicate through the WebSocket protocol to reduce the bandwidth consumption and latency of resource delivery. The mobile application is implemented using HTML5 and PhoneGap, which enabled the building of a hybrid app (i.e. mobile web app that

looks and functions as a native app). The choice of HTML5 and PhoneGap is influenced by the fact that they support single code for multiple platforms.

The advantages of the using the proposed policy-based middleware architecture for delivery and state synchronization of resources are illustrated by experiments conducted on a proof-of-concept prototype. The results show that the caching, pre-fetching and content adaptation information in the policy-file reduces the latency of resources delivery and synchronization in the wireless environment. Additionally, above information helps middleware to reduce device processing overhead along with bandwidth consumption. The performance benefit of hosting the middleware in the local cloud, in comparison to the distant cloud is also evaluated.

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

- This research introduced novel architecture of a policy-based middleware, which can support users to consume cloud hosted services from multiple devices in a seamless manner. The concept of application state based service delivery and synchronization presented in this thesis provides great performance benefit, introducing minimal overhead to the system.
- The experiments on the prototype reflect the impact of different information in the policy-file to enable better support to mobile clients. The experiments also proved the performance benefit of hosting middleware in the local cloud infrastructure in comparison to the distant cloud.
- The adaptation of the MVP pattern by the architecture provides decoupling among the components in distributed environments. The middleware provides a RESTful interface for consuming cloud hosted web services, and is lightweight and suitable for mobile clients.

- Using the local cache on the mobile devices reduces the cost of communication and makes the application available offline during intermittent loss of connectivity.
- The middleware supports user consumption of both SOAP-based and RESTful web services.
- The use of HTML5 and Web technology frameworks (e.g. PhoneGap) in the client side enables the development and the deployment of a hybrid mobile application for different platforms (i.e. BlackBerry, Android, iOS and Windows Phone).

CHAPTER 8
LIMITATIONS AND FUTURE WORKS

One of the main limitations of the framework proposed in this thesis is not considering security issues. Security is one of the major issues towards adoption of cloud platforms by different enterprises [57]. Another issue is how workload can be distributed in the many instances of the middleware for the concurrent clients. The following sections contain details about possible future works.

8.1 Middleware Support for Concurrent Clients and Load Distribution

The proposed middleware architecture is centralized, which provides single points of entry for the clients. The middleware provides good performance with a small number of clients. However, as the number of the clients will increase, it will degrade the quality of services as the middleware may be overloaded. More experiments need to be carried out in the future to prove the resilience and scalability of the middleware with increasing number concurrent clients. The architecture needs to distribute workload among multiples instances of the middleware with growing numbers of concurrent clients. Therefore, the future work will focus on investigating how load can be distrusted among multiple middleware instances and evaluate the system performance in a distributed environment.

8.2 Explore Implications in Different Domains and More Information in the Policy-File

The focus of this research is to provide a generic architecture to support user centric mobile cloud computing. The different features of the middleware are suitable for different applications and contexts. For example, we can consider when the service delivery environment is within an enterprise environment. If the organizations have bigger local computing structure then the use

of local cloud becomes feasible. In contrast, such an application where the connectivity is not very good and needs offline application support, local cache is more important. Another interesting aspect that can be explored is defining policies, which allow users to define workflow engines to carry out context based actions. In the future, the focus will be more about domain specific policy definition and its impacts for large scale enterprise adoption of this architecture.

For content adaptation, this research just considered the information for protocol transformation. More policies can be added in the policy-file of the middleware for content adaptation and one can explore the benefits of such policies. For future research, a goal might be to identify more influential information, for example, device-specific content adaptation, priority-based service delivery, service composition and caching policies.

