ENABLING MOBILE DEVICES TO HOST CONSUMERS AND PROVIDERS OF RESTFUL WEB SERVICES

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

RICHARD KWADZO LOMOTEY

Keywords: mobile Web resources hosting, middleware, cloud computing, REST

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

The strong growth in the use of mobile devices such as smartphones and tablets in Enterprise Information Systems has led to growing research in the area of mobile Web services. Web services are applications that are developed based on network standards such as Services Oriented Architecture and Representational State Transfer (REST). The mobile research community mostly focused on facilitating the mobile devices as client consumers especially in heterogeneous Web services. However, with the advancement in mobile device capabilities in terms of processing power and storage, this thesis seeks to utilize these devices as hosts of REST Web services.

In order to host services on mobile devices, some key challenges have to be addressed. Since data and services accessibility is facilitated by the mobile devices which communicate via unstable wireless networks, the challenges of network latency and synchronization of data (i.e. the Web resources) among the mobile participants must be addressed.

To address these challenges, this thesis proposes a cloud-based middleware that enables reliable communication between the mobile hosts in unreliable Wi-Fi networks. The middleware employs techniques such as message routing and Web resources state changes detection in order to push data to the mobile participants in real time. Additionally, to ensure high availability of data, the proposed middleware has a cache component which stores the replicas of the mobile hosts' Web resources. As a result, in case a mobile host is disconnected, the Web resources of the host can be accessed on the middleware. The key contributions of this thesis are the identification of mobile devices as hosts of RESTful Web services and the implementation of middleware frameworks that support mobile communication in unreliable networks.

ACKNOWLEDGEMENTS

First of all I would like to express my genuine thanks to my supervisor, Dr. Ralph Deters for his advice, intuitive criticisms, patience, and financial assistance throughout my entire duration of study. Also, my sincere thanks to the members of my advisory committee: Dr. Julita Vassileva, Dr. John Cooke, and Dr. Ha H. Nguyen for their valuable advices and insightful suggestions. Furthermore, I would like to thank Mr. Andrew Kostiuk and TRLabs for their support. Also, I would like to appreciate Ms. Jan Thompson and Ms. Gwen Lancaster, at the Department of Computer Science, University of Saskatchewan, who have been very helpful and kind throughout my study.

Thanks to all students of the Multi-Agent Distributed Mobile and Ubiquitous Computing (MADMUC) Lab for their friendship.

Finally, I would like to thank my wife, Emelia Lomotey; my mother, Doris Dartey; and my entire family for the unconditional love and generosity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
BASE	Basically Available Soft-state Eventual consistency
CAP	Consistency Availability and Partition-tolerance
CREST	Computational REST
EC2	Amazon Elastic Cloud Computing
EIS	Enterprise Information Systems
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
OS	Operating System
REST	Representational State Transfer
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WS	Web Services
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition

CHAPTER 1
INTRODUCTION

1.1 Background

Mobile devices such as smartphones and tablets have not only become part of us at homes and offices, but are also shaping how enterprise businesses and transactions are being carried out [1]. These devices are gaining widespread usage in Enterprise Information Systems (EIS) [2] - which are application-oriented infrastructures that manage information in organizations. Also, since Web services [3], that is network and Web centric applications, are platform independent and ensure easy interoperability between system components as reported by Beal [4], enterprises are looking more at modeling data and information as Web services.

However, in most mobile distributed systems, the mobile devices are employed as client consumers of heterogeneous Web services. Hence, the mobile node is used to render data that is hosted on back-end components such as servers. Conversely, with the increasing mobile processing power and advancement in storage space on these devices, our research aims at facilitating the mobile devices as a Web services hosting nodes. In the case of mobile hosting, the mobile devices become Web services provider nodes, thereby allowing other mobile participants to access Web resources that reside on the mobile hosts. Some examples of Enterprise Information Systems where our research is applicable are University Administration Application Systems, Insurance Sales Applications, and mission critical systems such as E-Health [5], [6], [7] - where data is expected to be accessed over a secure wireless network for health care delivery.

Though Web services can be built using the Service-Oriented Architecture (SOA) model [8], the model does not lend itself well in mobile distributed environment due to the constraints on network bandwidth in mobile networks. This is largely because SOA passes huge XML data

across system components [9], [10]. Thus, this thesis puts forward the use of the Representational State Transfer (REST) [11], [12] approach, which is a lightweight protocol that can be invoked over HTTP.

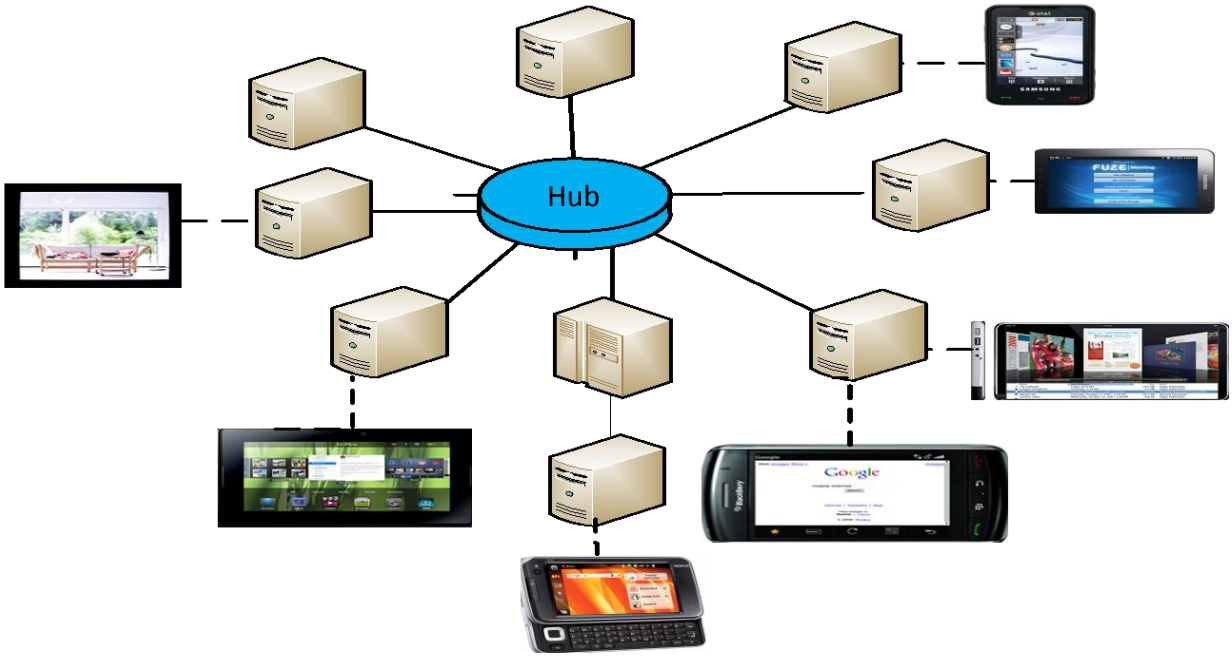


Figure 1.1: Distributed Mobile Environment

Mobile hosts use wireless channels to communicate within system components in a distributed mobile environment as shown above in Figure 1.1. Mobile service providers in Canada such as Fido, Rogers, and SaskTel offer their customers data plans for fees that enable mobile device users to access backend systems and download contents. In addition, mobile devices allow users to connect to hotspots via Wi-Fi connectivity. Thus, data in multimedia formats can be shared between smartphones, tablets and notebooks. Also, each of these devices has an embedded browser that enables the deployment of web-based applications on them [10].

However, in mobile distributed systems, connectivity is not guaranteed due to the intermittent loss of connectivity in wireless networks. Users can find themselves in spots within

buildings where there is no Wi-Fi connectivity. Disconnections can also be attributed to end-user mobility and unstable networks from the service provider. In view of the intermittent losses in connectivity the challenges of network latency and data synchronization among the mobile participants have to be addressed.

1.2 Challenges in Mobile Web Hosting Environment

Enterprise Information Systems have been using desktops in reliable wired environments in one central location. Now, enterprise services have been extended to mobile devices in unreliable network environments [1], [5]. Key challenges faced in this new paradigm which our research focuses on are: synchronization of Web resources updates (i.e. resources state changes) and reducing network latency that arises from resources propagation. Figure 1.2 shows the human experts and the various client devices that are under consideration in this work.

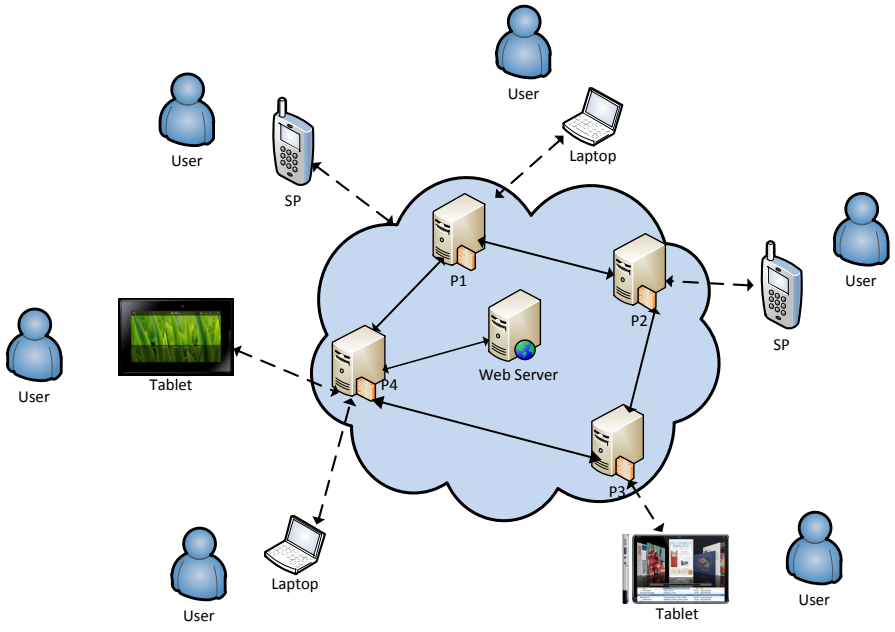


Figure 1.2: Mobile participants in wireless networks

The smartphones (SP) and tablets connect via data plans (e.g. Fido, SaskTel and so on) or Wi-Fi (e.g. 802.11g) to proxies (P1, P2, P3 and P4) to overcome problems related to fluctuating bandwidth. To achieve this, some researchers employ proxies, that are intermediary servers, with functionalities such as caching, event handling, and message routing to communicate with the client devices [13], [14], [15], [16]. Furthermore, cloud computing, which is a network infrastructure that supports mobile devices to access services in the cloud via the Internet, can be employed to lift the heavy-workload on the mobile client [10].

Also, users on a data plan have to pay based on the amount of data downloaded; so our research is recommending the use of Wi-Fi connection. However, in a Wi-Fi environment, connectivity is unstable due to blackout zones in buildings, and there are times when there are interferences and fluctuations due to the large number of users of the network at the same time. In addition, though there has been an improvement in memory size of mobile devices recently, the mobile capacity is still not as good as the modern desktops [17].

1.2.1 Network Latency

Updates in mobile distributed systems can be slow because it takes a certain time for a request-response to be successful in a bi-direction. Figure 1.3 shows the times ***t1, t2*** and ***t3*** taken to propagate a Web resource, *R*, on the tablet device of user 1 to the smartphone device of user 2 through the proxy. The total time is $t = t1 + t2 + t3$.

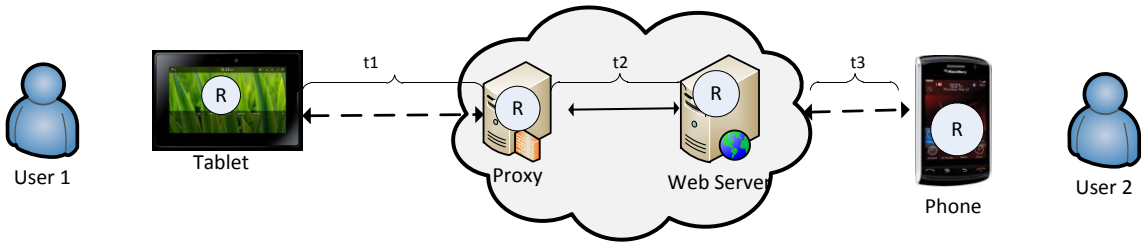


Figure 1.3: Total time for a push request

Latency can be caused by intermittent loss in connectivity or bandwidth fluctuation due to user mobility and limitations on geographical boundaries for Wi-Fi access. As shown in Figure 1.4, proxies which are hosted in the cloud are used to host resources and provide services to mobile clients but there is still update delay due to network loss.

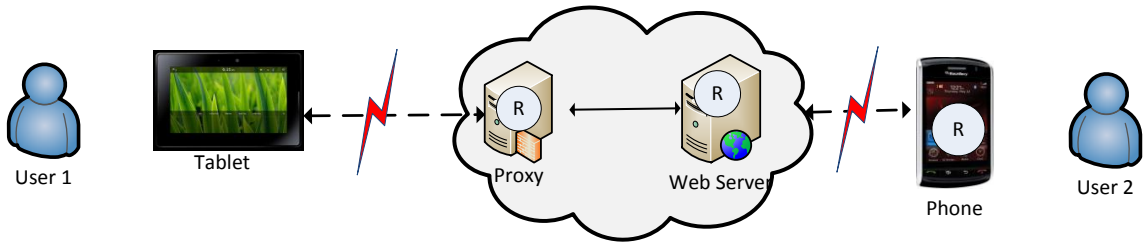


Figure 1.4: Unstable Wi-Fi connectivity

1.2.2 Resource State Synchronization

Due to network latency, updates that arrive on the proxy will not be seen immediately on the mobile hosts. The potential loss of connectivity (network partition failure) between mobile components also means the data in cache can be outdated on the mobile host at certain times. In order to ensure high data consistency on the mobile host and the backend means the availability of the system will be compromised. This is the situation that is described in the CAP theorem [18], [19], [20], [21]. Furthermore, there can be a cascading effect where an update applied to a data (or Web resources) can lead other Web resources to issue state changes. Thus, there is a need to push updated messages to the mobile client as they arrive on the proxy in real-time.

Moreover, a mobile client needs to be able to handle the asynchronous state change messages and update its own services or resources. The identification of which mobile consumers/providers have the latest updates of a resource and which resources need to be updated can be unsuccessful due to network latency. As a result, the REST-WS providers cannot send updated replicas to the appropriate consumers if there is network loss. Additionally, race

conditions arise with read and write requests from the mobile clients. This situation arises when read and write requests are issued concurrently for the same REST Web resources on the proxy or Web server.

Also, *events* such as connection of a mobile device, disconnection of a mobile device, and the resource state change; make synchronization of updates challenging. In most cases, clients register callbacks with proxies in order to identify who has the latest copy of an update but this becomes complex in *composite web services* (see appendix).

1.3 Conclusion

To address the challenges of network latency and Web resources state synchronization, our research is looking into building a lightweight middleware framework that supports the mobile hosts of REST Web services. Key issues in the development of the framework which will be discussed in the literature review to know the current state-of-the-art approaches towards addressing them are:

- Transient connection
- Consuming REST-WS on mobile devices
- Caching and event handling in distributed systems
- Middleware oriented approaches in distributed systems
- Approaches and techniques for dealing with state changes in physically distributed systems

The remaining sections of the thesis are structured as follows: Chapter 2 explores the related works within the mobile Web services domain. Chapter 3 presents our proposed mobile hosting architecture that addresses the challenges of latency and data synchronization. A prototypic application is implemented to test the feasibility of the proposed architecture and the

implementation details are described in Chapter 4. The evaluation of the implemented architecture is discussed in Chapter 5. Chapter 6 and Chapter 7 focus on the conclusion and future works respectively.

CHAPTER 2
LITERATURE REVIEW

In order to understand mobile platforms and which data formats can be efficiently consumed by mobile devices, this chapter focuses on Web services architecture and designs, and mobile REST Web services. Also, overcoming the problems of having unreliable data on mobile client nodes as a result of transient connectivity inspired our work to research on the CAP theorem. To further build a middleware platform that supports mobile hosts and providers of REST services, the following areas are studied in this thesis: cloud computing applications, middleware support for mobility, resource hosting on mobile clients and middleware, and service hosting and workflow on mobile clients.

2.1 The CAP Theorem

Web services running in distributed environments are expected to provide support for data *Consistency*, high system *Availability*, and *Partition-tolerance* to faults. However, Eric Brewer [18] in an invited talk in 2000, made an unproven proposition that; no distributed system can guarantee consistency, availability, and partition-tolerance at the same time. Brewer [18] again noted that two of the three requirements can be guaranteed simultaneously if one of the requirements can be traded-off [18], [19]. Gilbert et al. [19] used a formal model to prove Brewer's conjecture into a theorem. Figure 2.1 illustrates the possible options in the CAP theorem.

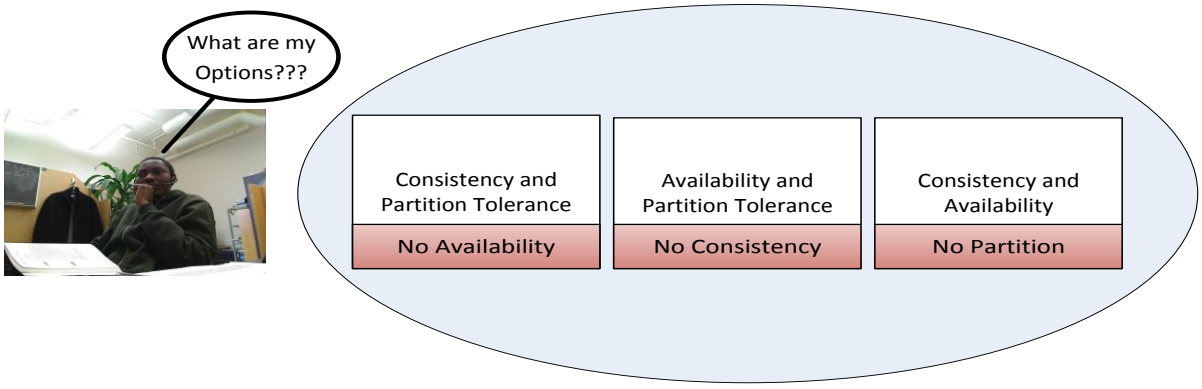


Figure 2.1: Options provided in the CAP theorem

Consistency is the requirement which guarantees that hard state stored in a distributed system is seen the same at every node by clients [20]. Gilbert et al. [19] described consistency as "atomic" which means every system transaction is one and must be completed fully or not started at all.

Availability ensures that when parts of the nodes in a distributed system become inaccessible as a result of failures, the other nodes should continue to operate [20]. It is important that intended responses are received for each request even if other parts of the system fail [22]. Gilbert et al. [19] report that systems tend to fail during peak performance, therefore making availability very difficult to achieve when it is needed most. Dean [23] also noted that though availability is very difficult to achieve due to unpredictability of system faults, it is still the most important requirement for any distributed system.

Partition-tolerance is achieved when a distributed system is built to "allow arbitrarily loss of messages sent from one node to another" [19]. The current demand from Web services consumers makes it impractical to keep all data at one source. This is because when the source fails, it means the entire system becomes unavailable. Partition-tolerance therefore allows for system states to be kept in different locations.

In mobile distributed systems, the CAP theorem phenomenon exists. The intermittent loss of connectivity in wireless networks makes partition tolerance a given. Developers are therefore faced with the option to choose between consistency and availability. Browne [21] commented that the pair of guarantees normally preferred by users is availability and partition-tolerance while trading-off consistency. The quest to build scalable and reliable heterogeneous web services that can overcome the challenges of the CAP theorem led to the development of ACID [19] and BASE [22].

2.1.1 ACID

ACID (Atomicity, Consistency, Isolation, and Durability) is a set of properties that developers follow to overcome mostly the issues of consistency and ensure reliability for distributed transactions [22]. *Atomicity* guarantees that transactions which will require a sequence of steps must be treated as one unit. If a part fails, then the entire transaction fails and the system must be restored to its original state [19]. *Consistency* property ensures that anytime a client sends a request to the database, it receives a valid and desirable data. According to Pritchett [22], "the database will be in a consistent state when the transaction begins and ends". The data received will be viewed to be the same on all requesting nodes at the same time. *Isolation* ensures that a transaction being processed is hidden from other processes until it comes to completion [22]. *Durability* means after a successful transaction, the operation cannot be rolled back even if there are system failures [22].

Distributed systems normally rely on the two-phase commit protocol [22] to provide ACID capabilities. Pritchett [22] explains two-phase commit as follows - if all databases agree on an operation (i.e. precommit) to the coordinator, then the operation can continue with the coordinator in the second phase asking the operations to give their consent but if one database

reports failure, the entire system must be rolled back. This guarantees consistency in the distributed system. Since most systems are horizontally scaled, it means partition-tolerance is ensured which means systems cannot be available according to CAP theorem. Pritchett [22] also pointed out that because of the use of database keys, distributed transactions are highly coupled. It is also impractical to build long-lived transactions in ACID if it uses two-phase commit. Instead, some developers came up with the idea of breaking down a long-running transaction into smaller transactions; a developmental approach known as SAGAS [24], [25].

2.1.2 SAGAS

SAGAS is not an acronym but rather a term coined by Garcia-Molina et al. [25] that describes long running transactions. Long running transactions in a mobile distributed environment is impractical to follow ACID principles because ACID normally uses locks and serialization [24]. Young [24] explains SAGAS in his article that long-lived transactions can be divided into short-lived transactions with linkages known as compensation handlers. The individual short-lived transactions can have ACID properties but not the entire SAGAS.

This approach is good for handling error in the system because unsuccessful short-lived transaction can be rolled back. SAGAS just like ACID also uses a controller to complete or terminate a transaction [24].

2.1.3 BASE

The developers who opt for the option of availability employ the Basically Available Soft-state Eventual consistency (BASE) model. BASE is used to minimize the level of coupling in ACID systems by ensuring availability over consistency [22], [26]. This process ignores two-phase commit and ensures latency. However, eventual consistency is achieved if the system allows for a time lag between operations [22].

Vogels [26] used the concept of replicas that can be achieved on hosts to configure the CAP theorem in distributed systems. Considering $\mathbf{N_h}$ hosts that store replicas, $\mathbf{N_w}$ successful writes to a replica, and $\mathbf{N_r}$ replicas that can be read from hosts, then $\mathbf{N_h >= N_w + N_r}$ ensures eventual consistency and partition tolerance within the distributed system. This means that outdated replicas can be seen on some hosts when a read operation is invoked. In cases where a host can also read from multiple replicas, it becomes difficult to "define a clear semantic about the return value" [27]. A proposed solution to overcome this challenge is the implementation of *client-centric consistency* [27]. This consistency model presents consistency of replica only in the view of a single client [27].

Also, Vogels [26] described the eventual consistency approach as a form of weak consistency and mentioned factors such as delays in communication, the demand on the system, and the number of data replications that the system has to produce as basis for determining an inconsistency window. Key eventual consistency models that are discussed by Vogels [26] are reported below.

- ***Causal consistency*:** This model assumes that if a replica is updated by a process X and duly notifies process Y about the update, then all read operations of Y should return the updated replica [26]. All write operations will equally override any existing write [26]. This is explained by Menascé et al. [28] that if:

  replica1 and replica2 are caches that contain resource A and B respectively;

  Process X => remove resource A and add resource C to replica1;

  Process Y => read replica1 and add resource D;

  Process Z => read replica2;

Then process X and Y have a causal relationship because process Y will see only resource C in *replica1* and return the current state of *replica1* to C, D. However, process Z is concurrent with both processes X and Y because they have no relationship. For a replica to be causally consistent, then the following condition must be met: "Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines" [28]. The difficulty in the implementation of this consistency model is how to identify which processes have causal relationships [27].

- ***Read-your-writes consistency*:** This is a special case of causal consistency where after process X has updated a replica, durability is ensured where by the older version of the replica cannot be read again but only the updated version [26]. From the parameters used above to explain Vogel's concept of replicas, if $\mathbf{N_r + N_w > N_h}$, then read-your-writes consistency is guaranteed [29]. This consistency model directs read requests to only hosts that have updated replicas and all requests to hosts that are yet to update their replicas are not served [30].
- ***Session consistency*:** Read-your-writes consistency is implemented to work only within a context of a session [26]. This is very practical because every session is guaranteed read-your-write consistency but when a session fails, a new session has to be started with no guarantee of consistency from the previous session [26].

Golding [31] proposes a weak-consistency architecture that employs session consistency and timestamps to propagate updates in wide-area systems. All processes send write request with a timestamp that is logged in storage. Each log, *L*, is a triple: *L = {list of sender id, timestamp, message}*. Each process also has a summary record of list of process id and timestamps in order to determine which updates they have read. This ensures that processes only have to exchange

session messages with each other that a partner process does not have. All processes also keep a second record that is described as acknowledgement storage. The list of process ids and timestamps are kept here in order to know which updates have been acknowledged by other partners. By this, the system ensures reliable synchronous message exchanges among components and enables replicas to be independent. All communications are logged by a partner replica within a group so the moment a disconnected replica reconnects, messages are delivered to it. Golding [31] also noted that the use of timestamps in message exchanges is good for message delivery orderings.

2.1.4 Summary

In distributed mobile networks, transactions cannot be guaranteed to follow the strict ACID module due to the asynchronous nature of passing messages between mobile clients in a Wi-Fi network [32]. This is because network latency and intermittent loss of connectivity can cause responses to requests to delay. The fact that ACID uses two-phase commit in distributed systems also makes it ineffective to build long-running mobile transactions since locking will be used [24].

However, mobile transactions can be successfully deployed if ACID can be compromised [32]. BASE lends itself well into mobile web services because allowing for an inconsistency window can cause for eventual consistency in cases of network loss [22], [32]. Furthermore, BASE ensures high system availability in mobile heterogeneous web services even though the system will at a point register inconsistency in replicas [22], [26]. Also, the various eventual consistency models can be combined in a single implementation in order to achieve high scalability and fault-tolerance [26].

2.2 Web Services

Web Services (WS) [3] are network oriented applications that serve information based on standards such as SOAP and REST. WS can be deployed as XML standards when built on SOAP technology and are accessed through a URI (Universal Resource Identifier) address [33]. Also, the development of Web services with standards such as SOAP, WSDL, UDDI and XSD ensures data availability and access at real-time [3].

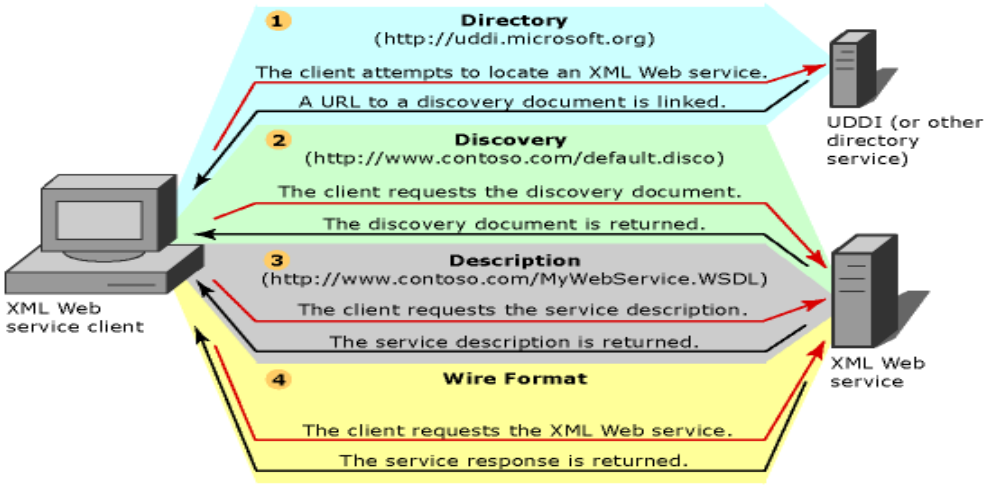


Figure 2.2: Web Services built on XML Structure [34]

As shown in Figure 2.2 above, WS can be exchanged between a Web service client and Web service in the following scenario.

1. The Directory is a location that stores XML based Web services using UDDI as a directory service [34].
2. URIs are used by the client to discover the XML based Web services [34].
3. The client uses Web Service Descriptions as a proxy to interpret what interactions the Web services support. There will be no communication if the XML Web services do not support the XML WS client needs [34].

4. The wire format provides a global protocol support (e.g. SOAP) for XML WS to be communicated with by many platforms [34].

Furthermore, Web services can be used to deploy business services and applications on many platforms since WS mostly use HTTP as a communication protocol [3]. Web services can be implemented with the following styles RPC, SOA and REST [3]. Pautasso et al. [9] compared WS-* which they described as "Big" Web Services, with RESTful Web Services. Their paper concluded that the best design architecture depends on the needs of the developer since they have different architectural designs. In another finding, Beal [4] in his article, "*Understanding Web Services*" reports that web services have shaped the paradigm of business communication between client and servers. Most organizations such as eBay, Amazon.com, and Social Networking sites such as Facebook use Web services to develop their API's [34]. The underlying protocol on which Web services run ensures the implementation of WS on ubiquitous objects [34].

Also, mobile Web services support the integration of various WS designs in mobile distributed systems. Farley et al. [35] identify mobile Web services as a way to deploy traditional Web services on the mobile device due to constraints such as screen size and the personalization of apps. Farley et al. also discuss a mobile Web services architecture that is easily consumable on multiple devices such as Personal Digital Assistants, laptops, mobile phones and tablets. The paper proposes a mobile Web services implementation that offers authentication services between the client and the web server. Users can access a map on their mobile devices from a map Web service provider. Mobile users can also share their personal profile contents with others.

As well, in mobile Web services, developers in the Mobile Computing field are not concerned with which particular Mobile Platform OS to choose since HTTP is platform independent [4]. To further understand which Web services framework best supports mobile devices, this thesis explores SOA and REST in the next sections.

2.3 SOA

The Service-Oriented Architecture (SOA) framework provides support to various components of Web services to interoperate. According to Wicks et al. [8], SOA focuses on reusability of software and integration. Another key thing about SOA is packaging, which makes changing of older versions of software very fast and at minimal cost [8], [36].

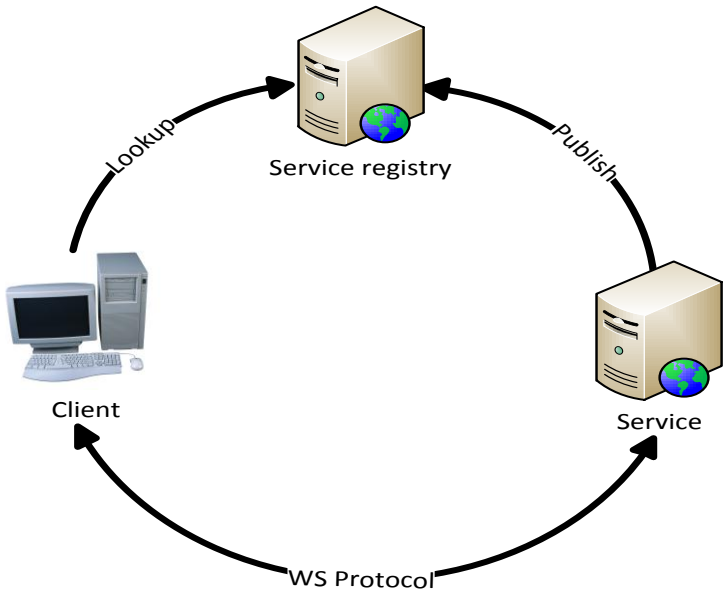


Figure 2.3: Service Oriented Architecture framework [10], [37]

Figure 2.3 describes the software components that are modeled as services [37] in a SOA framework. The *Service* constituent is a platform ("service interface") that is exposed to processing software that control a specific set of tasks [37]. The *Client* module discovers services

that are reusable and engages in a bi-directional message exchange via internet or intranet with the Service module. The client also uses UDDI and WSDL to discover services and produce SOAP messages [10]. The *Service registry* is a repository where service providers publish services.

According to a project report on the practicality of the SOA framework, Rusu et al. [38] report on the performance of CardioNet; an E-health app for heart patients. It is a distributed system implemented using Service Oriented Architecture (SOA) where everything (i.e. hardware, software, and medical activities) is modeled as "services". The system also uses the high level protocol SOAP. Rusu et al. report that CardioNet guarantees interoperability and permits heterogeneous systems' integration.

Though SOA aids Web services developers to overcome interoperability [9] problems, it has some limitations when implemented in mobile distributed systems. SOA is XML based since it uses SOAP; and this makes simple communication between system components in mobile distributed systems challenging. The difficulty is due to the fact that SOAP passes large XML data; thus the consumption of data becomes a problem on mobile clients which have limitations of processing and storage [10]. Another hurdle in the SOA framework is that, to achieve cross platform interoperability between Web services, a lot of standards have to be followed. Though standards such as security, integration and management have been projected for SOA, there is no common platform that integrates all the standards [36].

Also, Lee et al. [39] report on the challenges faced by Telco with issues on scalability due to earlier SOA design to build transactions. The paper, as well, identified a challenge of poor performance of the system under heavy workload. As a solution to address the challenges in SOA, they propose Resource Oriented Architecture (ROA) [39], [40] approach.

2.3.1 SOAP Based Web Services

Simple Object Access Protocol (SOAP) is explained by Pautasso et al. [9] as a Web specification that provides interoperability for heterogeneous Web services. This is achievable because SOAP messages can be exchanged over multiple communication protocols using HTTP and other transport protocols. Also, the use of HTTP for transporting SOAP messages has reduced the challenge for building services that can run on the Internet according to Cartwright [41]. Though the standard protocol used by SOAP is HTTP, it can use other protocols as well. Figure 2.4 shows a complete cycle of a client-web server request-response interaction based on the SOAP framework. The client envelops a SOAP message with clear description and sends it over a network transport protocol as a request which is opened when received at the web server for an appropriate response. The web server also sends the SOAP response back to the client in a standardized format that the client can understand.

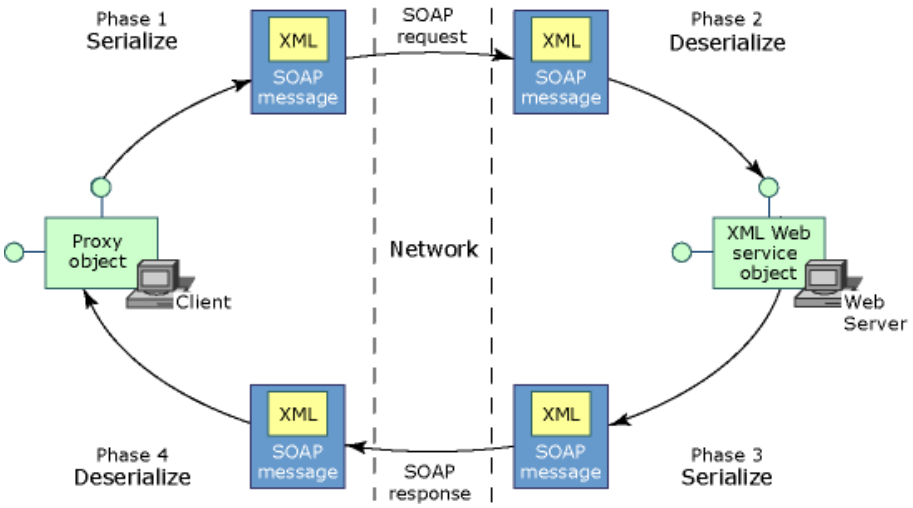


Figure 2.4: SOAP based Web Service lifetime [42]

A sample SOAP message is shown in Figure 2.5 where a client requests is sent to the server for the creation of a record for a patient with the name R. Kwadzo.

```
POST /NewPatient HTTP/1.1
Host: www.soapws-example.net
Content-Type: application/soap+xml; charset=”utf-8”
Content-Length: 324

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
<soap:Header>    </soap:Header>
<soap:Body>
<m:NewPatientRecord xmlns:m="www.soapws-example.net/patient">
<m:PatientName>R. Kwadzo</m:PatientName>
</m: NewPatientRecord>
</soap:Body>
</soap:Envelope>
```

Figure 2.5: Sample SOAP request message

The application of SOAP technology is seen in the research of Uribarren et al. [13], Beltran et al. [14] and Halteren et al. [15] because of the flexibility offered by SOAP. The key challenge that these works address is building software applications that are platform independent. The authors use SOAP as the Web services mechanism and as a result, the client devices are able to consume the services.

However there have been some challenges associated with the use of SOAP in heterogeneous Web services. Some developers describe SOAP as a complex way of design while others saw that the simple way of turning *legacy applications* (i.e. older versions but running application) into Web services can be misused [9], [43]. Furthermore, SOAP uses the HTTP POST method to send large messages therefore causes network traffic in mobile distributed environment [10]. As well, there are core memory problems that arise in mobile distributed systems with the use of SOAP because it’s a heavy protocol that sends and receive messages [10].

2.3.2 WSDL

Web Service Description Language (WSDL) is an XML based standard that formally describes a Web Service (WS) [8]. WSDL describes the transactions of Web services by looking at the messages used by the Web service, the data types, and the communication protocols [8].

WSDL provides a standard for describing services that are offered in electronic transactions, provides business services functionalities, and provides a data in a suitable format for Web based client consumers [3], [34].

2.3.3 Summary

The works presented on SOA show that Web services can be modeled and consumed as services in distributed systems. Since SOAP uses HTTP as a communication protocol and WSDL to give interpretation to web services in a client-server interaction, SOA Web services are interoperable, platform-independent, and reusable.

However SOAP messages are large so consuming them on mobile clients is difficult due to the memory and CPU constraints on the mobile devices. The exchange of these large XML data also increases bandwidth usage. As a solution to these challenges, the works presented by Wang [10] and Lee et al. [39] employed the ROA and REST designs.

In the next section, the current studies on ROA and REST are investigated to determine how they are being applied in modern WS designs.

2.4 REST

2.4.1 REST Architecture

REpresentational State Transfer (REST) is a term coined by Roy Fielding [11], in his Ph.D. dissertation, “Architectural Styles and the Design of Network-based Software

Architectures", as architectural principles that use the Web platform for distributed computing. REST is better understood in the context of identifying everything as a resource, representation, and state [44]. The design follows certain technological principles:

I. *Everything is a resource:* In REST design, all the identifiable entities must be considered as a resource and should be assigned an ID [45].

II. *Identification of resources through URI:* The key resources should be given Universal Resource Identifiers (URIs) which will facilitate interactions within the system. The URIs provide a global namespace for resource and service identification [9], [45]. Figure 2.6 lists sample legal URIs that can be used in identifying resources.

```
http://restfulwstest.com/patients/2011/1
http://restfulwstest.com/patientlist?age=60
http://restfulwstest.com/patients/1234
http://restfulwstest.com/patients/sickness-234
```

Figure 2.6: Sample REST URIs

III. *Uniform interface:* Resources can be manipulated through representations using HTTP methods. The following HTTP methods are noted in [11], [45].

- *GET:* This method is used for resource retrieval.
- *HEAD:* This method's request-response is similar to GET request but the requesting resource only receives the response headers without the entire message body.
- *POST*: This method is invoked to push or create a new resource.
- *PUT*: This method alters (update) the state of a resource.
- *DELETE:* The DELETE method is used for removing the specified resource.

- *PATCH:* This method is used to update parts of a resource without changing the state of the entire resource.
- *OPTIONS:* This method checks and returns the functions of a web server [45].

There are other HTTP methods as well such as TRACE and CONNECT. The use of the verbs must be controlled by the developers because while some are idempotent and safe, others are not [45]. Idempotent methods have the same state effect on a resource when invoked once or multiple times. Examples are GET, HEAD, DELETE, OPTIONS and TRACE methods. Safe methods have absolutely no state changes when invoked on a resource. Examples are GET and HEAD methods. The POST method is neither idempotent nor safe [45].

IV. *Self-descriptive messages:* Since resources are decoupled from their representation, it makes content accessibility very simple regardless of the format of resource content [9]. The available meta-data of the resources can be used to control caching, detect transmission failures, and enforce data security measures [9].

V. *Stateless interactions:* While resources have states, their interactions should be kept stateless. At the end of every transaction, resources should have information about themselves but not how the last interaction was done [45].

VI. *Hypermedia as the engine of application state (HATEOAS):* In order to navigate between resources, URIs such as hypertext can be used in a resource representation [46]. HATEOAS aids the client to know the next steps to take since the returned URI contains links to available options.

2.4.2 WADL

Web Application Description Language (WADL) is an XML document that defines the implementation of RESTful Web services [47]. WADL also uses HTTP as a communication

protocol. Burke [47] described WADL as the WSDL of RESTful Web services. Burke built a JAX-RS application using Jersey; and WADL is used to define the resources as well as the HTTP methods and the resource states.

WADL requires machine input for HTTP based web apps. The machine readable parts of web apps can be a set of resources, links between resource components, the HTTP uniform interfaces, and the representation of resources [48].

2.4.3 RESTful Web Services

RESTful Web Service (REST-WS) is a Web service framework that is built on the architectural principles of REST and communicates over HTTP [12]. In "*Richardson Maturity Model: steps toward the glory of REST*" [49], four levels of abstraction for creating Web services in REST are noted as reproduced in Figure 2.7.

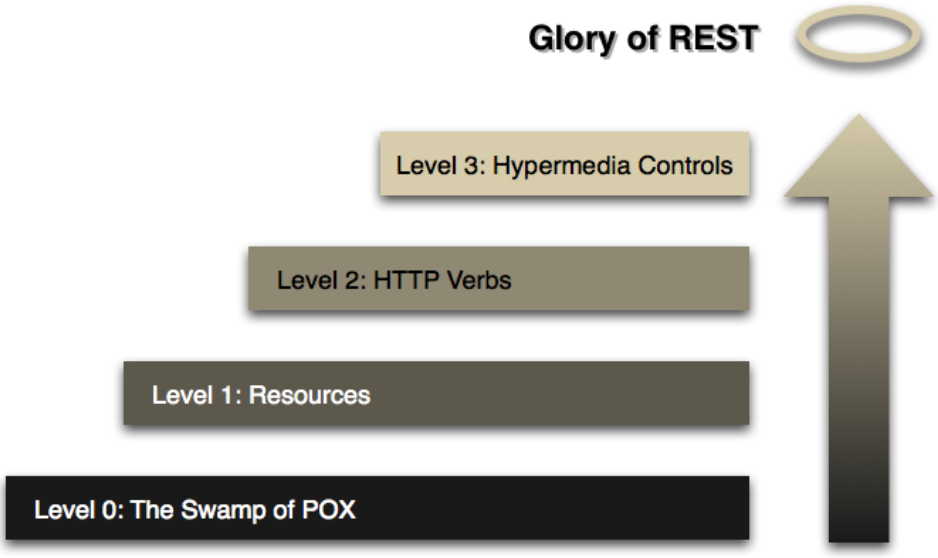


Figure 2.7: Levels in REST design [49]

- *Level 0*: This level is the lowest level in the RESTful design hierarchy. The design at this level describes the use of Plain Old XML (POX) to create simple request-response to a single endpoint over HTTP.
- *Level 1*: This level is one step above Level 0. The Level 1 design focuses on building resources that can interact individually in a *1* to *N* relationship. In this way, the traditional way of interacting with a single service endpoint in Level 0 is improved. Also, the HTTP methods employed at the Level 1 are GET and POST.
- *Level 2*: Designing RESTful Web services at this level encompasses all the Level 1 requirements in addition to all the other HTTP methods for most of the interactions. Level 2 supports the Create, Read, Update, and Delete (CRUD) methods. Also, the methods at this point are consumed by HTTP itself to determine safe operations.
- *Level 3*: RESTful Web services at Level 3 combine the requirements of Level 2 and uses hypermedia controls for making protocols more transparent in a global namespace. At level 3, the client requests are responded to with additional URIs that informs the client of the next steps to take.

Additionally, there are developers who strictly adhere to the use of the CRUD methods and hypermedia controls. These developers are described as "*purists*" [50]. However, there are developmental needs which may demand the use of POX with another method such as POST to create, update and delete a resource. The developers who opt for the Level 0 and Level 1 are classified as "*pragmatist*" [50]. Lee et al. [39] and Selonen et al. [51] in their research use the terms low-REST and lightweight REST respectively to describe the pragmatists view. In Figure 2.8, the opinions of the purists who use Hi-REST and the pragmatists who use Low-REST are illustrated.

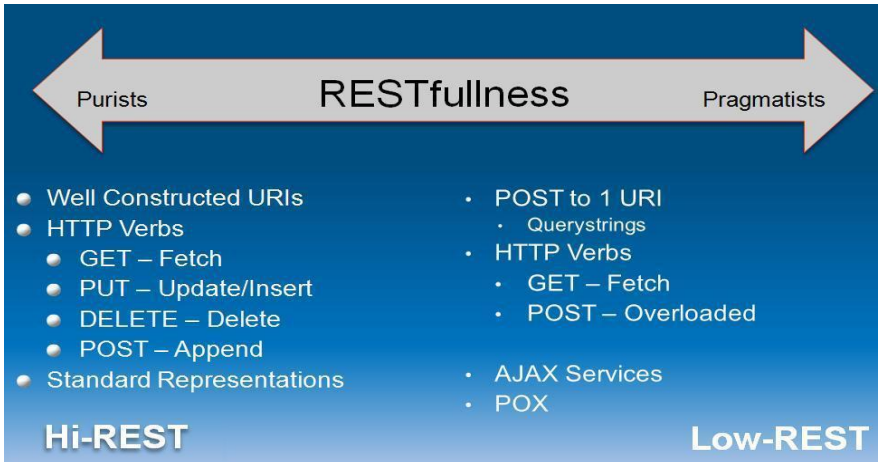


Figure 2.8: Views on REST [50]

3.4.4 Mobile RESTful Web Services

RESTful Web services provide a more efficient architecture that enables mobile clients to communicate with the server through proxies [12]. This is because the provision of stateless interaction in REST reduces the impact of network volatility in mobile distributed systems [12]. Also, REST uses URIs so Web services apps can easily be invoked from mobile clients; and since responses are transmitted over HTTP, the effect of network instability is reduced [12].

In order to make use of the flexibility provided by REST-WS, Selonen et al. [51] reported their findings on building mixed reality service on a mobile device. The paper adopted REST as an architectural style in order to achieve interoperability, decoupling, scalability and security. A service model is proposed which is later changed to resource model using Unified Modeling Language (UML) class diagrams and the final implementation done as a lightweight method of REST. The paper proposed a functional requirement that integrate client (mobile) and server interaction. Photos are sent from a smartphone with meta-data and stored in a registry on the server that is accessible by other clients and 3rd party applications using URIs. This approach, Selonen et al. [51] noted, aided their system to render services on multiple mobile clients and web browsers. Selonen et al. [51] use Web Application Description Language (WADL) based on

Java EE-Hibernate- Restlets which helped them to use uniform interface and define resource states.

Another research that explores mobile RESTful web services was conducted by Stirbu [52]. The paper aims at building an adaptive and multi-device application sharing service that gives the same look and feel of applications deployed on different mobile platforms. Stirbu [52] implemented an event-based RESTful architecture that efficiently consumes web services on the mobile device. Since resources contain self-descriptive messages, interactions between them are converted to events which are controlled remotely to deliver contents with the same functionality on different systems. This work employs the ARRESTED [52], [53] architecture as a means of overcoming the well-known challenge of REST where the client is usually the one that initiates conversation with the server. Using WADL, synchronization between client and application host is achieved since any of them can initiate conversation in a request response pattern. Stirbu [52] observed that using REST aids his system to scale with multiple users and devices.

According to Sletten [54], combining RESTful design with other technologies like caching provides great system scalability. Figure 2.9 shows Sletten's graphical presentation of how the combination of REST and caching can provide a flexible and scalable architecture.