8.3 Policies for Data Security on the Cloud Platform

The hosting of web services on the cloud platform has security challenges as it is in the multi-tenant environment where the sharing of resources among multiple clients increases the possibility of attacks. Takabi et al. [57] identified preserving privacy of data, authentication of user access and providing of transparency as key challenges towards ensuring a secure cloud computing environment. As a solution of the above mentioned problems, they later proposed a middleware architecture [58] which provides the user with an interface to define policies for providing secure access to their resources that are hosted in the different cloud platforms. Most of the current cloud service platforms have their own access mechanisms to ensure security. For example, the EC2 cloud service provider provides an interface which enables clients to define security rules for their resources [23]. Though this approach ensures the security for customer resources in the cloud, it overloads the customer with extra responsibilities to understand and configure security policies correctly, when required to interact with multiple cloud services.

In the future, I would like to extend the proposed architecture with different security policies in the policy-file of the middleware, to ensure secured resource delivery. The policy-file will contain all the authentication information for different services available for a client. The approach will reduce client side overhead of dealing with an authentication mechanism for accessing services hosted in the different platforms. At the beginning of a client interaction with middleware, the middleware will provide a key. The key will be used as an identifier for the clients. The middleware will perform the authentication with all services in the different cloud platforms according to the policy-file. The same key could be used by the clients when they are accessed from multiple devices. This approach is expected to reduce the client overhead of accessing cloud services significantly.

8.4 Embedding the Middleware with Different Existing Web-frameworks

The next target is to provide the middleware as an application package which will provide an easy way of installation and use in an application domain. The most important characteristics that must be considered for tool development are that it should be easy to deploy, and the application development process must be rapid. Further, the focus is to integrate the middleware with different existing web frameworks (e.g., Node.js, jQuery Mobile, .NET platform, Sencha Touch [53]). This approach will make it easy for developers to automatically define and change different information in the policy-file. Additionally, it must enable developers to develop and deploy the hybrid applications without introducing much overhead.

LIST OF REFERENCES

[1] Armbrust, M., et al. (2010)."A view of cloud computing", Communications of the ACM 53.4 (2010): pp. 50-58.

[2] Armstrong, J. (2007). "A history of Erlang", In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). ACM, New York, NY, USA, pp. 1-26.

[3] Balan, R. K., Gergle, D., Satyanarayanan, M., Herbsleb, J. (2007). "Simplifying cyber foraging for mobile devices", In Proceedings of the 5th international conference on Mobile systems, applications and services, ACM, pp. 272-285.

[4] Beccue, M. (2009). ABI research report RR-MCC: "Mobile Cloud Computing", pp. 1-64.

[5] Christensen, J. H. "Using RESTful web-services and cloud computing to create next generation mobile applications", Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2009, Orlando, Florida, USA, pp. 627-633.

[6] Chun, B. G., Ihm, S., Maniatis, P., Naik, M. (2010). "Clonecloud: boosting mobile device applications through cloud clone execution", arXiv preprint arXiv: 1009.3088, pp. 1-14.

[7] "Cloud Computing Subscribers to Total Nearly One Billion by 2014", Last accessed: October 7, 2012.

http://www.directionsmag.com/pressreleases/mobile-cloud-computing-subscribers-to-total-nearly-one-billion-by-2014/119248Mobile

[8] Cook, W. R., Barfield, J. (2006). "Web services versus distributed objects: A case study of performance and interface design".,In Web Services, 2006. ICWS'06. International Conference on IEEE, pp. 419-426.

[9] Cortez, R., Rajam, S., Vazhenin, A., Bhalla, S. (2010). "E-Learning Distributed Cloud Built on MVC Design Patterns for Service Task Management", In Proceedings of the Euro-American Conference on Telematics and Information Systems, pp. 22-24.

[10] Cox, P.A. (2011). "Mobile Cloud Computing Devices, trends, issues, and the enabling technologies", Last accessed: October 7, 2012

http://www.ibm.com/developerworks/cloud/library/cl-mobilecloudcomputing/

[11] Cuervo, E., Balasubramanian, A., Cho, D., K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P. (2010). "MAUI: making smartphones last longer with code offload", In Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10). ACM, New York, NY, USA, pp. 49-62.