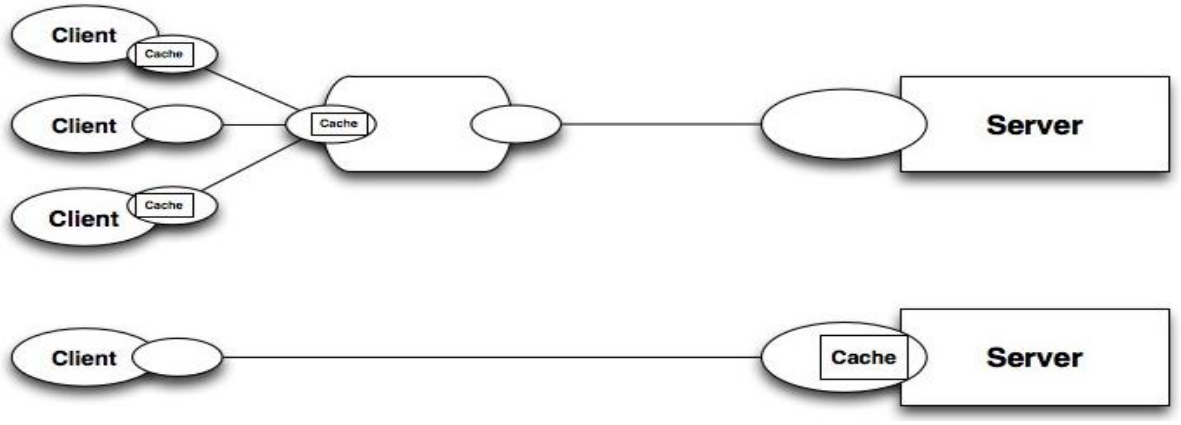


Figure 3.9: Caching in REST [54]

On the weaknesses of REST, Stirbu [52] highlighted some limitations imposed on the system at the network level which has to do with response request. Furthermore, Han et al. [55] argue that many firewalls permit only the GET and POST methods. There is also a size limit on URI for GET method encoding [55]. Also, HTTPS requests are not cacheable due to security concerns [55].

2.4.5 Computational REST (CREST)

Although the REST architectural pattern helps in achieving the dream of a scalable web based systems, Erenkrantz et al. [56] observed that REST does not focus on the architecture of individual components of the distributed system. The authors after exploring the best ways of building RESTful web applications and how future web services could fit into the REST paradigm, propose CREST as a way of expanding the current REST architecture. Figure 2.10 above shows a sample code for CREST in JavaScript.

```
{
        if (wordcount) {
           return wordcount(GET(
              "http://www.example.com/"));
        }

}
```

Figure 2.10: Sample CREST program in JavaScript [57]

The importance of shifting to CREST is because many current web communications involve computational results rather than contents [56]. Instead of receiving data as content or a hyperlink for consumption by the client from a request-response interaction, data is received in a form of computation(s). These computations determine what actions the client needs to take next after the execution of the current computations. Erenkrantz et al. [56] use closures and

continuations in JavaScript to build a computation based web service that exchanges messages between the client and server. URIs in this environment are recognized as computational resources and all interactions are treated as computational exchanges. The main design principles of CREST are outlined by Erenkrantz et al. [56], [57] as follows:

I. *URL defined resources are modeled as computations:* All entities that have a legal URI are computations and as such can be described as resources. Examples include images, orders, generated services, and simulation of an object.

II. *The representation of a computation is the collection of an expression and metadata to describe the expression:* Assuming resources are represented in a particular data format; plus a metadata to describe those formats, then there is a linkage between the abstract resource and its actual representation. In that case, In CREST, it is easy to negotiate for a representation.

III. *All computations are stateless (context-free):* CREST just like REST keeps the state of resources but the interactions are kept stateless. Each computation contains enough information that determines what actions to take next without knowing anything about the preceding computation.

IV. *Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged:* Representations are defined by participants based on their specific binding environments. That is, there are limited HTTP operations that are supported in CREST but the developer can define more operations on the server side and expose the operations globally to all clients.

V. *The presence of intermediaries is promoted:* There is transparency between the client and server so all modifications made to a representation in a request-response interaction is seen by all.

Additionally, apart from the above mentioned design principles governing CREST, there are certain principles that CREST focuses on according to Erenkrantz et al. [56], [57]. These principles are outlined below:

I. *Computational Namespaces:* CREST uses URIs as computational representations to make non-human readable expressions that can be stored on nodes for later message exchanges.

II. *Services*: CREST allows multiple URLs to access a single service. This is why CREST supports multiple interfaces.

III. *Time:* Computations in CREST may evolve over time based on a user's computational loads. In view of this, when multiple computations are being processed within the system, CPU consumption may increase.

IV. *State:* Multiple computations can be spawned simultaneously.

V. *Computation*: CREST references an object directly or indirectly using computational representations.

VI. *Transparency*: Because URLs are made available within computational namespaces, caching, routing, and inspection of computations can be achieved easily.

VII. *Migration and Latency*: Computations can be moved and stored on to multiple nodes thereby reducing the overhead of computation.

Also, CREST design does not necessarily have to follow all the above outlined principles. The design guideline of the CREST framework is tested by Erenkrantz et al. [57]. In the research, a sample Feed Reader Application is implemented to test CREST based on all of its

principles. The application consumes RSS feeds based on computations. The client JavaScript provides users with a graphical interface that allows users to create links to widgets, select URL and choose a date from calendar. The widget manager serializes all the user's preferences together as JSON and passes it over HTTP to the backend. Closure and continuation are used to render the requirements of the user in a browser as an atom feeds.

2.4.6 ROA

The Resource Oriented Architecture (ROA) presented by Overdick [40] follows the guidelines of REST. The ROA design is employed by Xu et al. [58] who used the Web services framework for JAVA called RESTlets to build a ROA system. A comparison of the result of the ROA system was made with service-based web services. The ROA system proved to be a more flexible approach to building business processes because of the use of hyperlinks that enable processes to communicate. The use of a uniform interface (HTTP method) exposes resources to be managed more easily than in SOA where service components have to be created at every operation. Xu et al. [58] also proved that ROA business processes allow process visibility. This is good especially in the sense that the client requester can communicate with the server periodically for updates.

Also, Selonen et al. [51] report on building a lightweight architecture style for building ROA as a solution to their "mixed reality" project. This aided them to achieve the aim of storing, retrieving, and managing interlinked content which otherwise couldn't be successful with SOA.

For the purpose of analyzing the importance of ROA and SOA, the two are compared in Table 2.1.

Table 2.1: Comparing ROA with SOA

Objects (Entities)	• In ROA, entities are modeled as resources but in SOA they are services.
Increased Adaptability	• ROA implementation is more adaptable than SOA [39], [58].
Scalability	• Lee et al. [39] and Selonen et al. [51] report that ROA is more scalable than SOA.
System Performance	• ROA design handles bigger workload than SOA [39].
Interoperability	• Pautasso et al. [9] view interoperability as the major strength of SOA but [58] and [51] report otherwise.
Decoupling	• ROA is better than SOA [51].

2.4.7 Summary

The works in REST show that it is a light-weight protocol that provides consumable Web services to resource constraint devices such as smartphones and tablets. The REST architecture provides the platform for resources to be cached anywhere in mobile distributed systems [50]. Additionally, the use of REST is good for bandwidth management [10].

Though REST has certain problems like no caching for POST request, it is still a better alternative to SOA. Other challenges such as the client being the initializer of a client-server interaction can also be overcome with CREST.

2.5 Cloud Computing

In order to maximize the full functionalities that smartphones and tablets provide its users, cloud computing is reviewed. In the work "*Using RESTful Web-Services and Cloud*

Computing to Create Next Generation Mobile Applications," Christensen [12] focuses on the network capabilities of smartphones by linking context-enable features on these devices to cloud computing. Smartphones, unlike desktop computers, are constraint with storage space and memory allocations. Hence, exploring the power of cloud computing aided Christensen to store some application features and data in the cloud. Since mobile devices have connections either through Wi-Fi or Bluetooth, data in the cloud can be accessed at real-time. Figure 2.11 shows how the cloud can be used as a platform, as a service, as an infrastructure, and as a storage.

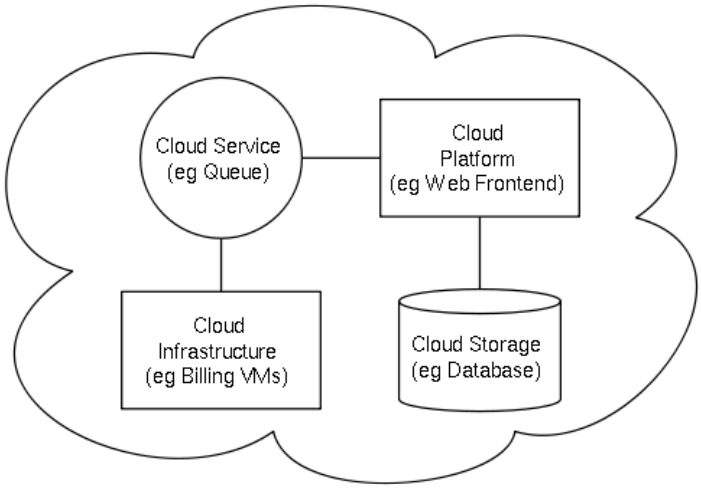


Figure 3.11: Cloud computing model [59]

Additionally, Christensen [12] mentioned the convenience of building Web services RESTful for smartphones since REST requires passing simple XML. REST allows the use of HTTP HEAD operation which enables efficient caching on the smart mobile device. Christensen [12] also reports on how the combination of smart mobile phone, REST based cloud computing and context enablement transformed the mobile application paradigm to activity based service. Furthermore, the ability to choose between Open ID and Open Auth provides cloud security.

In another research, Wang [10] shows that the cloud provides a stable network environment for data accessibility. As a result, network availability and reliability is highly guaranteed in the cloud. Wang [10] also reports that the server hosted in the cloud is scalable as the workload increases with good performance.

With the cloud providing high platform availability, the attention of this thesis switched to the middleware proxies and how mobility can be supported in mobile cloud environment.

2.6 Middleware Proxies

In an attempt to understand how current research in the mobile computing field has addressed latency and data sharing issues, middleware applications are reviewed. Middleware is explained as software that "enables applications to interoperate with each other, enabling them to form and present different views of, and ways of accessing, functionality and data from and across each other's systems, regardless of their native platform or operating system." [60]

Uribarren et al. [13] outline some issues they faced with building location adaptive and multiple platform application for ubiquitous objects in a distributed environment. The challenge that the paper attempted to solve is how to implement a system that automatically adapts to contextual information based on a user's usage preference and available network support. Other challenges were how to build a cross-platform system that can render platform-specific information on a desired client, and support for multiple data formats. Uribarren et al. [13] propose a middleware infrastructure that stores a user's location, a user's device and his/her current activities online. Whenever a user changes location, the middleware stores the new location and pushes the previous online activities to the user. This scenario makes it possible for users to access data on multiple nodes since the data is on the middleware. The middleware

accepts data in XML format so it is easy to pass it into other formats that can be consumed by other ubiquitous objects.

Also, Beltran et al. [14] use middleware oriented approach to deal with the issues of presence status of users in mobile services environment. Understanding the presence status such as offline, available, busy, and so on of an online user enables user applications to adapt to what information they can deliver to the client. Beltran et al. [14] propose a personal proxy middleware that ensures presence for adaptable client action based on the information from the proxy. The wired environment proxy enables presence states to be stored and managed on the middleware and the client only connects using an API. This approach aids the efficient use of resource constraint devices because the processing is done by the middleware. The middleware approach adopted by Beltran et al. [14] also help the system to be highly secured because each client is shielded from any other client since all communications are routed through the middleware proxy.

Halteren et al. [15] further explore the use of middleware in order to achieve reliable client and service interaction, system scalability, and reduce the processing requirement on the mobile devices. The middleware identifies the client services through HTTP connection and executes all the processes and pass the processed information back to the client. The wired network middleware enables the proxy to facilitate reliable message delivery. The proxy is tested in an e-Health domain and patient data is successfully sent from censors to doctors in other locations.

2.7 Mobility

2.7.1 Middleware Caching

Most of the literature on middleware use caching as a way of manipulating resources. Halteran et al. [15] use middleware caching as a way of reducing client side workload on the mobile device. As well, the research by Alarcón et al. [61] uses middleware caching to store executable applications that are exposed to mobile clients. In case a client wants to use an app, it searches through the middleware cache to check whether it exists. All existing apps in the middleware storage are then uploaded to the client. Alarcón et al. [61] concluded that this is an efficient way of reducing the processing demand and reducing memory usage on the resource constraint client.

Furthermore, Beltran et al. [14] use middleware storage to keep the user status (presence) in mobile presence service. The large amount of information regarding user presence that has to be processed for real-time delivery is processed on the middleware and current user status is stored. This allows mobile consumers to receive quick updates in the proposed distributed system.

2.7.2 Mobile Client caching

Sumari et al. [62] report on the use of mobile caching to store segments of videos in mobile video on demand systems. Due to limited bandwidth in Mobile Ad hoc Network (MANET), mobile clients have to wait for a long time before the video content is delivered to them. Sumari et al. [62] propose neighbor-based caching module, which is a system that allows mobile clients to contact other mobile clients and request for video segments from their cache. This approach coupled with middleware routing of video from client to client reduced the waiting time within MANET significantly.

Also, Gomaa et al. [63] explore the design of mobile caching policies and management. The paper as well as other works by Wang [10] and Sumari et al. [62] concludes that mobile caching reduces the impact on the volatile network bandwidth in the wireless environment. This is because mobile caching minimizes the number of requests in a client-server communication. Gomaa et al. also propose Least Frequently Used (LFU) caching as the best caching policy. The LFU is a cache validation methodology that is employed to remove data that is not regularly accessed in the cache.

Furthermore, the mobility of handheld device users outside a Wi-Fi or WLAN environment is another motivation for mobile caching. Users can still access local copies of media even if there is no connectivity though the media may be outdated [10], [63].

2.7.3 Mobile Service Hosting and Workflows

Mobile devices are evolving from client consumers to client service providers due to their sophistication in recent time. Pajunen et al. [64] propose application and services execution workflow module on mobile devices. The authors define a mobile workflow that can run in the absence of network oriented processes or services. The workflow also provides an easy integration system in case the network services become available. The authors also note that "one can start to expect mobile devices to be more than just interfaces to services in the network, but they can execute and offer services by themselves". Using Web services, Pajunen et al. [64] support their claim with the implementation of a workflow system that allows a supervisor to assign duties to a supervisee. A supervisee on his mobile device can take a decision by accepting a duty or rejecting it. Whatever decision is made by the supervisee will determine the next action that a manager can take. This implementation aids users to assign roles and receive feedbacks remotely.

Also, Rosenberg et al. [65] propose a RESTful Web service workflow that combines the principles of REST. The architecture proposed by the paper achieves simple human-computer interaction, collaboration services with external applications, and easy interfacing with back-end services. The implemented application called Bite, models all processes as resources and uses HTTP verbs to create, fetch, and remove process instances. This has led to a collaborative workflow between different client devices that want to share different processes.

2.8 Event Handling and State Change Propagation

2.8.1 Middleware Event Support

Pritchett [22] proposes the use of an event-driven architecture as a simple means of determining consistency of resources that are stored on multiple hosts in a distributed system. The event notification mechanism informs clients about updates that have arrived within the system. The use of notification events between system components enabled the client or server to take an action based on the state of resources. The components in the distributed system also rely on event mechanisms to determine whether a particular host is dead or alive.

Also, Sheng et al. [16] report on building a personalization app on a mobile device that takes into account a user's location and a user's service needs. The challenges faced with building the application were the constraints of mobile devices and how updates were sent to users based on their (users) needs in a wireless network. Sheng et al. [16] use event notifications that monitor updates to event sources. There are events that listen to incoming requests from all subscribers and based on the user's location, another event is fired up that pushes the user subscription messages to the mobile client. By this approach, they successfully had a system that rendered personalized data and context-aware data to the client.

In addition, Da Rocha et al. [66] propose an event based middleware approach for deploying ubiquitous context-aware application that achieves application transparency in a scalable distributed system. The paper explores publish/subscribe mechanism to manage events such as asynchronous communications and rendering personalized information to consumers. Subscribers use an API to register for the contextual events they want to receive. The event handler at each service host is tasked with delivering contextualize information to the registered consumer. The paper concludes that this approach enhances real-time delivery of information to the consumer.

2.8.2 Client Event Handling in RESTful WS

An event-driven messaging architecture has been proposed by Li et al. [67] in RESTful web services that aid client consumers to receive message updates from the server. The events are: the client can register to receive a message from another client, the client can accept or reject a message, and the client can cancel a registration. All incoming messages have associated events that the client uses to determine its action. This style of event implementation ensures that messages and events are sent to only the intended recipients. This approach is different from the bi-directional two-way messaging system where messages are pushed to the client even if the client has not requested for it.

In another research, Stirbu [52] proposes a mobile distributed system that allows sharing a RESTful WS application on adaptive and multiple hosts based on event-driven mechanism. The focus of his research was to render a single app on many devices with the same look and feel in device-aware context. Stirbu [52] modeled user interfaces as resources so each HTTP GET operation permitted a client consumer to acquire a resource state that was platform specific to the client. The HTTP operations GET, POST and PUT are all events. Anytime POST or PUT is

invoked, a corresponding update event is triggered that causes a remote Model-View-Controller, which is implemented, to model the updated resources for specific platforms.

2.9 Conclusion

Consuming WS on resource constraint devices and efficient bandwidth management: The background works show that Web Services (WS) can be built and deployed on many platforms. Also, WS can be built using SOA but REST-WS are consumed by resource constrained devices efficiently because REST passes simple data as compared to SOAP [10], [12]. REST also provides high system scalability due to its stateless interaction technique. Most of the related work compromise on some of the REST requirements in order to achieve their desired results. These types of REST systems are described as low-REST or lightweight-REST [39], [50], [51]. As well, in the REST design framework, resources are cacheable on the client.

In addition, cloud computing is used for storage of applications that can be consumed by smartphones and tablets via Wi-Fi. This approach is to reduce the workload on the limited resources of these devices [10], [12].

Network latency between the client devices and the middleware: The literature review also focuses on how to deal with transient connectivity issues between the mobile client and the Web server. Some researchers propose middleware systems that store resource states for real-time access by users [10], [14]. The middleware, which act as proxies, provide push techniques to clients so that updates can be received in a networked environment in real time. Another approach to dealing with latency issues in a wide-area system is implementing multiple servers and enabling clients that are closest to a server to establish communication [31].

Caching: Resource storage on the client is possible when Web services are developed in REST. Client caching is good to push data to the user even if there is loss in connectivity

between the client and server [62], [63]. Moreover, resources can also be cached in the middleware. Middleware storage facilitates resource routing and sharing to hosts in a mobile distributed environment [14], [15]. Mobile clients can be modeled as resource consumers and service providers. In this case, caching and process execution workflows can be implemented on the mobile client to efficiently provide services to users and also manage the constraints on the mobile devices.

Middleware and state propagation: The CAP theorem phenomenon exists in mobile distributed systems. Most of the related literature explained it in the context of databases but it is also applicable in mobile distributed environments that consume heterogeneous Web services [26]. As resource states are stored on client devices in an unstable Wi-Fi network, there will be partition failure due to the intermittent loss of connectivity. A middleware implementation is used to leverage the workload on the client and provide an interface barrier for client-server interaction [13], [14], [15]. The middleware responds to state change issues and accordingly notifies the clients. Golding [31] proposes the use of timestamps to requests as a way of knowing the order in which clients can be updated.

Events: Event notifications can be used on the middleware to ensure consistency in data and monitor state changes in resources [22]. Clients can register for specific events whiles proxies can use events to determine client states [16], [22], [52], [66], [67].

The list of papers reviewed within the identified problem areas are listed below in Table 2.2.

Table 2.2: List of reviewed papers within the project domain

Web Services	• Beal [4] explained that Web services foster high integration and are mostly platform independent. • Web services technology supports multiple styles such as RPC, SOA, and REST [3]. • Mobile Web services provide resource sharing capabilities on resource constraint devices [35].
SOAP	• SOAP is platform independent [41]. • SOAP supports language-independence [41]. • Han et al. [55] report that SOAP encourages protocol transparency. • SOAP is a complex way of design and the simple way of turning legacy applications into Web services can be misused [9], [43].
REST	• Martin Fowler [49] explained the four levels of the Richardson Maturity Model. • Rodriguez [11] covered the basics of REST and its underlying principles. • Adamczyk et al. [68] explain the principles of REST and how it applies to enterprise apps and caching.

	• Alarcón et al. [61] report on REST services application to some social networking sites. • Hadley et al. [46] report on the application of HATEOAS. • Parastatidis et al. [69] wrote on how to expose RESTful protocols in business workflow environment over the Web. • Kelly et al. [70] worked on REST and caching. • Engelke et al. [71] report on how to use REST design to modify existing applications. • The rationale behind the use of REST beyond the WWW infrastructure is explained by Fernandez et al. [72]. • The applicability of REST in distributed web environment is the focus of the work of Hernández et al. [73]. • Jacobi et al. [74] report on the use of REST as a way of reducing bandwidth consumption in messaging.
RESTful Mobile Clients	• Tilkov [45] writes that the stateless interactions between resources in a RESTful design, results in high system scalability. • Christensen [12] reports that REST is desirable in mobile environment because it reduces the impact on network usage. • Selonen et al. [51] justify the use of low-REST style in their work in order to achieve interoperability, decoupling,

	scalability and security. • Stirbu [52] reports that the use of REST style in his work helped to deploy content with same functionality on different systems. • At the network level, there can be limitations with the request-response interactions [52].
Computational REST	• Erenkrantz et al. [56], [57] mentioned that CREST Web applications are highly consumed by mobile clients. • CREST also focuses on computations rather than hyperlinks [56], [57].
Cloud Computing	• Christensen [12] reports on using cloud infrastructure as storage in order to efficiently manage the constraints on mobile devices. • Wang [10] also reports from his research findings that the cloud environment provides system availability.
CAP Theorem	• Two of the three requirements (Consistency, Availability and Partition-tolerance) can be guaranteed simultaneously if one requirement can be traded-off [18], [19], [20], [21].
ACID	• Atomicity, Consistency, Isolation, Durability • ACID is a set of properties that is followed to ensure high data consistency [22].

	• Transactions are atomic thus should be successful or fail entirely [19], [22].
SAGAS	• SAGAS avoid locks and serializations in long running transactions [24]. • Long-lived transactions can be divided into short-lived transactions with linkages known as compensation handlers [24].
BASE	• Basically Available Soft-state Eventual consistency (BASE) ensures high availability [22]. • Ignores two-phase commit and ensures latency. • Eventual consistency is achieved if the system allows for a time lag between operations [22].
Middleware	• Uribarren et al. [13] propose middleware for transforming data into a format that is consumable by ubiquitous devices. • Beltran et al. [14] use middleware to provide user presence services in mobile environment. • Halteren et al. [15] propose the implementation of middleware to ensure high scalability reliable message delivery in distributed systems.
Resource Caching	• Caching is possible on the middleware side in order to reduce the processing workload on the client [14], [15].

	• Client caching is beneficial for providing resources to users and other clients in an unavailable network environment [62], [63].
Event Handling	• Pritchett [22] uses event notification to ensure resource consistency. • Sheng et al. [16] use update notification events to monitor state changes of services. • Events can be implemented following the publish/subscribe technique [66]. • Clients can also register and manage events in a distributed system [52], [67].
Mobile Service Hosting and Workflows	• The mobile device can be a service provider as well as consumer [64]. • Workflows enable the mobile device to execute business logic and transactional services in a distributed environment [64], [65].

2.10 The Open Issues

In this thesis, the following unresolved issues in the problem definition have been identified.

- **Network Latency:** There is network latency within Wi-Fi environment due to transient connectivity. Short-lived disconnections can be hidden from client nodes and users by pushing cached data to them (i.e. ensuring distribution transparency); but this is not practical in long-lived disconnections. This therefore leads to the following open question:
    - How to model the mobile device as a resource and service provider that supports users at soft real-time?
- **State Synchronization:** Messages sent to nodes within mobile distributed systems can be lost due to the intermittent disconnections in mobile wireless networks. Hence, messages are sent frequently to all mobile nodes but this approach introduces a lot of overheads. The following challenges therefore remain:
    - How to minimize overhead?
    - How to identify which consumers and providers have the latest updates of a resource and which resources must be updated?
    - Which eventual consistency model is the best approach for reliable update propagation?

CHAPTER 3
DESIGN AND ARCHITECTURE

3.1 Overview

The objective of this thesis is to explore the capabilities of mobile devices as hosts of providers of RESTful Web services. To achieve this goal, a middleware is proposed to support the interactivity among the mobile devices in an unreliable Wi-Fi environment. The architectural design is classified into three tiers namely: mobile service requesters, middleware, and mobile service providers. Figure 3.1 highlights how the three tiers are linked. The middleware is hosted on the cloud; identical to Wang's work on "*Mobile Cloud Computing*" [10].

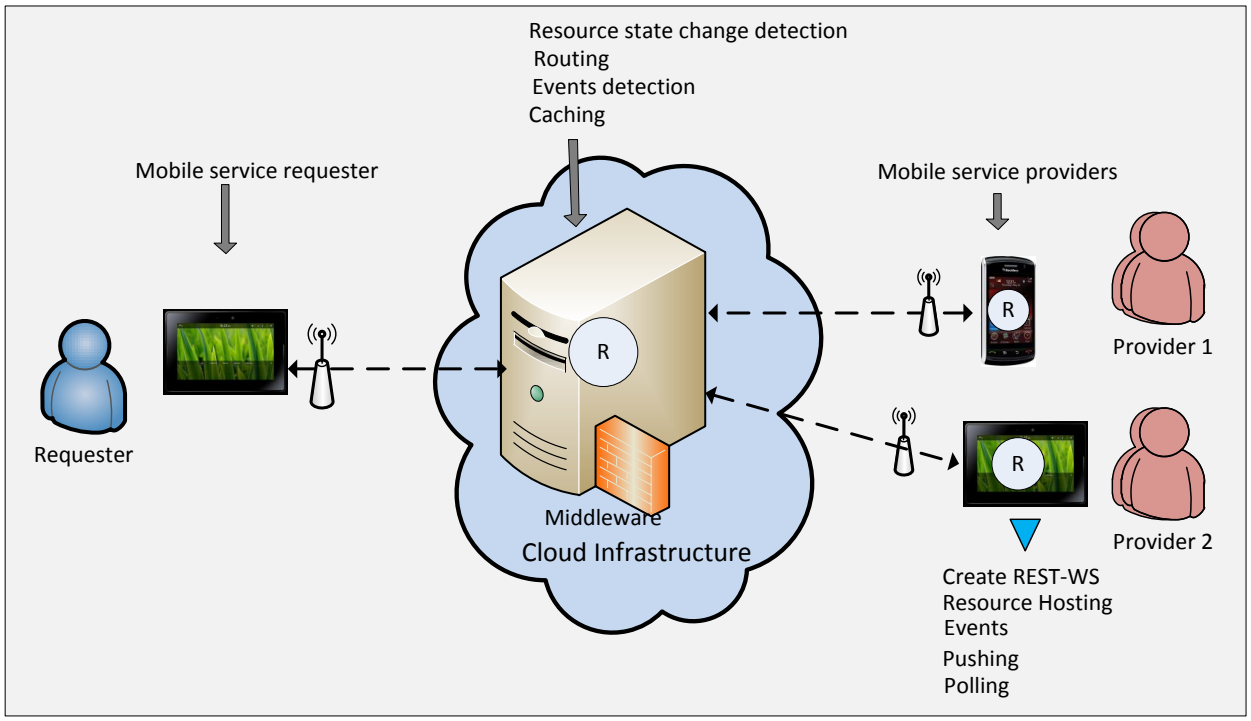


Figure 3.1: The proposed architectural design

To address the challenges of network latency and resource state synchronization as described in Chapter 1, the proposed architecture extended on some earlier research. The architecture combines techniques such as caching, cloud-hosted middleware, and REST - based

on the research by Wang [10], Beltran et al. [14] and Golding [31]. The middleware employs event notifications following the publish/subscribe model that is proposed by da Rocha et al. [66] to inform registered mobile participants of resources state changes.

The architecture also addresses the issue of resources state synchronization by means of the read-your-writes consistency mechanism (described in Section 2.1.3). By this model, all the write requests of a mobile provider will be visible to both the mobile provider and other participants on the next read request. In addition, the identification of which mobile service providers/requesters have the latest updates of a resource can be traced from the middleware's registry. Thus, the mobile participants that must be updated can make read requests to the mobile provider who has the latest update of a resource.

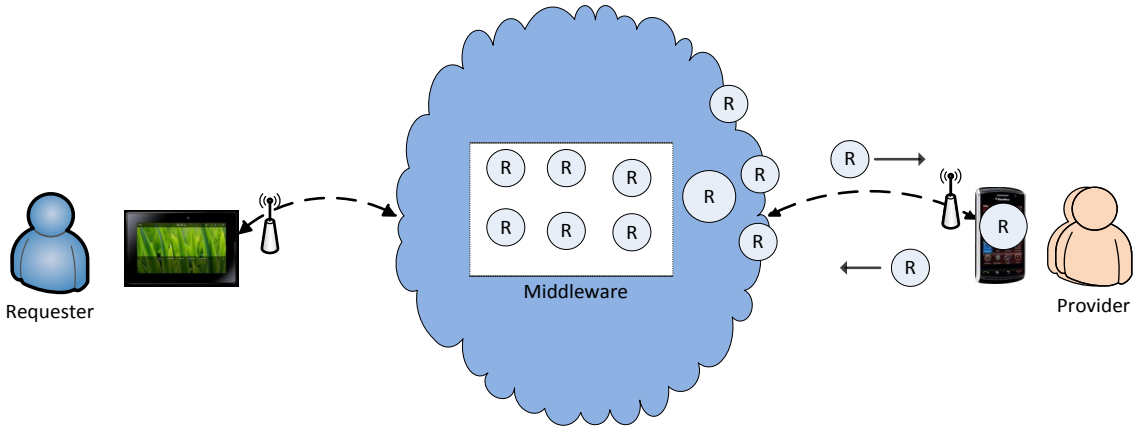


Figure 3.2: Resource replication

As illustrated in Figure 3.2, there are times when some resources, labeled *R*, are outside the cloud or they might be in the cloud but they are unreachable due to high latency. Thus, the architecture employs a resources replication technique. The resources on the mobile providers are replicated on the middleware; and the replicas are identical but they exist independently of each other. The replication approach is good for minimizing latency because the designed middleware responds to client requests faster. Furthermore, the replication technique is important

for ensuring high availability of the RESTful Web resources since the connectivity of the mobile providers cannot be guaranteed in the Wi-Fi network. In case a particular mobile provider is unreachable, the requester can contact the middleware and request for that provider's resources. As a result, the middleware also acts as a back-up for the mobile providers.

Also, the communications between the mobile service requesters are directed through the middleware; which shields all activities of the mobile service providers. In order to keep resources state updated for real-time delivery of service to the mobile requester, the mobile provider uses a *long-polling* technique to fetch updates from the middleware. With long-polling, when a client sends a request to the middleware for an update, the middleware waits for an update to arrive in case there is no update before it responds to the client. Additionally, the mobile providers employ pushing techniques to notify the middleware of resources state changes.

For a complete interaction between the mobile service requester and the provider, the following steps are followed.

1. The mobile requester sends a read or write request with a unique service id. A read request is sent as an HTTP GET method while a write request can be an HTTP POST, PUT or DELETE methods.
2. The middleware receives the request and extract the HTTP method and executes step 3 or 4.
3. If the HTTP method is GET, the middleware pushes the read request to the specified mobile provider. If the provider is reachable, it searches through its local storage for the particular service id and returns the response to the requester through the middleware. However, if the mobile provider is not reachable, the middleware searches for that service

id in its cache and return the resource to the mobile requester with the message that the mobile provider is temporary not available. In case the two client devices are disconnected, the entire request is cancelled.

4. If the request is a POST method, the middleware requests the creation of the requested resource or service on the mobile provider. The middleware stores a copy of the service and notifies all clients that have subscribed to the system about the arrival of a new update. All disconnected clients will be notified of the write request when they reconnect.

Apart from the above described scenario, a mobile service provider can access its own resources in the local cache. If the user of a mobile service provider updates a resource or creates a new resource, a copy of the resource is first stored locally before the middleware is notified to apply the update to its cache.

3.1.1 Network Latency

The cloud-based middleware serves as a router for all the asynchronous messaging between the mobile service requesters and the mobile service providers. All the information of the mobile provider is logged in the middleware cache for callbacks; based on the proposed framework by Golding [31] on achieving eventual consistency. The middleware, therefore, uses pushing to send updates to the mobile participants that have subscribed for services, as and when they arrive in real-time. The middleware storage facility also serves as a repository for the mobile clients to read data that might be with other disconnected mobile providers. Thus, the sending of read/write requests to mobile hosts that are disconnected is prevented.

3.1.2 Resource State Synchronization

Due to the intermittent loss in connectivity in a Wi-Fi network, the architecture employs the asynchronous messaging mechanism for all communications. The synchronous messaging technique does not lend itself well in mobile distributed systems because propagated state changes from the middleware or acknowledgements from the mobile hosts can get lost. Hence, our proposed resources update management cannot follow the update technique proposed by Golding [31] which follows synchronous messaging in a wired network. The middleware therefore, extended on Golding's [31] research to propagate new state changes to subscribed mobile clients in a Wi-Fi network. Furthermore, only updated resources are pushed from the middleware to the mobile participants that re-connect.

The architecture also uses read-your-write consistency based on the report by Monash [29] that this consistency technique works best for NoSQL back-end systems. All updates applied to resource states are visible to the requester on the next read request. The architecture also uses timestamps to ensure reliability of the data. The new states are pushed to all clients within an inconsistency window so that eventually, all clients will receive the update. For instance, if a provider sends a resources state change message to another provider/requester, the latter will update its local cache with the new state and only the new state will be seen subsequently on every read request. If the request from the provider cannot reach the requester because the latter is disconnected, the proposed middleware will notify the provider with temporal unavailability response. The next section discusses the middleware and how all the functionalities are achieved.

3.2 Middleware Platform/Framework

The cloud hosted middleware acts as a proxy between the mobile service requesters and the mobile service providers to control read/write replicas. The middleware is centralized in order to minimize the challenges of data synchronization that otherwise could have arisen if the middleware is distributed. The middleware redirects the HTTP requests from the requester and responses from the service providers.

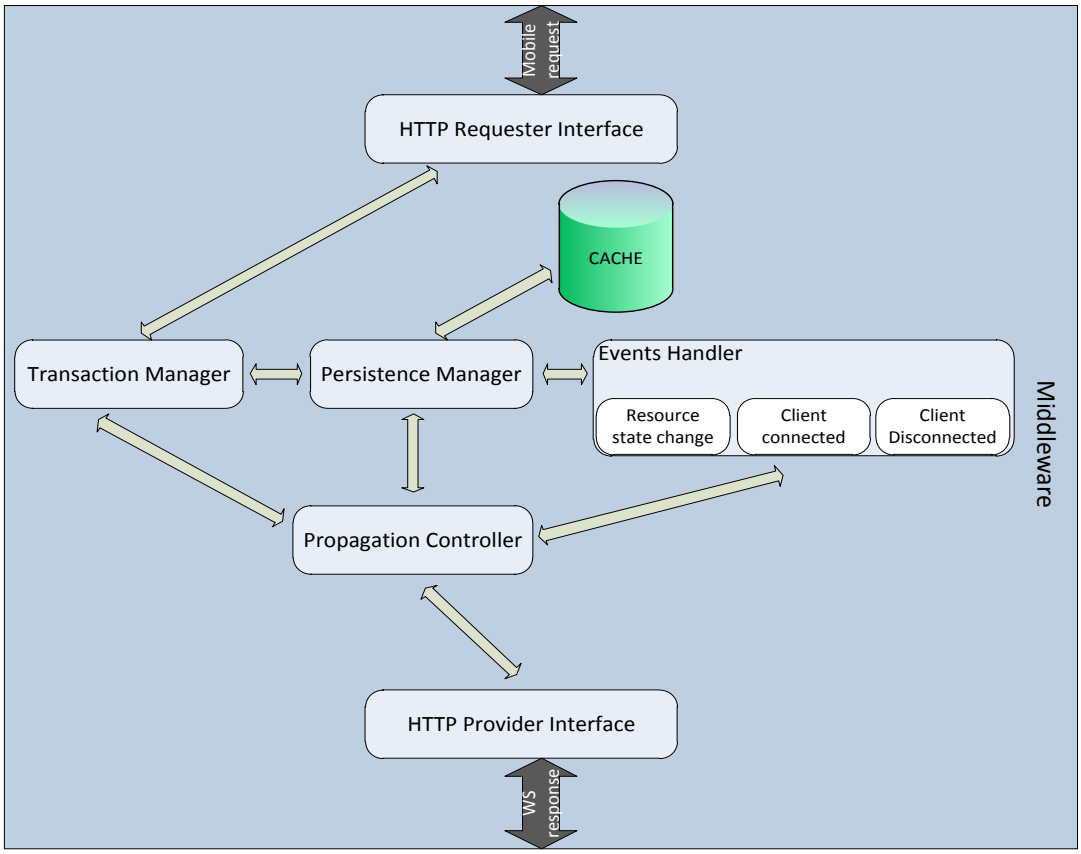


Figure 3.3: Middleware features

The internal framework of the middleware is shown in Figure 3.3. When an HTTP method is invoked from a mobile participant, the request first identifies the HTTP requester interface of the middleware through a specified port. All requests are then forwarded to the transaction manager which determines the nature of the request (e.g. read or write request). The

transaction manager communicates with the propagation controller to notify the mobile host/provider of the request through the HTTP provider interface. The response of the provider is sent back to the requester through the transaction manager. All responses are also stored in the cache by the persistence manager. In case the propagation controller responds that the specified provider is unavailable, the persistence manager searches through the cache and return the result. The cache stores replicas of the resources that are created on the mobile service provider. Also, the cache is structured to store every user's record in individual tables. In addition, anytime that the provider's resources are updated, the updates are pushed to the middleware through the HTTP provider interface. The updates are then forwarded by the propagation controller to the persistence manager for it to update the cache. All events are determined by the event handler for appropriate notification (e.g. resource state change, client connected, and client disconnected).

The main functionalities of the middleware are discussed below.

- **Read/Write Request** - The middleware controls the race condition between a read request and a write request that are issued at the same time. This is achieved by introducing timestamps on each request. The transaction manager determines which request comes first and processes that request with a higher priority.
- **Caching** – Since REST affords stateless communication between system components, resources on the mobile provider are replicated independently on the middleware. The persistence manager controls the storage of resources updates from the mobile provider. The caching component stores states of mobile service providers that subscribe for services and also stores state of resources as well. The cache has a RESTful interface that supports the CRUD methods. This means that the content of the cache can be created, read, updated, and deleted. All interactions with the cache are handled by the persistence manager. The

transaction manager requests from the persistence manager for a particular resource in its cache through a read method. The propagation controller also sends a request to the persistence manager for the creation of a service or an update of a service state through a write request.

- **Routing** - All communications are routed through the middleware because some mobile participants can disconnect due to network instability. In such cases, other clients who want to access resources and services from disconnected mobile nodes can access them from the middleware. The propagation controller acts as a reverse proxy by using the long-polling technique to detect state changes of resources on the mobile provider. The propagation controller also informs the persistence manager to store the new state of the resource or service in its local cache. The propagation controller then uses pushing techniques to inform the transaction manager to propagate the new state of resources to subscribed service providers and requesters via HTTP.
- **Events** - The other duty of the middleware is managing events. The events that the events handler is managing are related to state changes in resources, disconnection of a client device, and connection of client service providers.
    - Resource state change: When updates arrive in the cache, the new states of resources override the older versions in the cache. The events handler then informs the propagation controller to propagate the changes to all mobile service providers who have subscribed.
    - Connection of a client: The events handler notifies the propagation controller of all connected clients so that updates will be sent to them as and when they arrive in real-time.

- Disconnection of a client: Updates are pushed to disconnected clients when they reconnect. The events handler notifies the propagation controller to send only updated services that these disconnected clients don't have.

The other desired properties of the middleware are fault-tolerance and scalability. Hosting the middleware on the cloud services will facilitate such results. Since the entire client requests are routed through the middleware, the number of requests that get to the mobile providers directly reduces. Hence, the workload on the entire system will be evenly distributed on the "powerful computer" in the cloud. In the next section, the mobile side functionalities are explained.

3.3 Mobile Side Framework

The mobile side framework consists of the mobile service requesters and the mobile service providers/hosts. The requesters are the users plus the devices that send HTTP messages to the middleware and the provider either for a resource to be created or to fetch an existing resource. The mobile service providers on the other hand are the users plus devices that host the RESTful Web services. In our architecture, the mobile service providers behave as Web servers.

Recently, more developers are looking towards HTML5 mobile applications as a solution for targeting multiple devices and platforms. This is because the browser is becoming the default platform and its de facto standards are HTML5 and JavaScript [75]. Smartphones and tablets support the development of native applications which are platform specific. However, these devices also have an embedded browser which makes it possible to deploy mobile web applications. Thus, with the advancement of HTML5, a hybrid app methodology is adopted in the proposed design of the architecture to build the mobile web app that looks and functions as a native app. The new stack of HTML5 is shown in Figure 3.4.

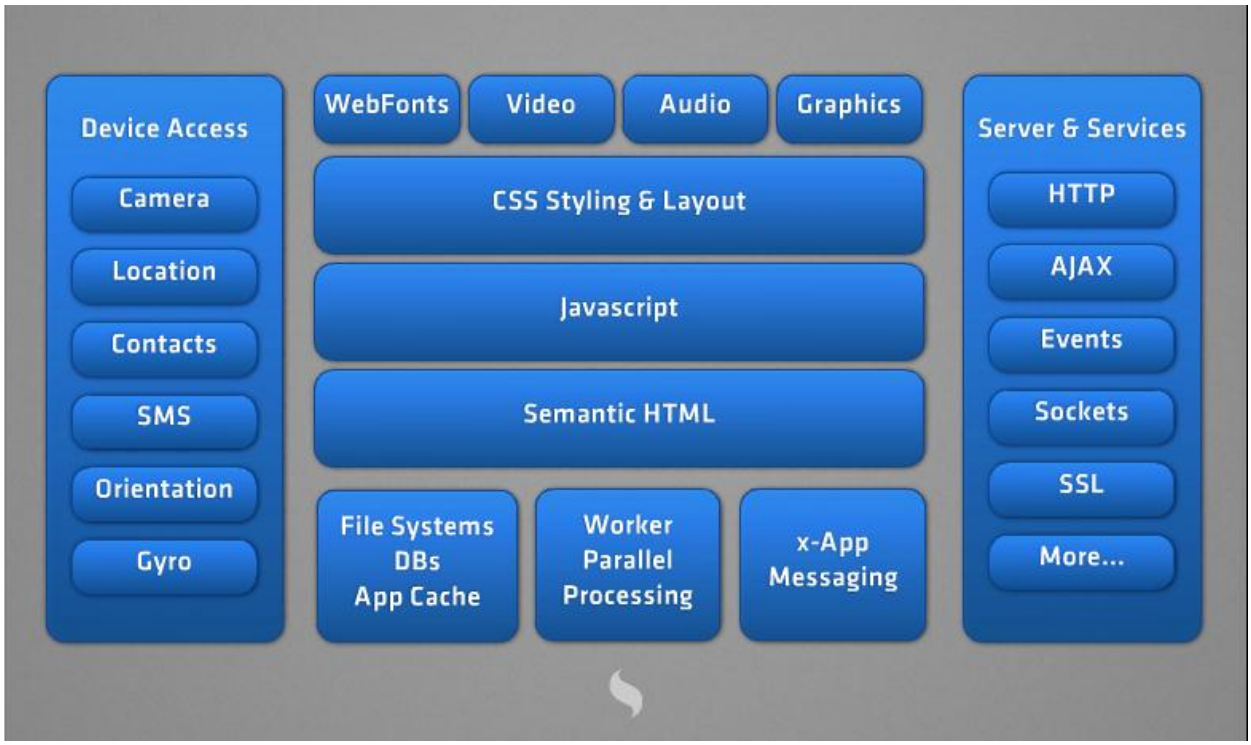


Figure 3.4: HTML5 stack on the mobile platform [76]

As illustrated in the HTML5 stack, the native functionalities that have access to the device can be imported into the mobile Web app. HTML5 also supports server and services which makes it affordable to consume Web services. Thus, the Web centric nature of REST enabled the proposed architecture to embed a Web server into the mobile device. Also, Web services can be invoked in the embedded browser regardless of the mobile device platform. Embedded browsers provide rich graphical user interfaces and support multiple web based languages such as HTML, JavaScript, CSS, and so on.

Furthermore, all interactions with the middleware are initiated from the mobile devices. The responses from the middleware are pushed to the embedded browser which displays it on the screen. The main features of the mobile requester and the provider are discussed in the next section.

3.3.1 Mobile Service Requester

The mobile service requester is a client consumer of the RESTful web services that are being propagated by the middleware and the mobile provider. Key features of the requesters' application are discussed below.

- **Client Side Caching**: Resource replication on the mobile service requesters is addressed by exploring mobile caching techniques. The mobile cache resides on the mobile device and keeps local copies of resources. The cache components are built to minimize the amount of calls that are made to the middleware. This is good for the management of bandwidth usage. The successful responses to HTTP GET requests are cached but failure attempts are not cached. Also, the client cache validation is done using HTTP HEAD and GET requests. The mobile client compares the *Etag* – which is a unique attribute of a Web resource, to determine whether the local copy of the resource is the same as the providers' resource. A change in Etag is an indication of a possible update of an existing resource. Furthermore, the last successful cached copy of the resource is pushed to the screen whenever the client is disconnected.

- **Application Execution Workflows:** The workflow is to monitor state changes of resources from the middleware and the provider. There is the tendency of having outdated resources state in the client cache especially in cases of network communication loss between the client and the middleware. Thus, new updates on the middleware might not reflect on the client side at the same time. The workflow manages the CAP theorem phenomenon by ensuring high availability over data consistency since partition tolerance cannot be avoided due to the disruptions in connectivity in a Wi-Fi environment. The mobile service requester sends a request to the middleware to determine whether a new update has arrived on the

mobile hosts. Since network connectivity is crucial in mobile technology, a periodic HTTP HEAD request is sent to the middleware for only the Etag. In addition, for short-lived disconnections between the client and the middleware, the workflow ensures distribution transparency by trying to reconnect. When the disconnection is long-lived, distribution transparency is compromised and the last successful stored update in the cache is pushed to the screen. The user is notified that the resource is from the cache and since the resources are time stamped, the user gets to know how old the data is in the cache.

3.3.2 Mobile Service Provider/Host

Though the mobile providers have the same functionalities as those of the requesters, they have additional functionalities. The idea is that the mobile provider will act as a Web server as well as a consumer.