[12] Farley, P., Capp, M. (2005). "Mobile Web Services", BT Technology Journal Vol. 23, pp. 202-13.

[13] Fernandez J., Fernandez A., Pazos J. (2005). "Optimizing Web Services Performance Using Caching", Proceedings of the International Conference on Next Generation Web Services Practices, pp. 1-6.

[14] Feuerlicht, G., Govardhan, S. (2009). "SOA: Trends and Directions", Systems Integration, pp. 149-155.

[15] Fielding, R. (2000). "Architectural Styles and the Design of Network-based Software Architectures", University of California, pp. 1-180.

[16] Flores, H., Srirama, S. N., Paniagua, C. (2011). "A generic middleware framework for handling process intensive hybrid cloud services from mobiles", In Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia ACM, pp. 87-94.

[17] Fowler, M. (2006). "GUI Architectures", Last accessed: July 19, 2012. http://www.martinfowler.com/eaaDev/uiArchs.html

[18] Gartner, Inc. (2012) "Gartner's Roadmap to SOA", Last accessed: June 27, 2012. http://ebreez.com.au/Documents/Gartner%20Roadmap%20to%20SOA.pdf

[19] Gehlen, G., Mavromatis, G. (2005). “Mobile Web Service based Middleware for Context-Aware Applications”, in Proceedings of the 11th European Wireless Conference 2005, vol. 2. Nicosia, Cyprus: VDE Verlag, pp. 784-790.

[20] Gomaa H., Messier G., Davies R., & Williamson, C. (2009). “Media caching support for mobile transit clients”, Wireless and Mobile Computing, Networking and Communications, 2009. WIMOB 2009. IEEE International Conference on [0-7695-3841-X], 2009, pp. 79 -84.

[21] Hornsby, A. (2011, January). “XMPP message-based MVC architecture for event-driven real-time interactive applications”, In Consumer Electronics (ICCE), 2011 IEEE International Conference on IEEE, pp. 617-618.

[22] Huang, D. (2011). “Mobile Cloud Computing”. IEEE COMSOC Multimedia Communications Technical Committee (MMTC) E-Letter, Vol. 6, No. 10, pp. 27-31.

[23] Jaeger, T., Schiffman, J. (2010). “Outlook: Cloudy with a chance of security challenges and improvements. Security & Privacy”, IEEE, 8(1), pp. 77-80.

[24] Jamal, S., Deters, R. (2011). “Using a cloud-hosted proxy to support mobile consumers of RESTful services”, Procedia Computer Science, 5, pp. 625-632.

[25] Joshi, K. P., Finin, T., Yesha, Y., Joshi, A., Golpayegani, N., Adam, N. (2012). “A Policy-based Approach to Smart Cloud Services”, In Proceedings of the Annual Service Research and Innovation Institute Global Conference, pp. 1-10.

[26] Jquery Mobile Last accessed: December 05, 2012. http://jquerymobile.com/

[27] Kaungas P, Dumas M. (2011). “Configurable SOAP proxy cache for data provisioning web services”. In Proc. the 2011 ACM Symposium on Applied Computing (SAC2011), Taiwan, China, May 21-24, 2011, pp. 1614-1621.

[28] Kazi, A., Kazi, R., Deters, R. (2012). “Supporting the Personal Cloud”, IEEE Asia Pacific Cloud Computing Congress 2012, pp.1-6.

[29] Kemp, R., Palmer, N., Kielmann, T., Bal, H. (2012). “Cuckoo: a computation offloading framework for smartphones”, Mobile Computing, Applications, and Services, pp. 59-79.

[30] Kosta, S., Aucinas, A., Hui, P., Mortier, R., & Zhang, X. (2011). “Unleashing the Power of Mobile Cloud Computing using ThinkAir”, arXiv preprint arXiv: 1105.3232, pp. 1-17.