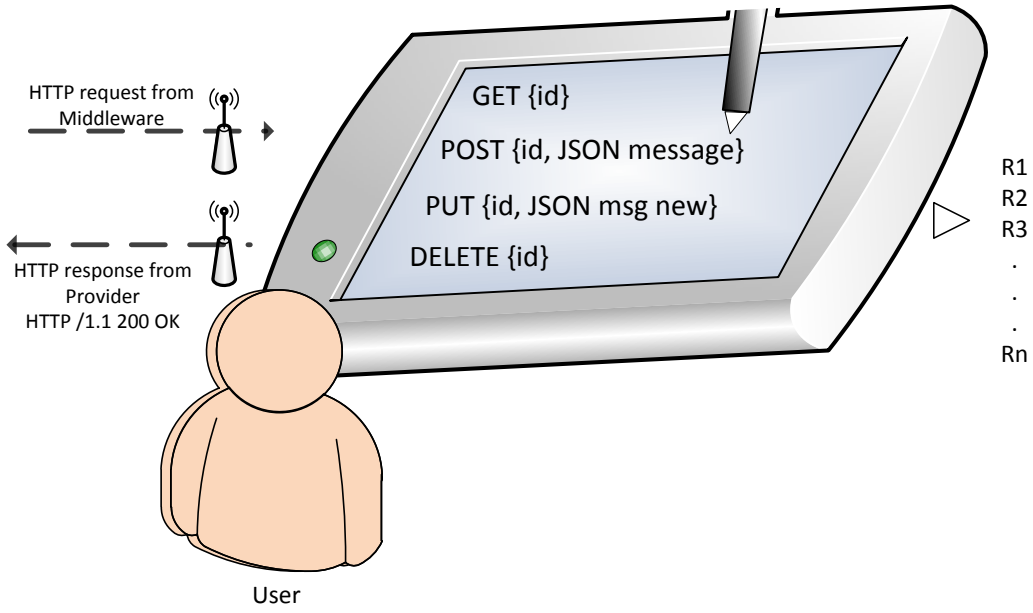


Figure 3.5: The functionalities of the mobile provider

The RESTful Web Services (REST-WS) created are hosted on the mobile provider as resources, labeled R1 to Rn in Figure 3.5. When a user sends a write request (i.e. HTTP POST

method) from the mobile device, the request is created as a REST-WS resource. The newly created resource is given a globally unique identifier which enables other mobile participants to select or search for that resource. The mobile providers' resources are accessed using URIs that are provided by the middleware. The communication protocol used is HTTP because JSON data can be passed across it.

Also, the responses from the provider make use of the HTTP status codes. This notifies the middleware on the state of a particular resource. The requester is informed whether there is an error in a request (with a 400 status code) or the request is successful (with a 200 status code). Figure 3.6 shows how the provider executes incoming requests from the middleware.

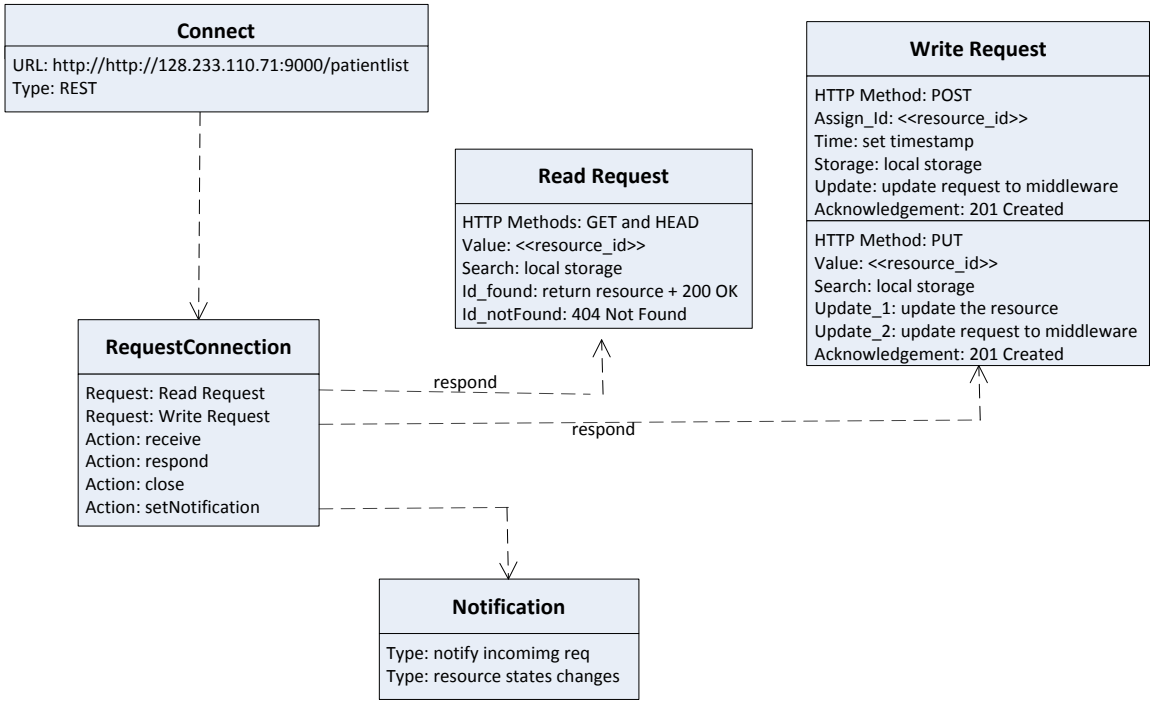


Figure 3.6: Request execution by the mobile provider

The mobile provider accepts a read request in the form of an HTTP GET method and an identifier of the specific resource. However, write requests are HTTP POST method with an identifier and JSON message for the creation of a new resource. Another write request is the

HTTP PUT method with an existing resource identifier and the new JSON message for updating an existing resource. Also, resources can be deleted with a write request if the HTTP DELETE method is invoked with a specified identifier of the resource.

When a write request is pushed with a JSON message from the provider, the request is immediately processed on the middleware and a response is sent back. The provider uses the combination of polling and pushing to synchronize data on the middleware.

3.4 Summary

An architecture has been proposed in this chapter to address the issue of network latency and the challenge of resources state management in a Wi-Fi network. The proposed architecture is comprised of the following main components: mobile service requesters, middleware, and mobile service providers.

The mobile provider hosts the RESTful Web resources; and a copy of each resource is replicated on the middleware. The replication technique is to ensure the availability of resources to the mobile service requester when the provider is unreachable. In addition, techniques such as pushing and polling are proposed to minimize latency.

Furthermore, resources state synchronization challenges are addressed by proposing the read-your-write consistency mechanism. This approach has been described in other research as a strong consistency approach especially in NoSQL database systems.

In the next chapter, an implementation is done to determine the feasibility of the architecture. The various tools that are employed in the implementation are justified and explained. Also, the integration of the tools is described.

CHAPTER 4
IMPLEMENTATION OF THE ARCHITECTURE

A prototypic E-health application, called “Patient App”, that model patient records, is built based on the proposed architecture. Using the application, health care professionals can access the patient records that are being hosted on their colleagues’ mobile devices. A health care professional can create a new patient record, request for an existing patient record, or update a record. The forthcoming discussions focus on how the implementation is done, and the justification of the various programming languages employed as well as the mobile platforms.

4.1 The Mobile Implementation

The mobile side implementation is done using HTML5 and JavaScript. Due to browser and WebKit diversity; as shown in Figure 4.1, it is impractical for all the functionalities of HTML5 to run in all browsers. In order to overcome the challenges of browser diversity, some developers proposed Web technology frameworks adoption [76]. Thus, the “Patient App” employs Web tech frameworks such as jQuerymobile [77] and jQtouch [78] to overcome the limitations of HTML5.

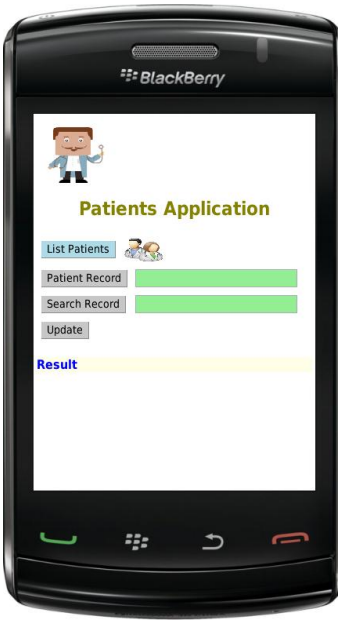
Figure 4.1: WebKit and browser diversity

Also, the frameworks aided us to: build compelling user interfaces, achieve browser interoperability, and compile the mobile Web app as a native app. The mobile platforms that are considered in the implementation are the BlackBerry Smartphone, BlackBerry Playbook and Android tablet devices. Since the mobile platforms are heterogeneous in terms of their underlying operating systems, building a native app will mean that multiple versions of the same application have to be built in different programming languages. To avoid this situation, we employed HTML5 and the Web tech frameworks to write a single code version which is deployed on the various mobile platforms.

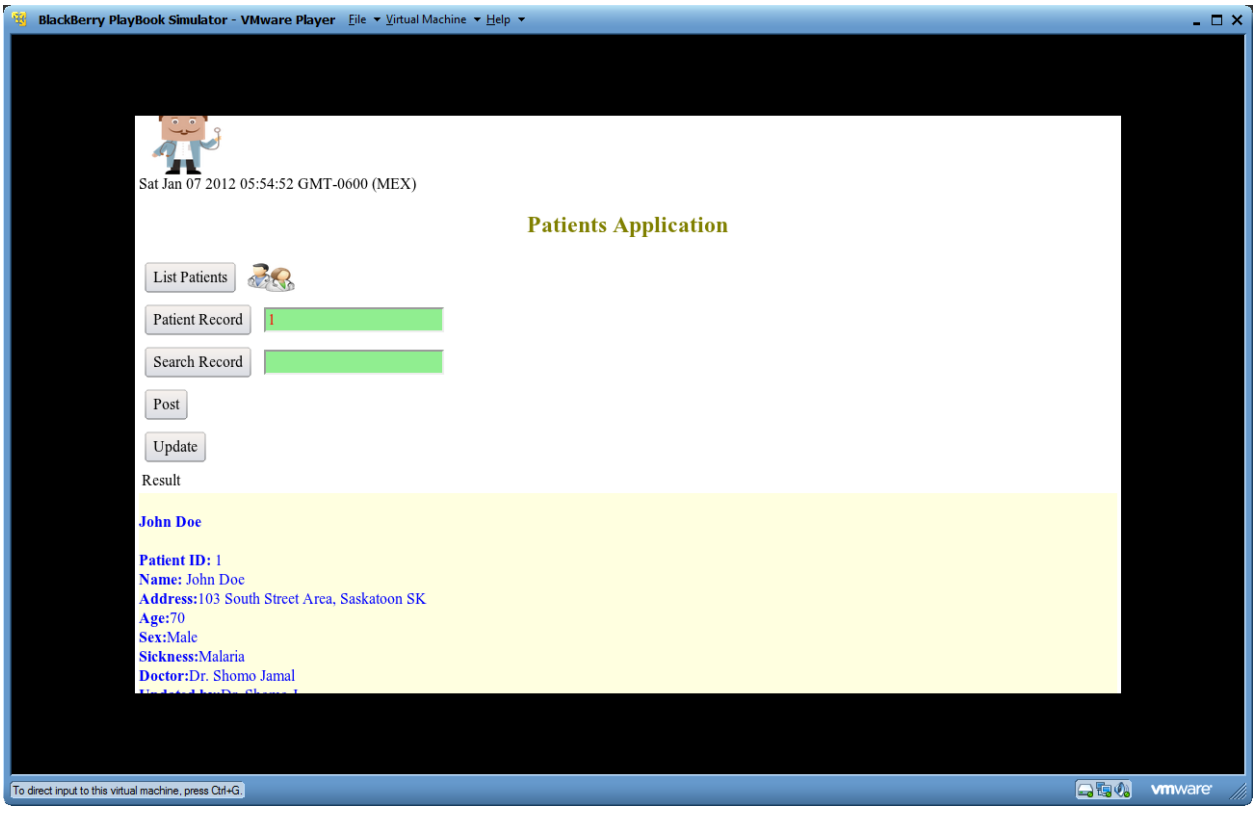
4.1.1 Mobile Client Platforms

The BlackBerry version of the application is built as a BlackBerry WebWorks project, which is invoked in the embedded browser. The Blackberry WebWorks platform supports multiple web based languages and web technologies like CSS, HTML5, and JavaScript. Using WebWorks, the jQuerymobile and jQtouch mobile web frameworks were adopted into the application. As a result of using the WebWorks platform, the single code base application is deployed successfully on the BlackBerry smartphone and the Playbook.

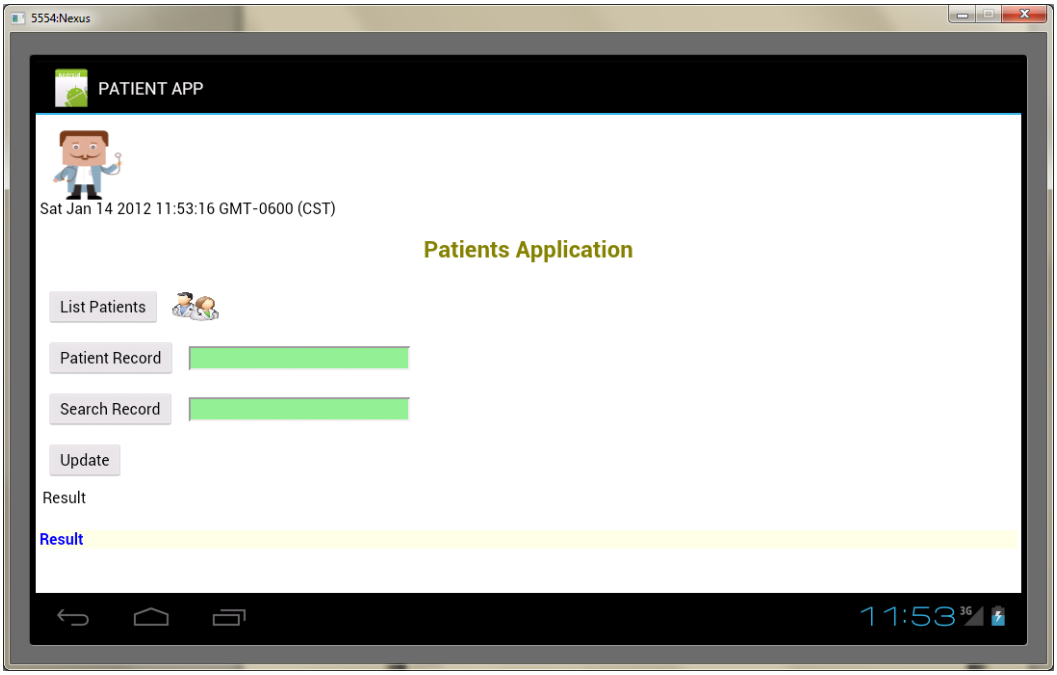
The mobile side application is classified into three components: user interface component, workflow, and cache manager. The user interface is rendered in an embedded browser but looks like a native application. Also, the use of the HTML5 frameworks, aided the application to have the same look and feel across different platforms. All responses from the middleware, which are controlled by the workflow, are pushed to the embedded browser and it displays on the screen. Figure 4.2 shows the user interface of the application on the BlackBerry smartphone, Playbook simulator, and the Android emulator. The information shown on the Playbook represents a dummy record due to privacy in the E-health domain.


BlackBerry
Patients Application
List Patients
Patient Record
Search Record
Update
Result

(a)


BlackBerry PlayBook Simulator - VMware Player
File
Virtual Machine
Help
Sat Jan 07 2012 05:54:52 GMT-0600 (MEX)
Patients Application
List Patients
Patient Record
1
Search Record
Post
Update
Result
John Doe
Patient ID: 1
Name: John Doe
Address:103 South Street Area, Saskatoon SK
Age:70
Sex:Male
Sickness:Malaria
Doctor:Dr. Shomo Jamal
To direct input to this virtual machine, press Ctrl+G.
vmware

(b)



(c)

Figure 4.2: The UI of the application rendered on the BlackBerry smartphone (a) and BlackBerry Playbook simulator (b), and Android emulator (c)

Since the BlackBerry version of the application is implemented in WebWorks, the same code was reused for the compilation of the Android WebView project. The WebView is a framework that supports the deployment of mobile web applications on the Android devices. The use of HTML5 in the BlackBerry implementation facilitated the integration of the app on the Android tablet without writing additional codes.

In terms of resources manipulation, the application uses the following HTTP methods: HEAD, GET, and PUT. The requests are created using the XMLHttpRequest() class provided in JavaScript. As a result, users can make asynchronous requests to the middleware through the interaction with the UI. The workflow coordinates the HTTP requests and responses to determine whether a resource state has changed. A periodic HEAD operation is invoked by the workflow to the middleware and the Etag value of the response is examined to determine whether updates

have arrived on the middleware. In addition, the GET and HEAD methods use the HTTP 200 OK response code to inform the workflow of the success of a request-response interaction. The HTTP 400 response is sent to the client if there is a problem with the request that the middleware doesn't understand.

Furthermore, the mobile side application has a local cache where all the RESTful Web resources are stored. When the user makes a request to the middleware and the middleware cannot be reached, the workflow searches through the local cache and pushes the cached data to the screen with a clear message that the middleware is temporarily unavailable. The timestamp on each data informs the user of the "age" of the data. Also, the validation of the cache is done through the comparison of the resources Etag values. Whenever, an update is applied to a resource on the middleware or the mobile provider, the cache validation function is invoked and the new state of the resource is stored in the cache to replace the outdated resource. Figure 4.3 shows a code snippet of the validation function in JavaScript.

```
function validate(){
        var addr = document.getElementById("readRecord").value;
        var seturl = "http://128.233.110.172:9000/patientdb/" + addr;

        var request = new XMLHttpRequest();
            request.open("GET", seturl, true);
            request.onreadystatechange = function() {
            if(request.readyState == 4 && request.status == 200) {
              var Etag = request.getResponseHeader("Etag");

              if (rvalidate.get(Etag) == undefined)
                 {
                   rvalidate.set(Etag, addr);     //Store the Etag
                   rid.update(addr, jsonObject); // Update Cache
                 }
              else
                 {
                  alert("Resource " + addr + " is updated");
                 }
```

Figure 4.3: Code snippet of the validate function

Also, the workflow coordinates all the activities on the mobile client as well as controlling the request-response interaction between the client and the middleware. The architecture of the mobile implementation is shown in Figure 4.4. Though the embedded browser approach facilitated the deployment of a single code on the BlackBerry and the Android devices, these mobile platforms have certain files that are platform specific. For instance, on the BlackBerry platform, the WebWorks application cannot be deployed without the *config.xml* file. This file - which is the configuration file, defines the permissions, the attributes, and the application name.

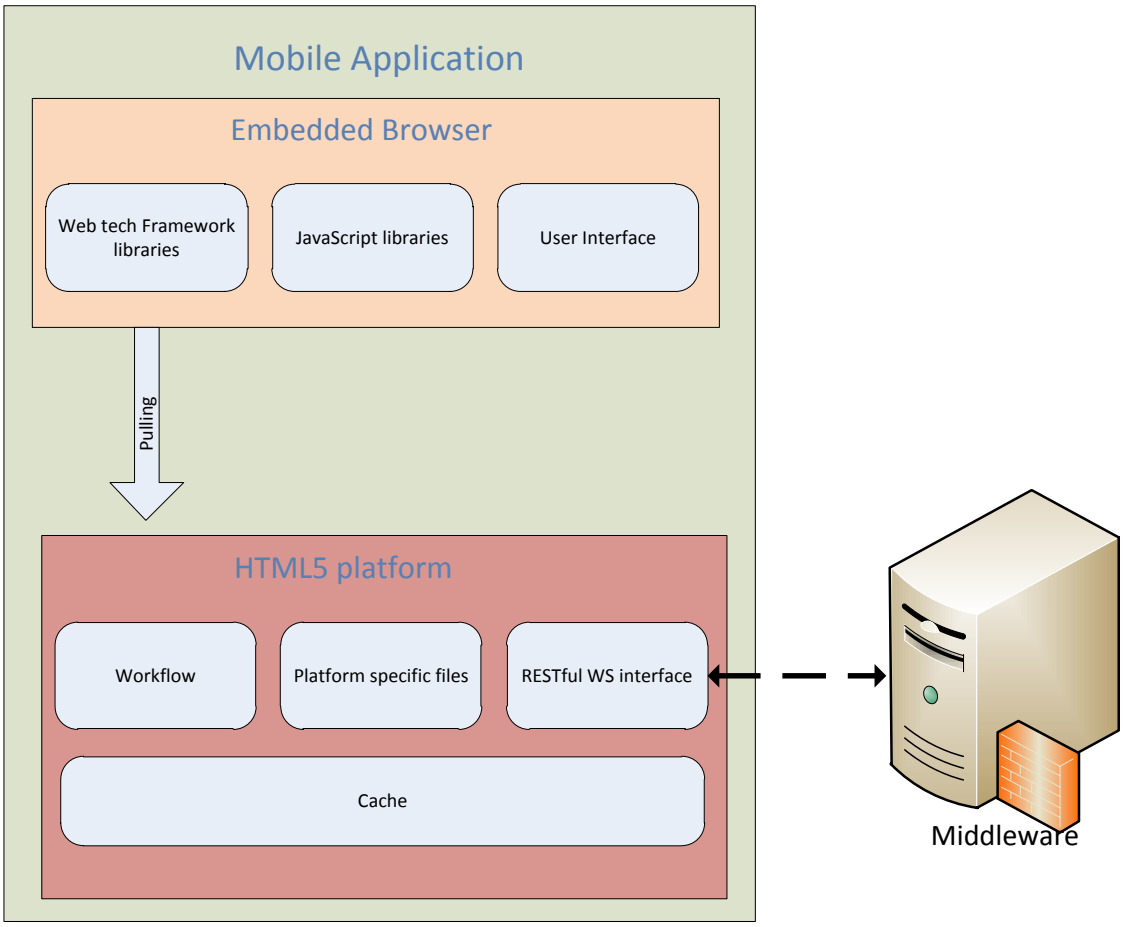


Figure 4.4: Design of the mobile app

4.1.2 Enabling the Mobile Device as a Provider

To model the mobile device as a host of the RESTful Web resources, the providers' application is implemented with a tool called Couchbase Mobile [79]. The Couchbase Mobile framework supports the local storage of the RESTful resources on the mobile device; and it also facilitates resource synchronization with NoSQL back-end systems in real-time. Also, the Couchbase Mobile framework supports data in JSON format which makes it efficient for the mobile consumption.

However, the Internet Protocol (IP) address of a mobile device changes as the user moves from one hotspot to another. In addition, registering the device name within a network domain is not ideal because the user can move from one network domain to a different domain. In view of these challenges, the Couchbase Mobile framework is configured to establish a dual communication channel between the mobile device and the middleware. Thus, the mobile provider can push RESTful resources to the middleware or poll resources from the middleware. In addition, the mobile requester can send an HTTP request to the provider using the IP address or the registered computer name of the middleware – which is running in the cloud.

In situations of unavailable connectivity, the user can create resources on the mobile provider and these resources are stored and queued in the Couchbase Mobile framework. Requests that will be arriving on the middleware also get queued in the order in which they arrive. Whenever the connection is restored, the Couchbase Mobile establishes a connection with the middleware which allows the data on the provider to be pushed to the middleware and the data on the middleware to be pulled by the mobile provider.

4.2 Middleware Implementation

In our architecture, the middleware acts like a hub; linking the entire mobile participants and does message routing from one mobile node to another. In view of this, two of the leading programming platforms that support backend systems for mobile communication are explored to implement two versions of the middleware. These programming platforms are Erlang [80] and CouchDB [81]. The programming platforms have NoSQL database systems which are good for enforcing read-your-write consistency [29]. In addition, they support concurrent requests and database distributions. The implementation of the two versions of the middleware using the various programming platforms is discussed in the next sections.

4.2.1 Middleware Implementation in Erlang

The Erlang middleware is built on the Generic Server Behaviour (gen_server) process. The gen_server is a module that enables the implementation of a server that supports request-response interaction between a client and a server. Thus, the implemented middleware has two HTTP interfaces. The first HTTP interface is exposed to the mobile requester who uses a set of HTTP methods to send RESTful requests. The second HTTP interface allows the mobile provider to continuously push data from its local storage to the middleware. To receive a request from the mobile participants, the middleware uses the httpc module. The module supports the HTTP/1.1 companionable clients and facilitates our middleware to use the following HTTP methods: HEAD, GET, PUT, and POST. There are other HTTP methods such as TRACE, OPTIONS, and DELETE that the httpc module supports though they were not used in our implementation.

Also, the middleware uses the DETS storage facility of Erlang to keep the replicas of the providers' resources. The DETS aided the middleware to store data on the disk as objects. The

middleware storage is designed to consume the RESTful resources and as a result, the CRUD operations are supported. Figure 4.5 shows write and read operations in the DETS table.

Every user has a DETS table where all the user activities are stored. All established communications are recorded with the time of the communication. This aids the middleware to report to a mobile service requester on the reliability of the replica data on the middleware.

```
insert(Key, Value) ->                  %% POST or PUT command
    case cache: lookup(Key) of     %% Checks if key already exists
        {ok, Pid} ->
            element: replace(Pid, Value) ;
        {error, _} ->
            {ok, Pid} = element: create(Value) ,
            cache: insert(Key, Pid)
    end.

lookup(Key) ->                              %% read command
    try
        { ok, Pid} = cache:lookup(Key) ,   %% Fetches pid for keys
        { ok, Value} = element:fetch( Pid) ,
        { ok, Value}
    catch
        _Class:_Exception ->
            {error, not_found}
    end.
```

Figure 4.5: Read and Write operations in the DETS table

Also, the gen_server module has functionalities for error reporting and debugging. Thus, failure requests can be traced as well as preventing unsupported operations from users. The steps that the middleware follows to complete a request-response interaction between a mobile service provider and a requester are illustrated in Figure 4.6. If the provider is not reachable, steps 2, 3, and 4 will fail so in such situations, the middleware pushes the providers' resource in the DETS to the requester to accomplish step 5.

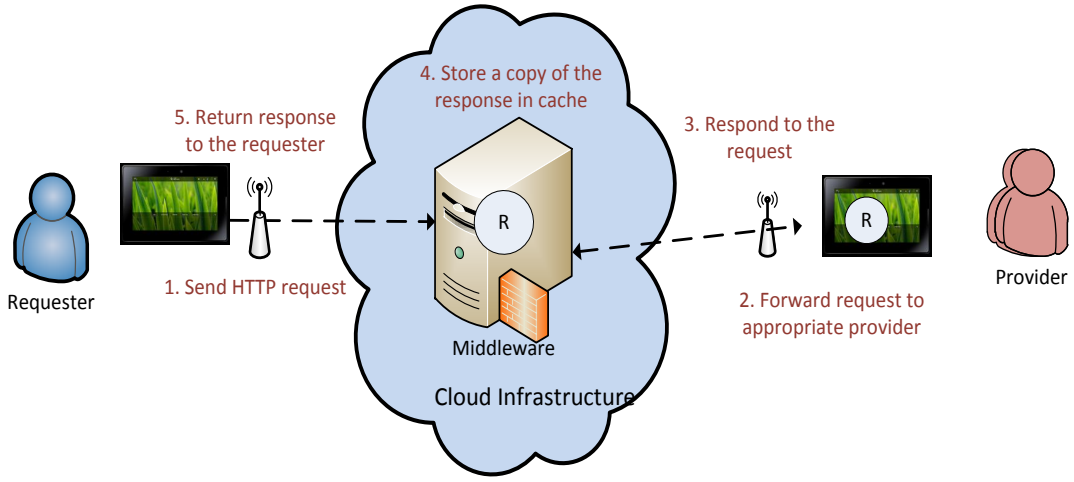


Figure 4.6: The middleware as the coordinator of the request-response interactions

4.2.2 Middleware implementation in CouchDB

The second version of the middleware is built on the Apache CouchDB [81] framework. CouchDB supports RESTful requests through HTTP interfaces. The CouchDB framework is a document-centric database that supports data in JSON format only. Furthermore, a unique feature of CouchDB is the ability to do database replication on multiple systems.

The CouchDB middleware synchronizes the data on the middleware with the Couchbase Mobile framework that is running on the mobile provider in real time. When the synchronization process is terminated, upon reconnection, the synchronization process resumes form where it last stopped.

Also, the CouchDB middleware has two HTTP interfaces that are exposed to the mobile participants. All the interactions in the system are RESTful and as result, the CRUD operations are supported as well. The only issue in the CouchDB implementation is that the HTTP POST method is not supported at the moment. Thus, the HTTP PUT method is use for updating an existing record or creating a new record. Each document has a revision number set by CouchDB to track the changes that are being made to the document.

Furthermore, the CouchDB framework provides views for querying the database. The views are created using JavaScript functions to output the type of data that the user wants to see. Figure 4.7 shows a JSON data of a patient exported from the CouchDB middleware.

```
{
"_id":"2","_
rev":"4-095e72000ca093d49b49b7992a3f185e",
"name":"Richard Lomo",
"address":"104 South Lane, Regina, SK",
"age":"60",
"sickness":"Fever",
"Personal doctor":"Dr. Andrew R",
"updated by":"Dr. Rich"
"last update": "October 15, 2011"
}
```

Figure 4.7: Record from CouchDB

4.3 Known Issues

The implementation of the architecture focuses on enabling the mobile device as host of providers and consumers of RESTful Web services. Though the chosen domain for the app development is E-health, some challenges within the E-health domain have been overlooked in this thesis; hence will be investigated in another work.

Firstly, E-health relies on integrating multiple health information infrastructures which are sometimes operated by different health care providers. As a result, it is a challenge to integrate new applications into the existing health information systems. In view of this, we deployed the middleware on an independent computer platform to support the mobile requesters and the providers. The integration of the middleware into the existing heath information system is outside the scope of this thesis.

Also, individual patients attend different health care facilities in different locations. Additionally, many health care professionals can attend to the same patient in different

hospitals/clinics. However, in our implementation, the aggregation of all of the patient's data into one has not been looked into. This is also considered outside the scope of our research.

4.4 Summary

In Chapter 4, a prototypic application that aids health care professionals to share the data of their patients with colleagues is built. The mobile-side of the application employs HTML5 and Web tech frameworks which enabled the deployment of a hybrid application (i.e. mobile Web app that looks and has the same functionalities as a native app). Also the application is cross-platform independent hence, was deployed successfully on BlackBerry Playbook, BlackBerry smartphone OS 5.0, and Android tablet devices.

Furthermore, two versions of the middleware were implemented using the leading concurrency languages for building NoSQL systems, Erlang and CouchDB. The middleware acts as a proxy and redirect all requests from a mobile service requester to the appropriate mobile service provider. In the next section, an evaluation of the implementation is conducted to determine how latency is minimized and how resources state changes are managed.

CHAPTER 5
EXPERIMENTS

The BlackBerry smartphone OS 5.0, BlackBerry Playbook tablet, and Android tablet device are the mobile devices put forward for the testing and simulation in the experiments. The middleware is hosted on the Amazon EC2 cloud infrastructure. The approaches adopted for the evaluation of the problems (Section 1.2) presented in this thesis are fault-injection and calculation of overheads.

All the experiments are simulated in order to enable the observer (i.e. the person conducting the experiments) to have more control over the environment. The experiments focus on how the implemented architecture minimizes latency; so the overhead introduced by the middleware is calculated. Since there are two versions of the middleware, that is the Erlang version and the CouchDB version, they are evaluated separately. Also, the experiments justify the resources replication technique that is employed in the architecture.

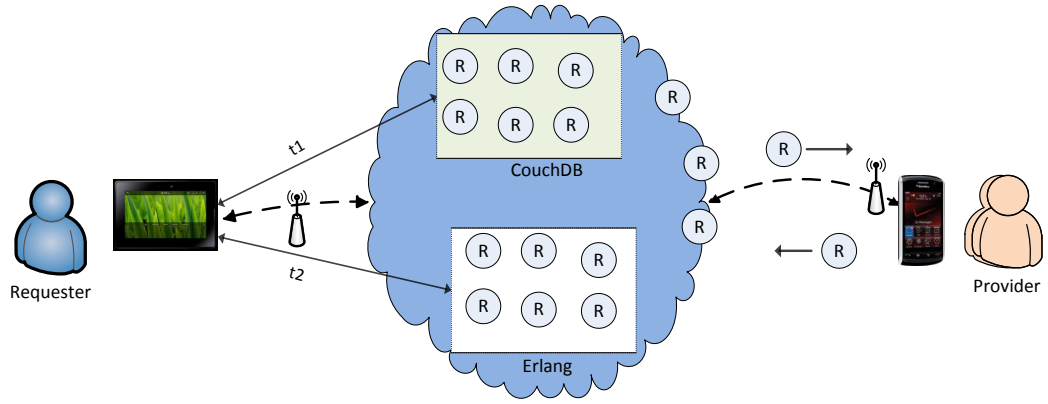


Figure 5.1: Times taken to access resources through the Erlang middleware and the CouchDB middleware

Figure 5.1 illustrates the time, *t1*, taken to access a resource, *R*, from the CouchDB middleware or from the provider through the CouchDB middleware is measured. Also, the time, *t2*, taken to access the providers' resources through the Erlang middleware or the replicated resources on the Erlang middleware is measured. The two durations, *t1* and *t2***,** are compared to

determine which one provides a faster time. Furthermore, the accessibility of resources is also measured by observing how long it takes for an update to be visible to all the mobile devices that are connected.

The remaining sections of this chapter explain the details of the experimental setups and the workloads used. Also, the various tools that are used are explained and full discussions of the results are presented.

5.1 System Requirements and Experiment Goals

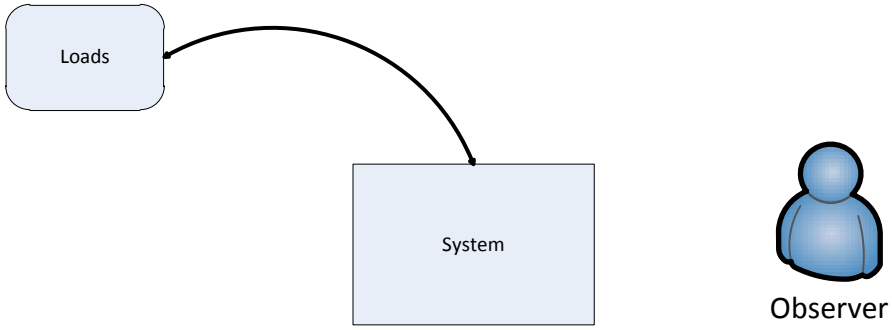


Figure 5.2: Experimental Setup

The mobile devices under consideration are hosted outside the cloud. The factors that influence client-server interaction such as network latency, scalability and reliability in Wi-Fi networks are measured and the empirical data is recorded by an observer (Figure 5.2). In a situation where heavy workloads are needed, simulation is done with tools such as Apache bench [82] and nodeload [83]. The BlackBerry Playbook simulator which is running as the requester is setup on a computer with the following specifications in the laboratory.

Processor: Intel Core i5

CPU 650 @ 3.20GHz

3.19GHz

RAM: 4GB (2.99GB usable)

System 32-bit operating system

Also, the Android Galaxy tablet emulator which is running as the provider is setup on a computer with the following specifications in the laboratory.

Processor: Intel Core i5

CPU 650 @ 3.22GHz

3.22GHz

RAM: 4GB (2.99GB usable)

System 32-bit operating system

The Erlang middleware and the CouchDB version are hosted on the computer with the following specifications (identical to the Amazon EC2 cloud infrastructure).

Windows Edition:

Windows 7 Enterprise

Service Pack 1

System:

Processor: Intel(R) Xeon(R)

CPU E5140 @ 2.33GHz 2.33GHz (2 processors)

Installed memory (RAM): 16.00GB

System type: 64-bit Operating System

The following is an outline of the experiments to be conducted to determine how the implemented architecture deals with latency and resources state change management (section 2).

Goal 1: Testing resource accessibility on the resources' hosts

Experiment 1: Accessibility of RESTful resources by the requester on the mobile provider using the CouchDB middleware as an intermediary proxy.

Experiment 2: Accessibility of the RESTful resources by the requester on the mobile provider using the Erlang middleware as an intermediary proxy.

Experiment 3: Accessibility of the RESTful resources by the mobile requester on the CouchDB storage.

Experiment 4: Accessibility of the RESTful resources by the mobile requester on the Erlang middleware storage (DETS table).

Goal 2: Scalability testing of the middleware platforms

Experiment 5: Scalability of the CouchDB middleware platform.

Experiment 6: Scalability of the Erlang middleware platform.

Goal 3: Determining the inconsistency window of our system

Goal 4: Fault injection approaches to determine system responses

5.2 Evaluation of Overhead

The evaluation of the system focuses on the latency overhead introduced by the middleware. The latency is measured based on the response times in a requester-provider interaction. The reason for this experiment is to test how slow/fast it takes to receive responses to requests in a Wi-Fi network. In addition another evaluation of latency is conducted in a client-server interaction to determine which version of the middleware responds faster to mobile requests. The durations for read-write requests are observed with the Erlang middleware and the CouchDB middleware as shown in the sections below. Also, the scalability test is run to determine how much workload the middleware can handle.

5.2.1 Resource Accessibility from the Mobile Provider by a Requester

The experiment on how accessible the RESTful Web resources are on the mobile service provider is conducted to determine the latency. The assumption in this experiment is that, the mobile service provider is available and reachable. The test at this stage is carried out by measuring the round-trip duration for an HTTP GET request from the mobile requester to the mobile provider through the middleware. As shown in Figure 5.3, an observer sends an HTTP GET request from the mobile requester to the mobile provider through the middleware which is hosted in the University LAN and transfers data through a gigabit Ethernet connection. The mobile requester (running on a BlackBerry Playbook simulator) and the provider (running on an Android Galaxy tablet emulator) connect through the University of Saskatchewan secure Wi-Fi network using 802.11g. The RESTful Web resources, R, representing dummy patients records, are hosted on the Android tablet emulator.

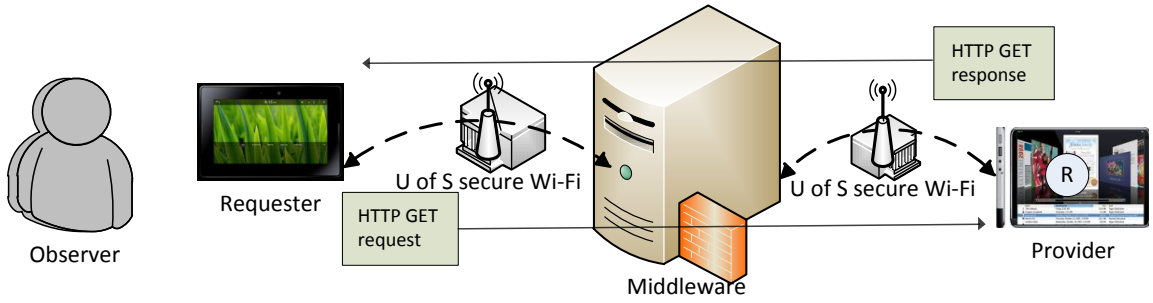


Figure 5.3: Setup for testing resource accessibility on the provider

At the beginning of the experiment, 600 RESTful Web resources are created and hosted on the mobile provider with no resources on the middleware. We hosted 600 RESTful Web resources due to the device constraint. The idea in the testing at this point is to determine how long it takes to access a providers' record that is not on the middleware. Furthermore, each response from the provider is stored on the middleware before being delivered to the mobile

requester. The file size of each resource is approximately 5kb and all the HTTP GET requests are sent in a closed loop. In addition, an HTTP request is sent sequentially after the preceding request-response interaction is complete.

Also, to be sure that the requests have not been served from the middleware's cache, the requests are sent to fetch Web resources that have odd numbered ids (e.g. id=1, id=3, id=55 …). The mean round-trip time is calculated. Also, to further understand how the values are dispersed from the mean round-trip time, the standard deviation (σ) is calculated using the formula below.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where x_i is individual request, μ is the mean of all requests, and N is the number of requests.

Furthermore, if N number of requests from the mobile service requester are served in time T, then the

$$Maximum\ Request\ Rate = N/T$$

5.2.1.1 Experiment 1: Measuring Round-trip Time through CouchDB

The first evaluation of the resource accessibility on the mobile provider focused on using the CouchDB version of the middleware. The mobile requester sends HTTP GET requests to the CouchDB middleware which does the message routing to the mobile provider. The response message from the mobile provider is stored on the CouchDB middleware before finally it is delivered to the mobile requester.

The tests are repeated five (5) times on each round starting from 1 request to 25 requests. The result of the round-trip time between the mobile service requester and the mobile service provider through the CouchDB middleware is presented in Table 5.1. Also, the maximum number of requests, sequentially in a closed loop, from the mobile requester is observed to be 25

(i.e. all the requests beyond 25 failed due to the limitation of the processing capacity of the mobile host).

Table 5.1: Request-response duration through CouchDB

Mean request-response time for sending 25 requests (ms)	Maximum request-response time for sending 25 requests (ms)	Minimum request-response time for sending 25 requests (ms)	Standard deviation
503.68	817.20	212.2	165.99

5.2.1.2 Experiment 2: Measuring Round-trip Time through Erlang

The second analysis of the resource accessibility on the mobile service provider focused on using the Erlang middleware. The mobile requester sends the HTTP GET requests to the mobile service provider through the Erlang middleware which acts as a proxy between the two mobile components. The response from the mobile provider is stored on the Erlang DETS table before finally it is delivered to the mobile requester.

The tests are repeated five (5) times on each round starting from 1 request to 25 requests. The result of the round-trip test between the mobile service requester and the mobile service provider through the Erlang middleware is presented in Table 5.2. Also, the maximum number of requests from the mobile requester is observed to be 25 (i.e. all the requests beyond 25 failed due to the limitation of the processing capacity of the mobile host).

Table 5.2: Request-response duration through Erlang

Mean request-response time for sending 25 requests (ms)	Maximum request-response time for sending 25 requests (ms)	Minimum request-response time for sending 25 requests (ms)	Standard deviation
563.24	889.60	319.40	158.76

5.2.1.3 Discussion of Experiment 1 and Experiment 2

The results from Experiment 1 and Experiment 2 give us an idea about how the middleware influences latency in mobile distributed systems. The two experiments focused on how accessible RESTful Web resources are on the mobile service provider. Hence, Figure 5.4 shows the graph of the average duration for a request-response interaction through the middleware in both Experiments. Also, Table 5.3 shows a comparative analysis of the two experiments based on the maximum request rate.

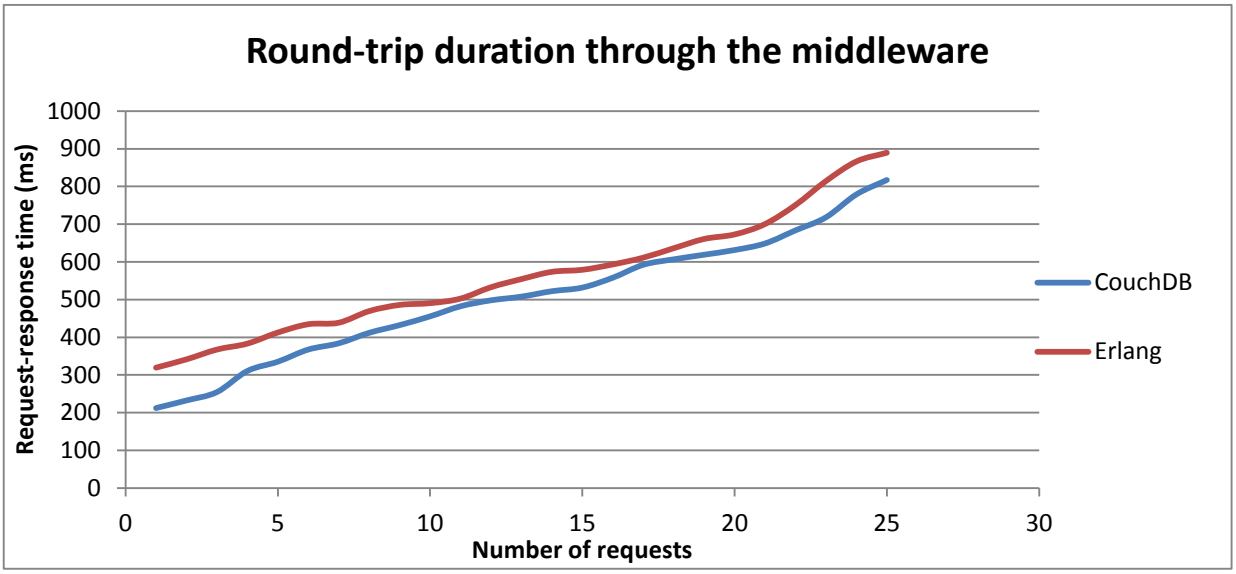


Figure 5.4: Graph of the requester-provider interaction through the middleware

Table 5.3: Results comparing Experiment 1 and Experiment 2

Middleware platform	Average request-response time for sending 25 requests (ms)	Standard deviation	Maximum request rate (request/s)
CouchDB	503.68	165.99	49.63
Erlang	563.24	158.76	44.39

The results show that the CouchDB version of the middleware serves the mobile client requester 11.80% more requests than the Erlang middleware under the same conditions. In

addition, the CouchDB middleware responds faster than the Erlang middleware considering the fact that the CouchDB middleware shows a lower average request-response time. A possible reason for the better outcome for the CouchDB middleware is the fact that the mobile providers' resources are stored in Couchbase Mobile. As a result, CouchDB establishes a two-way consistency replication technique with the providers' storage to synchronize the data without the intervention of the developer. However, the way the Erlang middleware is designed in the implementation, it is only when a requester wants to contact the provider that the middleware attempts to fetch the data from the provider. Thus, it takes some time for the data to be synchronized with the DETS table.

Also, finding that the maximum number of requests that the requester can make in a closed loop is 25, aids us to understand the processing workload of the mobile provider under consideration in this thesis. The middleware therefore has to be designed to keep the number of concurrent requests to the mobile provider below 25. Furthermore, the results show that it takes an average of 0.5 seconds for a requester to receive response from the provider through the CouchDB version of the middleware; while 0.56 seconds is the time for the Erlang middleware. In both cases, the time is appreciable considering the fact that the requester is served in less than 1 second.

The assumption in the Experiments 1 and 2 is that, the provider is always within the reach of the requester. However, the reality is there are times when the provider cannot be reached due to unstable Wi-Fi connections. In such situations, the requester is served from the middleware's storage. The next section focuses on the evaluation of accessing the RESTful Web resources from the middleware.

5.2.2 Resource Accessibility on the Middleware

The evaluation of the RESTful Web resources accessibility on the middleware is based on the assumption that the mobile service provider is disconnected and cannot be reached by the requester. Thus it is the responsibility of the middleware to serve the mobile service requester from the middleware storage. The set-up for the experiment is illustrated in Figure 5.5.

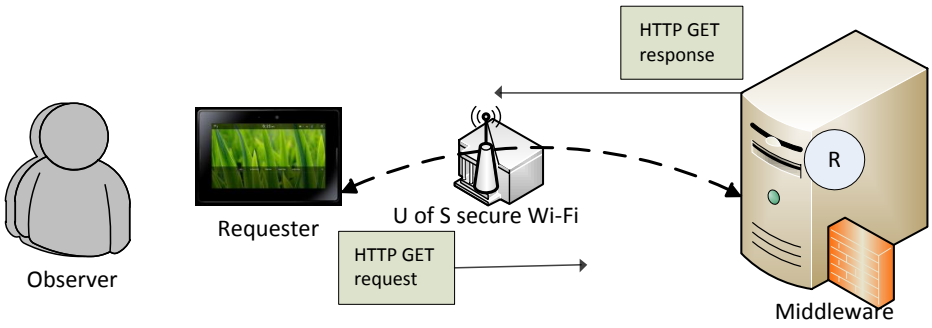


Figure 5.5: Set-up for the evaluation of resource accessibility on the middleware.

In order to explore the full capacity of the mobile tablet device, a request mechanism is implemented on the mobile device that sends two (2) concurrent HTTP GET requests in a closed loop. The request mechanism is achieved by using the setTime() and setTimeout() functions in JavaScript. When the first HTTP request is sent, the setTime() function is called which records the start time. When the second request is sent, the setTime() function records the start time of the second request as well. The two requests end with the setTimeout() function being called and the request completion times are recorded. The average time to complete the two requests is considered as the duration for that number of requests.