[31] Kovachev, D., Cao, Y., Klamma, R. (2012). “Augmenting pervasive environments with an XMPP-based mobile cloud middleware”. Mobile Computing, Applications, and Services, pp. 361-372.

[32] Kovachev, D., Cao, Y., Klamma, R. (2011). “Mobile cloud computing: a comparison of application models”, arXiv preprint arXiv: 1107.4940, pp. 1-8.

[33] Kozuch, M., Satyanarayanan, M. (2002). “Internet suspend/resume”, In Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on, pp. 40-46.

[34] Kristensen, D., M. (2008).” Execution plans for cyber foraging. In Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device”, (MobMid '08). ACM, NY, USA, pp. 2-6.

[35] La, H. J., Kim, S. D. (2010). “Balanced MVC Architecture for Developing Service-based Mobile Applications”, In e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference, pp. 292-299.

[36] Lassila, O. (2005). “Applying Semantic Web in Mobile and Ubiquitous Computing: Will Policy-Awareness Help?” In L. Kagal, T. Finin, and J. Hendler, editors, Proceedings of the Semantic Web Policy Workshop, 4th International Semantic Web Conference, pp. 6-11.

[37] Lee, S., Lee, K. W., Ryu, K. D., Choi, J. D., & Verma, D. (2006). “Deployment-time Binding Selection to Improve the Performance of Distributed Applications”, IBM research report, pp. 1-19.

[38] Liu, X., Deters, R. (2007). “An efficient dual caching strategy for web service-enabled PDAs”, SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA: ACM, 2007, pp. 788–794.

[39] Luthria, H., Rabhi, F. (2009). "Service oriented computing in practice: an agenda for research into the factors influencing the organizational adoption of service oriented architectures", Journal of Theoretical and Applied Electronic Commerce Research Archive Volume 4 Issue 1, pp. 39-56.

[40] Mei, C., Shimek, J., Wang, C., Chandra, A., Weissman, J. (2011). "Dynamic Outsourcing Mobile Computation to the Cloud", Technical Report Department of Computer Science and Engineering, University of Minnesota, pp. 1-11.

[41] "Mobile Cloud Computing will skyrocket in 2012", Last accessed: July 11, 2012.
http://cloudcomputingtopics.com/2012/01/mobile-cloud-computing-will-skyrocket-in-2012/

[42] mongoDB (2013). Last accessed: April 26, 2013. http://www.mongodb.org/

[43] "MVC –Model View Controller", Last accessed: July 19, 2012.
http://molecularsciences.org/zend/mvc_model_view_controller

[44] Natchetoi, Y., Kaufman, V., Shapiro, A. (2008). "Service-oriented architecture for mobile applications", In Proceedings of the 1st international workshop on Software architectures and mobility (SAM '08). ACM, New York, NY, USA, pp. 27-32.

[45] Oracle. (2012). "Stub and Skeletons", Last accessed: September 16, 2012.
http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmi-arch2.html

[46] Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y.-M. (2011). "Enabling Automatic Offloading of Resource-Intensive Smartphone Applications", Tech. rep, pp. 1-18.

[47] PhoneGap (2012). Last accessed: April 28, 2013. http://phonegap.com/

[48] Raines, G. (2009). "Cloud Computing and SOA" October 2009. The MITRE Corporation Last accessed: June 28, 2012. pp. 1-12,
http://www.mitre.org/work/tech_papers/tech_papers_09/09_0743

[49] Rodriguez, A. (2010). "Restful Web services: The basics", Last accessed: June 28, 2012.
http://www.ibm.com/developerworks/webservices/library/ws-restful/

[50] Satyanarayanan, M. (2001). "Pervasive computing: vision and challenges", Personal Communications, IEEE, vol.8, no.4, pp. 10-17.