In addition, the HTTP GET request from the mobile requester is not sent to the middleware to fetch only one resource; but for multiple resources. To control how this is done, from the two concurrent requests, the first concurrent request is sent to fetch Web resources that have odd numbered ids (e.g. id=1, id=3, id=55 …). The second concurrent request is sent to fetch

Web resources that have even numbered ids (e.g. id=2, id=4, id=56 …). The reason for accessing resources based on the nature of the id is to aid the observer to repeat the experiment multiple times. The experiments are repeated five (5) times starting from 1 request to 70 requests. The time taken for the entire process to complete is recorded and analyzed in Experiment 3 and Experiment 4.

5.2.2.1 Experiment 3: Accessing RESTful Resources on the CouchDB Middleware

The experiment at this stage is carried out by measuring the duration for the CouchDB middleware to serve a mobile service requester with RESTful resources from the middleware storage. The mobile service requester sends an HTTP GET request to the CouchDB to fetch the replicated resources that are on the middleware.

The CouchDB storage is initially populated with 10000 records of RESTful resources, representing dummy patients' records. The size of each patient record is approximately 5kb. The HTTP GET interaction between the mobile service requester and the CouchDB middleware is shown in Figure 5.6 for a patient data with an id 1. An HTTP request is sent after the preceding request-response interaction is complete.

HTTP GET Request
GET id=1 HTTP/1.1
Host: xoxo.usask.ca
Accept: Application/json

CouchDB Middleware Response
HTTP/1.1 200 OK
Server: CouchDB/1.0.2 (Erlang OTP/R13B)
Etag: "9-a981e1f8d8ba2fcf86f088b98119e374"
Content-Type: application/json
Content-Length: 298

{"_id":"1","_rev":"9-a981e1f8d8ba2fcf86f088b98119e374","name":"Agnes Kwadzo","address":"103 Cumberland, Saskatoon SK","sickness":"Malaria","doctor":"Dr. Richard J","age":70,"updated":"Dr. Shomo J","_attachments":{"patient1-002.jpg":{"content_type":"image/jpeg","revpos":3,"length":2077,"stub":true}}}

Figure 5.6: Request response data between the mobile device and CouchDB

The maximum number of requests that the mobile service requester, running on the Blackberry Playbook, could make is observed to be seventy (70); and the requests are sequential. All the requests beyond 70 return an error due to the processing constraint of the mobile requester. The mean round trip time is calculated and recorded as shown in Table 5.4. The standard deviation from the mean is also calculated.

Table 5.4: Resource accessibility on the CouchDB middleware

Mean request-response time for sending 70 requests (ms)	Maximum request-response time for sending 70 requests (ms)	Minimum request-response time for sending 70 requests (ms)	Standard deviation
397.39	751.60	27.40	195.15

5.2.2.2 Experiment 4: Accessing RESTful resources on the Erlang Middleware

The evaluation in Experiment 3 is emulated in Experiment 4 with the Erlang middleware. Thus, resources which were on the CouchDB middleware have been replicated on the Erlang middleware. The middleware cache is built in Erlang DETS - which supports read requests. The HTTP GET interaction between the mobile service requester and the Erlang middleware is shown in Figure 5.7 for a patient data with an id 1.

HTTP GET Request
GET id=1 HTTP/1.1
Host: Semeru.usask.ca
Accept: Application/json

Erlang Middleware Response
HTTP/1.1 200 OK
Server: Erlang OTP/R13B
Etag: "9-a981e1f8d8ba2fcf86f088b98119e374"
Content-Type: application/json
Content-Length: 298

{"_id":"1","_rev":"9-a981e1f8d8ba2fcf86f088b98119e374","name":"Agnes Kwadzo","address":"103 Cumberland, Saskatoon SK","sickness":"Malaria","doctor":"Dr. Richard J","age":70,"updated":"Dr. Shomo J","_attachments":{"patient1-002.jpg":{"content_type":"image/jpeg","revpos":3,"length":2077,"stub":true}}}

Figure 5.7: Request response data between the mobile device and the Erlang middleware

Since the resources hosted on the CouchDB middleware are identical to those replicated on the Erlang middleware, the properties such as the file size is the same (i.e. 5kb). Also, the maximum number of requests that the mobile service requester, running on the Blackberry Playbook, could make in a closed loop is observed to be seventy (70). All the requests beyond 70 return an error as well due to the processing constraint of the mobile requester. The requests are sent sequentially. The result from Experiment 4 is recorded in Table 5.5 below.

Table 5.5: Resource accessibility on the Erlang middleware

Mean request-response time for sending 70 requests (ms)	Maximum request-response time for sending 70 requests (ms)	Minimum request-response time for sending 70 requests (ms)	Standard deviation
366.85	689.60	40.40	183.35

5.2.2.3 Discussion of Results from Experiment 3 and Experiment 4

The results from Experiment 3 and Experiment 4 are analyzed and a comparison is made to understand how fast a mobile requester/provider can be served from the back-end system. Thus, Table 5.3 presents a comparative analysis of the two experiments based on the maximum request rate. The averages are also plotted as illustrated in Figure 5.8.

Table 5.6: Results for Web resources accessibility on the middleware

Middleware platform	Average request-response time for sending 70 requests (ms)	Standard deviation	Maximum request rate (request/s)
CouchDB	397.39	195.15	176.15
Erlang	366.85	183.35	190.81

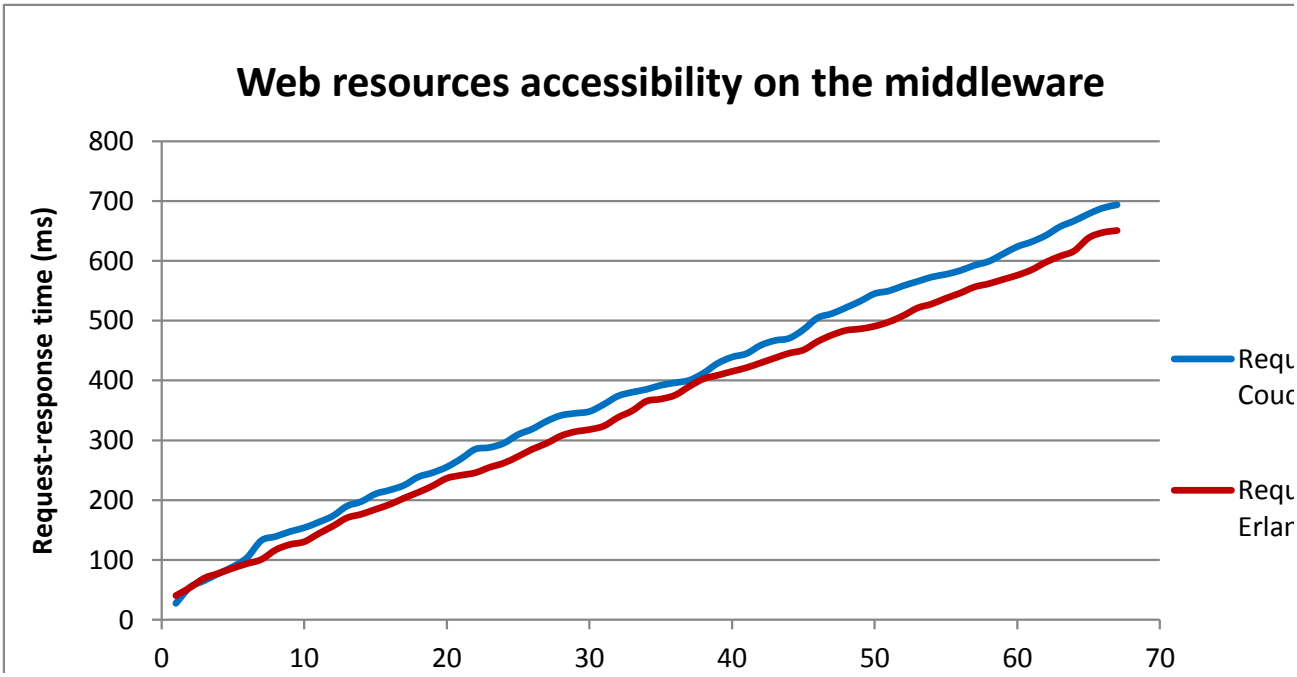


Figure 5.8: Web resources accessibility on the middleware

The results show that the Erlang middleware serves the mobile client requester/provider 8.32% more requests than the CouchDB middleware under the same conditions. In addition, the Erlang middleware responds faster to the client request whenever the provider is not available.

Also, the fact that the mobile tablet device can make a maximum of 70 requests in a closed loop gives us an idea about how many updates can be read on the mobile device within a particular duration. In cases of loss of connectivity between a health care professional's mobile device and the backend systems, patient records can be created/updated on the middleware by other health care professionals. Disconnected mobile devices that reconnect later can read up to seventy (70) requests sequentially in a closed loop. This information is vital in determining the best way of modeling the mobile side application in order not to overload the mobile tablet device with requests especially when the user is disconnected for a long period.

Though the result shows that the Erlang middleware facilitates faster request-response time, the percentage difference is not much compared to the CouchDB middleware. The reason is that, CouchDB database also runs on Erlang Open Telecom Platform (OTP) framework.

Since the entire communication between the mobile requesters and providers are routed through the middleware, the middleware should be able to handle large amount of concurrent requests. In view of this, a scalability test is proposed since generally, the performance of systems slows down drastically during peak loads.

5.2.3 Testing for Scalability

In order to determine the performance of the middleware when the users as well as the users' requests increase, the scalability test is conducted. Since Experiments 1, 2, 3 and 4 focused on the interaction between a single requester and a single provider, the full potential of the middleware could not be determined. Thus, a load generating tool called Apache Bench [82] is used as the client to send concurrent HTTP requests to the middleware for resources at a controlled rate. The load generating tool is installed on a computer with the specifications in section 5.1 in the laboratory. The set-up for the load capacity testing of the middleware is illustrated in Figure 5.9.

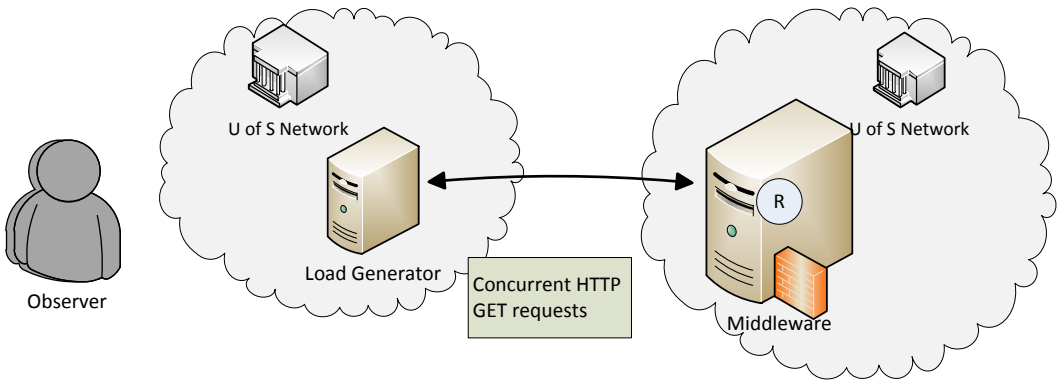


Figure 5.9: Set-up for the scalability test

The computer on which the load generator is running connects to the middleware's computer through a Gigabit Ethernet connection. The load generator is configured to simulate the activities of 500 concurrent users of the system. The number of concurrent HTTP requests that the users can send ranges from 500 to 40000. The Apache Bench tool is also configured to return the mean throughput (i.e. request per second) every 10 seconds.

5.2.3.1 Experiment 5: CouchDB Middleware Load Testing

The load generator is configured to send concurrent HTTP requests to the CouchDB middleware to consume the RESTful resources that reside on the middleware. The RESTful resources are represented as patients' demographic records, as illustrated in the JSON data in Figure 5.10. The size of each Web resource is 5kb. Also, the load generator sends requests at a rate of 1 request per 10 seconds. The rate at which the request is sent follows the exponential distribution of mean, 0.1 requests/second.


```
{
"First Name":"Richard", "Last Name":"Kwadzo",
"Phone":"(306) 888 8888", "Email":"lomoteyr@myemail.com",
 "Address":"103 Cumberland, Saskatoon SK",
"Room Number":"345", "Top Diagnosis":"Malaria", "Doctor":"Dr. Richard J",
"Age":70, "Emergency Contact":"Dr. Shomo J",
"_attachments":{"patient1-
002.jpg":{"content_type":"image/jpeg","revpos":3,"length":2077,"stub":true}}
}
```


Figure 5.10: Patient Demographic record

The result of the performance of the CouchDB middleware is presented in Table 5.7.

Table 5.7: Outcome of the CouchDB middleware's performance

Mean Throughput (req/s)	Maximum Throughput (req/s)	Minimum Throughput (req/s)	Standard deviation
124.23	135.20	115.51	4.90

5.2.3.2 Experiment 6: Erlang Middleware Load Testing

The experiment here focuses on the scalability of the Erlang middleware. The load generator sends concurrent requests to the middleware for the RESTful resources which are stored in the DETS table. These Web resources are identical but independent of the ones on the CouchDB middleware, hence are of the same JSON format as shown in Figure 5.10. Also, the size of each Web resource is 5kb. The request rate follows the exponential distribution of mean, 0.1 requests/second.

The result of the performance of the Erlang middleware is presented in Table 5.8 below.

Table 5.8: Outcome of the Erlang middleware's performance

Mean Throughput (req/s)	Maximum Throughput (req/s)	Minimum Throughput (req/s)	Standard deviation
320.42	378.70	275.60	28.89

5.2.3.3 Discussions of Results from Experiment 5 and Experiment 6

The results from Experiment 5 and Experiment 6 are illustrated in the graph shown in Figure 5.11.

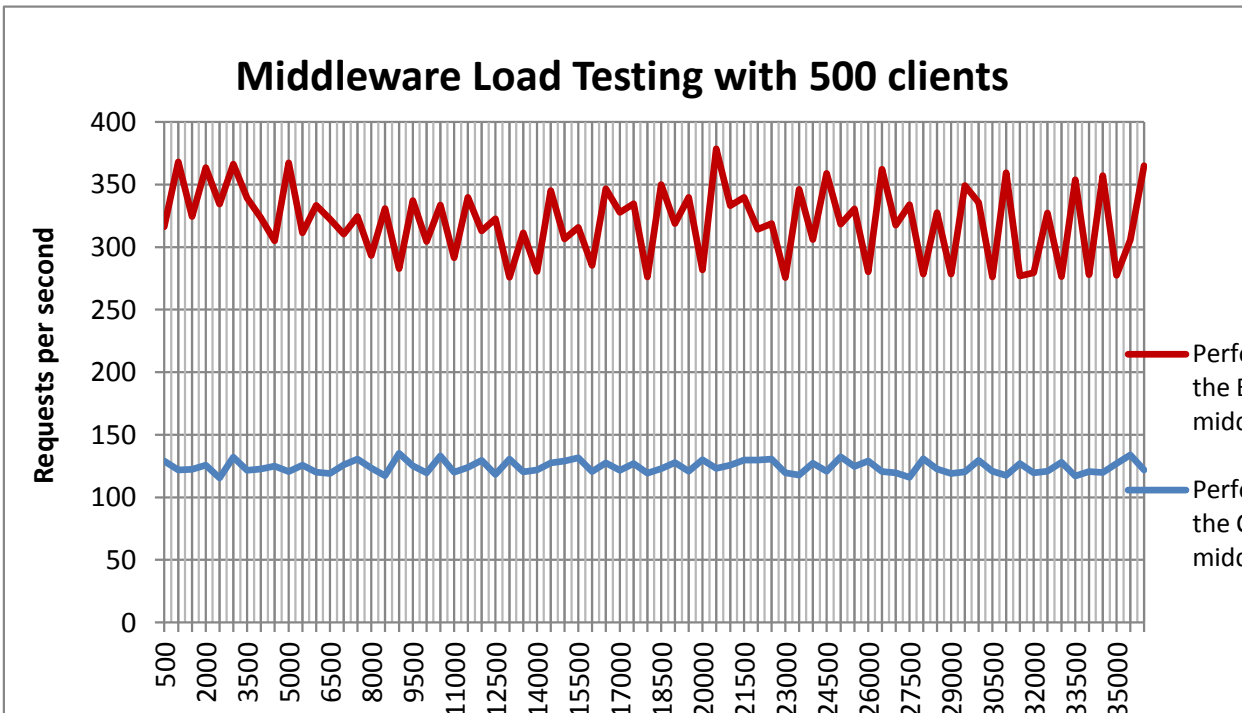


Figure 5.11: The performance of the two middleware

Again, the outcome from the scalability testing shows that under the same workload, the Erlang middleware is more efficient. From the analysis, the Erlang middleware produced a percentage increase of 157.92% throughput in comparison with the CouchDB version. The outcome shows that the Erlang middleware responds to the mobile requests faster than the CouchDB counterpart. In view of this, it is justifiable to do the RESTful Web resources replication on the Erlang middleware as a way of minimizing latency.

Also, during the testing in Experiments 5 and 6, it is observed that both middleware versions respond successfully to all the concurrent HTTP requests ranging from 500 to 40000. This proves that the middleware is reliable and the cloud environment for hosting the data is highly available. Though the maximum number of requests used for the analysis is 40000 requests, the Erlang middleware can handle more requests beyond this number. The limitation however is that, it takes a longer time (almost half an hour) for a request to be served for all

requests beyond 40000. The CouchDB middleware on the other hand, crashes after 40000 requests.

5.3 Determining the Inconsistency Window (Using Read and Write Mix)

The read and write mix experiment focuses on determining the inconsistency window of the designed architecture. The inconsistency window is the time frame within which updated Web resources will be visible to all connected mobile participants. The eventual consistency technique that is employed in the architecture is the read-your write consistency. Additionally, the read-your-writes consistency technique is described as a strong consistency model for NoSQL backend systems [29]. Thus, the model was adopted in our implementation since the Erlang DETS storage and CouchDB storage are NoSQL oriented. The setup for the test is illustrated in Figure 5.12.

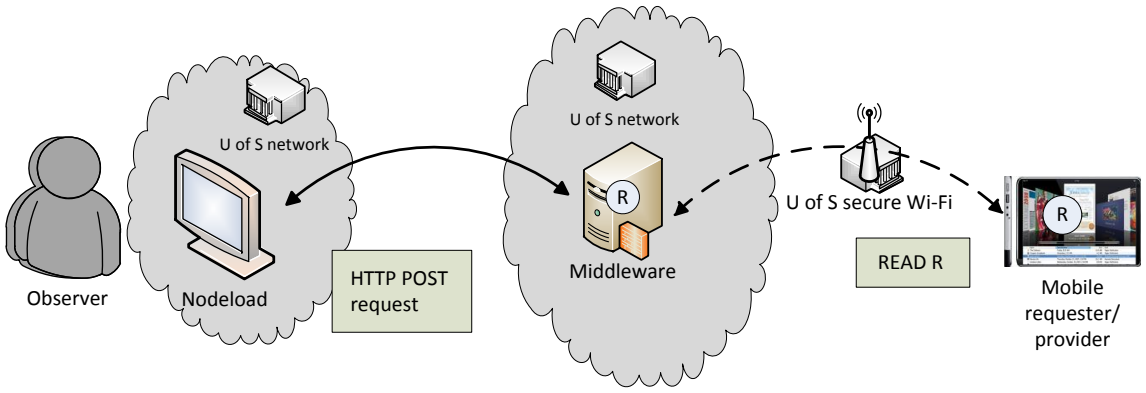


Figure 5.12: Setup for determining the inconsistency window

In conducting the inconsistency window testing, a tool called nodeload [83] is installed on a desktop computer in the laboratory. Nodeload is a JavaScript based tool that generates HTTP read and write requests in a closed loop. In our experiment, the nodeload tool is used for generating HTTP POST (i.e. write) requests to the middleware. A sample HTTP POST request

to the Erlang middleware is shown in Figure 5.13. The same write request works for the CouchDB middleware but uses the HTTP PUT method to create a new Web resource.

The middleware creates a copy of the request as RESTful resources and forwards the newly created resource to the intended mobile participant. The mobile device uses continuous polling technique to read the newly created resources from the middleware.

```
POST patientdb/1 HTTP/1.1
Host: Semeru.usask.ca
Content-Length: 94
Content-Type: application/json

{"First Name":"Richard", "Last Name":"Kwadzo", "Phone":"(306) 888 8888",
"Email":"lomoteyr@myemail.com",  "Address":"103 Cumberland, Saskatoon SK",
"Room Number":"345", "Top Diagnosis":"Malaria", "Doctor":"Dr. Richard J", "Age":70,
"Emergency Contact":"Dr. Shomo J", "_attachments":{"patient1-
002.jpg":{"content_type":"image/jpeg","revpos":3,"length":2077,"stub":true}}}
```

Figure 5.13: Sample POST request to create a patient record

The experiment started with the CouchDB middleware as a proxy between Nodeload and the mobile device. No initial resources were created on the middleware or the mobile provider. The Nodeload tool is configured to send a maximum of 500 HTTP POST requests sequentially in a closed loop to the middleware. The experiment used 500 requests as the benchmark because it was observed that sending more than 500 write requests returns an error (failed request). Each Web resource is approximately 5kb in size.

The observer started the experiment by writing 500 resources (i.e. 100% write) to the middleware before the connection is established between the mobile device and the middleware. The mobile device fetches the successfully created resources from the middleware using HTTP GET request. The time taken by the mobile device to read all the requests is recorded. The experiment is then repeated by creating 450 initial write requests (i.e. 90% write). The mobile device establishes the connection to start reading the resources while the Nodeload tool

completes the remaining 50 write requests. The total time to read all the 500 requests is recorded. The experiment is repeated by reducing the initial write request while recording the total time to read all the requests as shown in Table 5.9.

Table 5.9: Read and write mix

Write Requests	Read Requests
500 (i.e. 100%)	Start Reading
450 (i.e. 90%)	Start Reading
400 (i.e. 80%)	Start Reading
350 (i.e. 70%)	Start Reading
300 (i.e. 60%)	Start Reading
250 (i.e. 50%)	Start Reading
200 (i.e. 40%)	Start Reading
150 (i.e. 30%)	Start Reading
100 (i.e. 20%)	Start Reading
50 (i.e. 10%)	Start Reading
0 (i.e.0%)	Start Reading

The experiment is repeated with the Erlang version of the middleware acting as the proxy between Nodeload and the mobile device. The result from the read and write mix experiment is presented in Table 5.10 below. The variables that are considered in this test are the mean inconsistency window, the maximum inconsistency window, and the minimum inconsistency window.

Table 5.10: Result of the inconsistency window testing

Middleware platform	Mean inconsistency window (ms)	Maximum inconsistency window (ms)	Minimum inconsistency window (ms)
CouchDB	94.95	165.60	25.00
Erlang	90.11	160.20	18.40

The graph in Figure 5.14 illustrates the durations for each round of the read/write mixed experiments using the two versions of the middleware as proxies.