[51] Satyanarayanan, M., Bahl, P., Caceres, R. and Davies, N. (2009). "The Case for VM-based Cloudlets in Mobile Computing", Journal IEEE Pervasive Computing Volume 8 Issue 4, pp. 14-23.

[52] Selonen, P., Belimpasakis, P., You, Y., Pylvänäinen, T., Uusitalo, S. (2012). "Mixed Reality Web Service Platform", Multimedia Systems, Springer Berlin / Heidelberg Volume 18 Issue 3, pp. 215-230.

[53] Sencha (2012). Last accessed: April 28, 2013. http://www.sencha.com/products/touch

[54] Smaldone, S., Gilbert, B., Bila, N., Iftode, L., de Lara, E., & Satyanarayanan, M. (2009, June). "Leveraging smart phones to reduce mobility footprints", InProceedings of the 7th international conference on Mobile systems, applications, and services, ACM, pp. 109-122.

[55] Song, H., Choi, E., Bae, C. S., Lee, J. W. (2011). Web Based Application Program Management Framework in Multi-Device Environments for Personal Cloud Computing, In IT Convergence and Services, Springer Netherlands. pp. 529-536.

[56] Stirbu, V. (2010). "A RESTful Architecture for Adaptive and Multi-device Application Sharing", ACM International Conference Proceeding Series, Proceedings of the 1st International Workshop on RESTful Design, WS-REST 2010; Raleigh, NC, USA. pp. 62-65.

[57] Takabi, H., Joshi, J. B., Ahn, G. J. "Security and Privacy Challenges in Cloud Computing Environments", IEEE Security and Privacy, Vol. 8, No. 6, pp. 25-31.

[58] Takabi, H., Joshi, J. B. (2012). "Policy management as a service: an approach to manage policy heterogeneity in cloud computing environment", In System Science (HICSS), 2012 45th Hawaii International Conference on, pp. 5500-5508.

[59] Tang, W., Lee, J. H., Song, B., Islam, M., Na, S., & Huh, E. N. (2011). "Multi-Platform Mobile Thin Client Architecture in Cloud Environment", Procedia Environmental Sciences, 11, pp. 499-504.

[60] Wang, Q. (2011). "Mobile Cloud Computing", M.Sc. Thesis Submitted to the College of Graduate Studies and Research, Department of Computer Science, University of Saskatchewan, Saskatoon, Canada. January, pp.

1-91.

[61] WebSocket Protocol (2011). Last accessed: December 08, 2012. http://tools.ietf.org/html/rfc6455

[62] Web SQL Database (2012). Last accessed: December 05, 2012. http://dev.w3.org/html5/webdatabase/

[63] Wikipedia (2012). "Web Service", Last accessed: June 26, 2012. http://en.wikipedia.org/wiki/Web_service

[64] Williamson, A. (2010). "Amazon EC2 Latency", Last accessed: June 20, 2012.

http://alan.blog-city.com/amazon_ec2_latency_the_pretty_graphs.htm

[65] Yaws (2013). Last accessed: June 20, 2013. http://yaws.hyber.org/

[66] Zhang, X., Deters, R. (2012, November). MVC apps in the personal cloud. In Cloud Computing and Communications (LATINCLOUD), 2012 IEEE Latin America Conference, pp. 7-12.

[67] Zhang, X., Kunjithapatham, A., Jeong, S., Gibbs, S. (2011). "Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing", Mobile Networks and Applications, 16(3), 270-284.

[68] Zhao, B., Xu, Z., Chi, C., Zhu, S., Cao, G. (2012). "Mirroring smartphones for good: A feasibility study", In Mobile and Ubiquitous Systems: Computing, Networking, and Services, Springer, pp. 26-38.

[69] Zulkernine, F., Martin, P., Craddock0F1, C., and Wilson1F2, K., (2008). "A Policy-based Middleware for Web Services SLA Negotiation". In Proceedings of the 2HIEEEInternational Conference on Web Services (ICWS'08), IEEE Press, pp. 1-8.