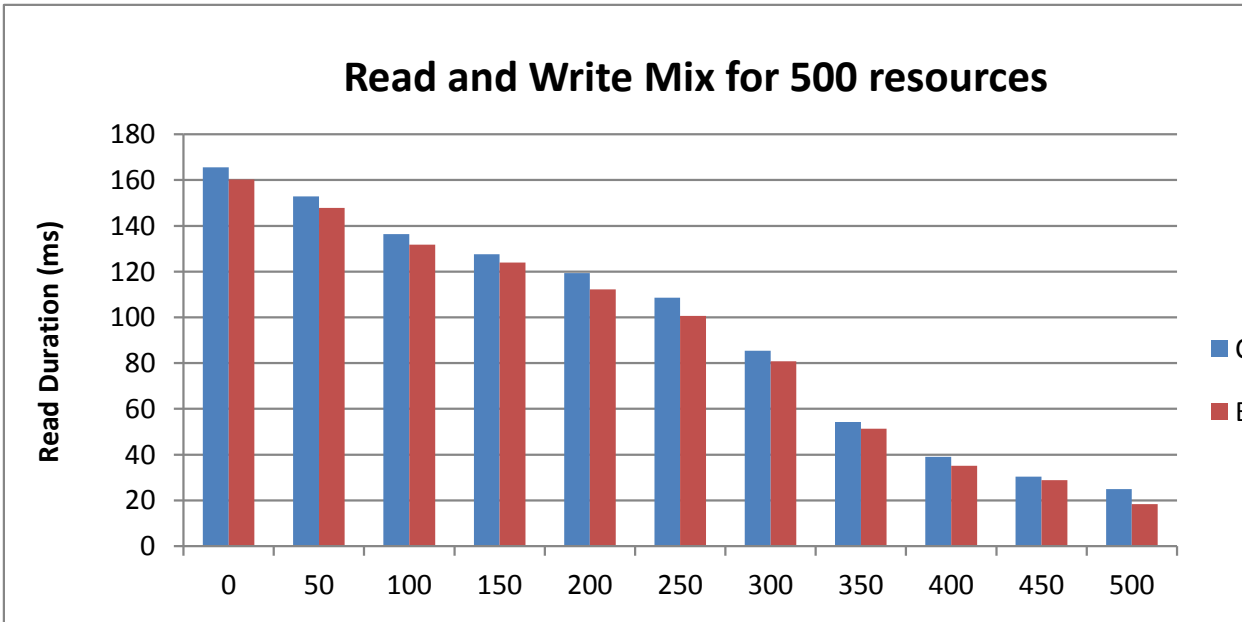


Figure 5.14: Read and Write mixed requests

From the graph, it can be inferred that it takes a longer duration for an update event to travel to the mobile client when the mobile device has to wait for more write requests. However, from the analysis, the average duration for 500 newly created resources to be visible to a connected client when the write request is routed through the CouchDB middleware is 0.095 seconds. On the other hand, it will take approximately 0.090 seconds if the write request is routed through the Erlang middleware. The resulted inconsistency window suggests that the time for an update to move from one client through any of the middleware versions is relatively fast though the Erlang middleware minimizes latency better.

However, a limitation of our read and write test is that; it did not take into account an HTTP PUT request. The PUT request is another form of a write operation that makes changes to an existing resource. Considering the fact that mobile client caching is proposed and implemented in our architecture, it will be interesting to know how the read and "update the existing resources" mix experiment can be employed to determine the inconsistency window.

The experiment will involve recording the duration it takes for the middleware to make changes to its resources. In addition, the updates will be sent to the mobile device which will also synchronize its cache with the new resources state. The entire duration for an update to be visible on all nodes will be the inconsistency window. Though the described scenario has not been evaluated in our system, it has been fully factored into the implementation.

Another caveat with our approach is the continuous polling of the middleware from the mobile device. This consumes network bandwidth but the bandwidth consumption has not been evaluated in this thesis since it is not a major challenge under consideration. The next set of experiments focuses on how the implemented architecture responds to inputs and requests from users. The observer employ fault injection approaches as described in the next section.

5.4 Fault Injection Approaches

The fault injection testing is adopted to evaluate the resilience of our system. The test is suitable for the cases involving crashes and disconnections between the mobile client and the back-end components. The mobile provider and the requester are intentionally disconnected periodically whiles requests are sent from other requesters; and it is expected that the requester receives some message. The idea is that distribution transparency will be minimized in situations of long lived disconnection; with a clear message sent to the user. Distributed transparency is a way of hiding detailed abstraction of network components from the user.

Also, the fault injection testing evaluates the efficiency of the mobile side caching in terms of supporting user mobility. As the mobile provider/requester is disconnected from the middleware, the last successful cached update on the client is expected to be pushed to the screen with a clear message of disconnection to the user. Timing will be used to provide a global world view of requests. It is also expected that updates are pushed to the client the moment

connectivity is restored using the long-polling technique. The various scenarios considered in the experiment are described in the cases in the next section.

5.4.1 Case 1: Disconnected Provider

The setup for the unavailable mobile service provider testing is shown in Figure 5.15. The experiment focuses on the responses of the middleware when the mobile provider is not available. At the beginning of the experiment, the Couchbase Mobile storage on the mobile provider is populated with 100 Web resources representing patients' records. The CouchDB middleware connects to the Couchbase Mobile storage on the mobile provider and synchronizes the data. After the synchronization, the mobile provider is disconnected and an HTTP GET request is sent from the mobile requester to the disconnected provider through the middleware.

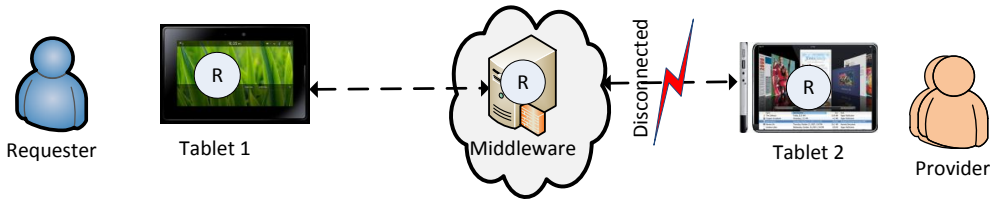


Figure 5.15 Disconnected mobile service provider

The unavailable provider's case is handled by the use of timestamps. The middleware sends a response to the mobile requester with the message of temporary unavailable message as shown in the screen shot below in Figure 5.16. Figure 5.17 shows the message from the middleware when the button labeled OK is clicked. Due to privacy in the E-health domain, the names and information showing on the device in Figure 5.16 and Figure 5.17 are not real patient and doctor's information.

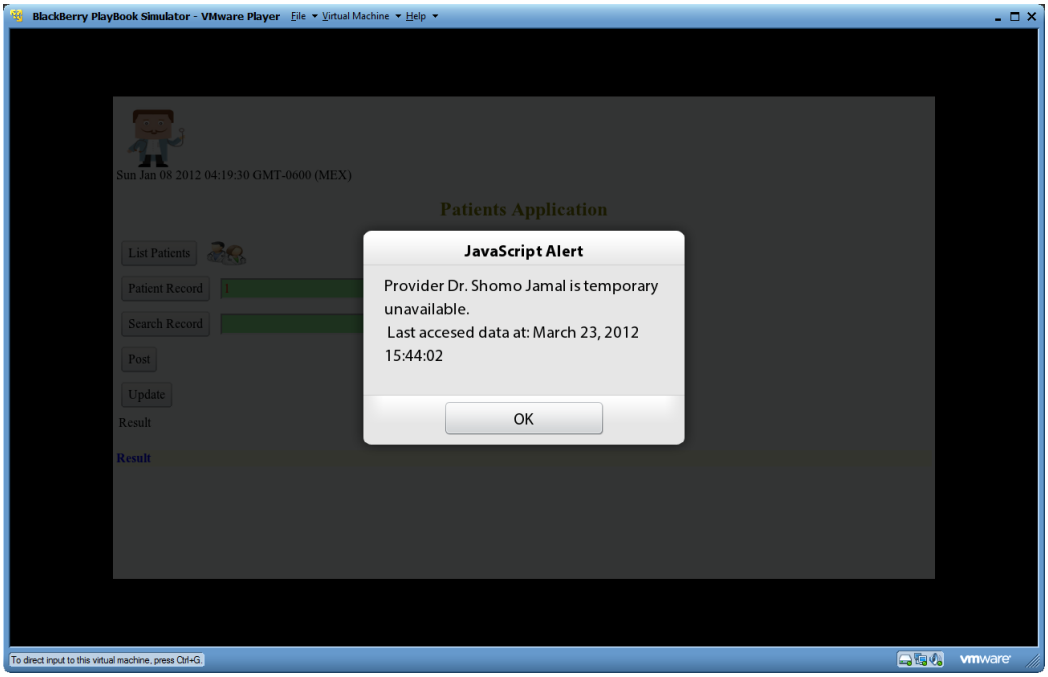


Figure 5.16: Temporary unavailable message

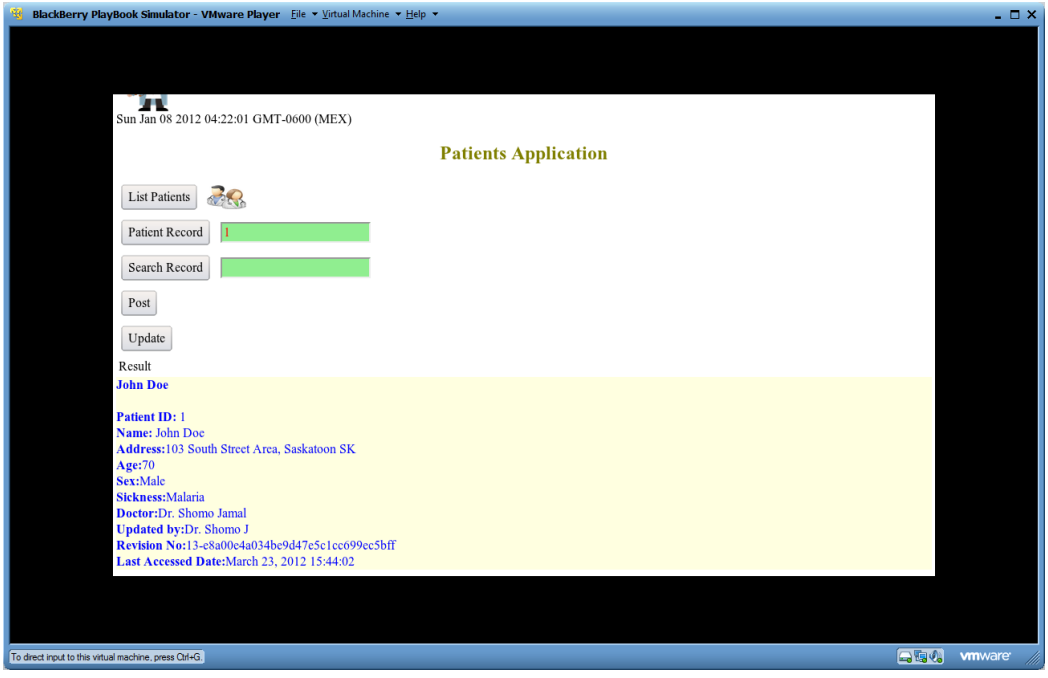


Figure 5.17: Message from the middleware

5.4.2 Case 2: Disconnected Requester

The experimental setup in Figure 5.18 illustrates the situation where a requester sends a request and could not get back the response because it lost connectivity.

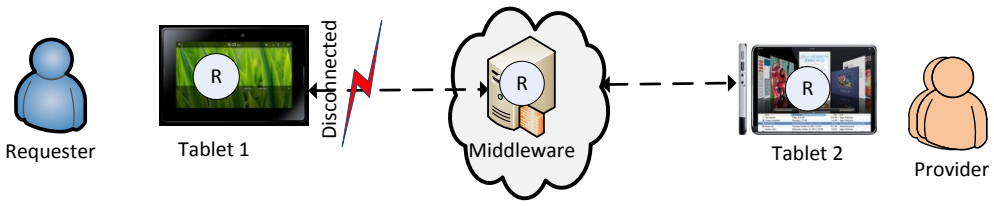


Figure 5.18: Disconnected requester

Currently the way this situation is addressed in our system is for the requester to re-connect to the middleware and make the request again. Since the provider has responded to the request, it will be stored on the middleware. Hence, even though the requester has to make the request again, it will take a shorter time to serve the client based on the experimental results on resource accessibility on the middleware.

5.4.3 Case 4: Disconnected Provider and Requester

Case 4 is depicted in Figure 5.19. This is a situation where by the requester sends a request to the provider and the provider is unavailable. At the same time that the middleware is getting back to the requester with the unavailability message of the provider, the requester is also disconnected. When such a situation occurs in our system, the transaction is cancelled and not registered.

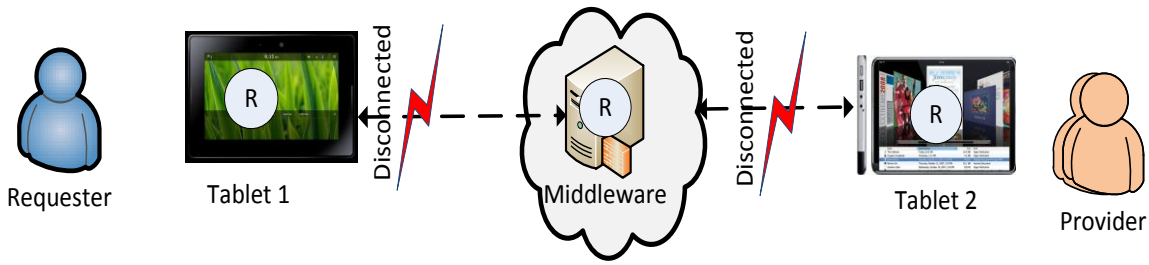


Figure 5.19: Both the requester and the provider are disconnected

5.5 Summary

This Chapter described the various experiments that were conducted to evaluate our system's solutions to the challenges of latency and state synchronization management. The first part of the evaluation focused on the measurement of overhead introduced by the designed system on minimizing latency in a Wi-Fi network. The overhead testing employed simulation in a laboratory to determine how long it takes for data to be accessed on participating hosts. The evaluation focused mainly on requester-provider and requester/provider-middleware interactions. In addition, since the performance of systems slow down under heavy workloads, resulting in high latency, scalability testing was conducted.

Furthermore, based on the adoption of the read-your-writes consistency model, the inconsistency window of the implemented system was determined by conducting read-write mixed experiment. Also, the fault injection approach was used to determine the responses of the middleware to certain events. The summary of the experiments are listed below based on the goals.

Goal 1: Testing resource accessibility on the resources' hosts

Experiment 1: Evaluated resource accessibility on the mobile provider over the CouchDB middleware. The result shows that for a maximum of 25 read requests in a

closed loop, the average response time for a requester to be served is approximately 0.5 seconds.

Experiment 2: Evaluated the accessibility of RESTful resources on the mobile service provider over the Erlang middleware. It takes approximately 0.56 seconds for a maximum of 25 sequential requests to be served in a closed loop.

Experiment 3: Evaluated the accessibility of RESTful resources on the CouchDB middleware storage. On average, a maximum of 70 read requests are served in approximately 0.40 seconds

Experiment 4: Evaluated the accessibility of RESTful resources on the Erlang DETS table by a mobile requester. The result shows that it takes approximately 0.37 seconds for a maximum of 70 sequential requests to be responded to.

Goal 2: Scalability testing of the middleware platforms

Experiment 5: The throughput of the CouchDB middleware was evaluated. Considering 500 users of the system who can send concurrent HTTP GET requests ranging from 500 to 40000, the CouchDB middleware processes an average of 124.23 requests per second.

Experiment 6: The evaluation shows that the Erlang environment is more scalable than the CouchDB since it processes an average of 320.42 requests per second under the same conditions as the CouchDB.

Goal 3: Determining the inconsistency window of our system

This goal is evaluated based on the time it takes for all connected clients to have access to newly created resources that are propagated from the client to the middleware. The test employed the read-write mixed technique for 500 RESTful resources. The results show that, if

the CouchDB middleware is used, the inconsistency window is 0.095 seconds whereas if the Erlang middleware is employed, the inconsistency window is 0.090 seconds.

Goal 4: Fault injection approaches to determine system responses

The evaluation takes into account failure events of participating mobile nodes. Assuming the provider is disconnected, the middleware pushes a cached resource to the requester with a temporary unavailable provider message. Also, in case the provider is reachable but the requester is disconnected before the response arrives, the requester is expected to re-connect. Furthermore, if the provider and the requester are both disconnected, the system cancels the entire transaction.

CHAPTER 6
SUMMARY AND CONTRIBUTION

This thesis has shown that mobile devices such as smartphones, tablets, and notebooks can be hosts and providers of RESTful Web services in Enterprise Information Systems. Using REST as an architectural design, Web services can be efficiently built and deployed on these devices because REST is a lightweight protocol. However, there are challenges such as network latency and state change propagation in mobile distributed systems due to transient connectivity. In addition, the CAP theorem phenomenon is applicable in the mobile distributed environment. Out of the three system requirements - data consistency, system availability, and partition failure; only two can be simultaneously attained in a distributed system. The focus of our work is to ensure high availability so we traded-off data consistency since the unstable connectivity in mobile distributed systems enforces partition tolerance. Thus, the challenge is how to ensure eventual consistency.

Our research addressed the challenges of latency and resources state management by proposing an architecture that comprises a mobile service requester, a middleware, and a mobile service provider. The mobile application's user interface is implemented using HTML5 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 is also influenced by the fact that we aimed to achieve deploying a single code base on multiple platforms. The application was successfully deployed on various platforms including BlackBerry Playbook, BlackBerry smartphone OS 5.0, and Android tablet. Also, in order to model the mobile device as a provider, Couchbase Mobile framework was configured and hosted on the Android tablet device which enabled an HTTP communication to be established between the mobile device and the middleware. As a result, requests can be sent to the mobile provider just as a server. By this approach, mobility of users is supported even if there

is loss of connectivity. The mobile provider's records or messages can be updated by the user and when the connection is restored, the mobile side data will be synchronized with the cloud-hosted middleware. Also, messages being sent to a disconnected mobile provider are queued on the middleware and upon establishing a connection, the middleware pushes the updates to the provider.

Furthermore, two of the leading NoSQL frameworks, Erlang and CouchDB, were explored to determine which one supports the building of the middleware better. The middleware employed caching techniques and event notifications such as client connections, client disconnections, and resources state change; to update all participating mobile service requesters and providers. The middleware also uses long-polling and pushing to receive and send updates to mobile clients which have subscribed for services. In addition, the middleware is hosted on the Amazon EC2 cloud and all the REST Web resources of the mobile providers are replicated on the middleware to ensure high availability. An eventual consistency model following the read-your writes consistency is employed to synchronize data between the middleware and the mobile participants.

The contribution and the findings of our work are summarized below.

- It is feasible to host providers and consumers of RESTful Web services on the mobile device in an enterprise domain.

- The use of HTML5 and its supporting Web tech frameworks enhance the deployment of a hybrid mobile app.

- It is a good idea to replicate the RESTful resources of the mobile providers on a cloud hosted middleware. This is to aid mobile requesters to access the resources of a disconnected mobile provider.

- Read-your-writes consistency approach can be adopted together with pushing and polling, in a mobile distributed network, to synchronize data in real time.

- Though Erlang and CouchDB have good support for mobile devices in terms of acting as middleware, the Erlang platform is more scalable and has higher performance for minimizing latency.

Though the goal of hosting providers and consumers of RESTful Web Services (REST-WS) on the mobile device has been achieved, there are some limitations with our work. In our architecture, a centralized middleware system was proposed without paying attention to the fault-tolerance of such a system. Since the middleware acts like a hub in our system, the moment it crashes, no mobile provider can be reached by a requester. Though the choice of Erlang programming language and CouchDB is to ensure system resilience, it is not enough to overcome hardware failures. One approach to solving this problem could have been the adoption of database distribution on multiple computers. However, the overhead of this approach has to be evaluated to determine its viability.

Also, enough attention is not given to data safety on the mobile provider; in case the mobile device gets into the wrong hands. Thus, in our future work, these factors will be considered.

CHAPTER 7
FUTURE WORKS

7.1 Mobile P2P Provisioning

The work presented in this thesis focuses on using a middleware in a centralized distributed system to support the communication of the mobile hosts of RESTful Web services. The middleware keeps track of updates that are being propagated from the mobile participants by listening to events such as resources state change. Additionally, the middleware notifies all the mobile participants of possible updates in real time.

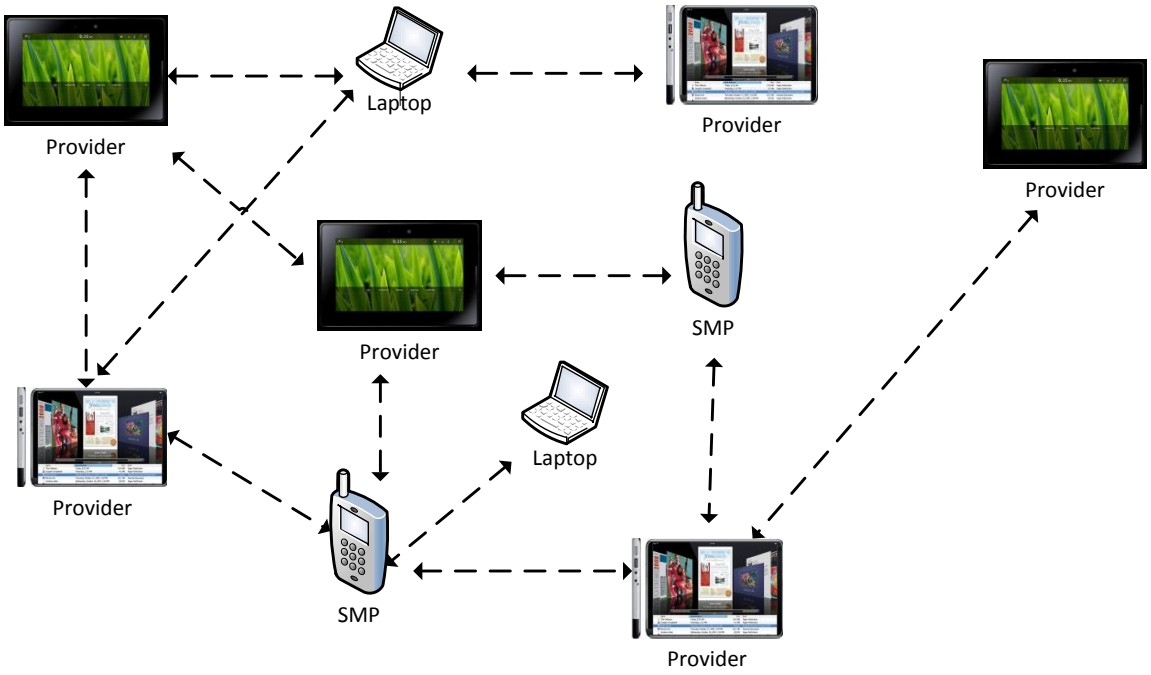


Figure 7.1: Network of mobile providers and consumers

However, in the future, we hope to investigate the possibility of sharing Web resources between mobile participants without using a centralized middleware that is hosted in a wired network. Rather, the mobile participants could be modeled to communicate directly in a peer-to-peer (P2P) environment. The P2P approach means that individual mobile providers' resources

will be stored by other providers as illustrated in Figure 7.1 in order to increase resources availability.

The challenge however in the P2P approach is how to ensure the reliability of the data that is being kept by a neighboring provider. The reliability problem arises because of the intermittent loss of connectivity between the mobile devices. As a result, mobile devices could potentially be hosting resources that are outdated and tracking the updates of users when the mobile distributed system is large can be challenging. Additionally, assuming a mobile provider, *HostA*, has its replica resources stored on a neighboring provider, *HostB*, there are chances that both *HostA* and *HostB* could be unreachable simultaneously. This situation could lead to total unavailability of services or Web resources.

A possible solution for mobile hosting in P2P environment could be the use of an audit trail and timestamps techniques. As the mobile providers are sharing their resources, the changes that are being done by users to resources could be logged. The users who are doing the changes and the times at which the changes are made could also be recorded. With the audit trail concept, reliability of the data could be determined.

7.2 Decision Tracking in Mobile Provisioning Systems

In our present studies, the focus is on the availability and the reliability of the data that is shared between the mobile participants. Thus, the architecture employs the *Level 1* and part of *Level 2* of the Richardson's Maturity Model. As a result, the mobile participants interact with multiple Web resources of the provider using the HTTP methods such as GET, POST, and PUT.

However, the issue of Web services modeling can become more complex than just sharing data especially in Enterprise Information Systems such as E-health. In our current architecture, when an update is pushed to a mobile participant, say updated patients'

demographic record, the decision about how to use the data is made by the human expert (i.e. the health care professional). This means that the mobile providers just deliver readings to the human experts. The situation becomes complex if decision making is introduced into the system where for instance, previous decisions are reversed based on inferred data. A potential data inference scenario in the E-health domain could be seen when a systems' decision is based on blood test result from the laboratory. Assuming a decision is made based on an earlier result and then a new result arrives from the lab but the intended provider is temporary disconnected; whenever the provider becomes available, it has to make a new decision based on the new result. The scenario is further explained in Figure 7.2 when timestamps are employed.

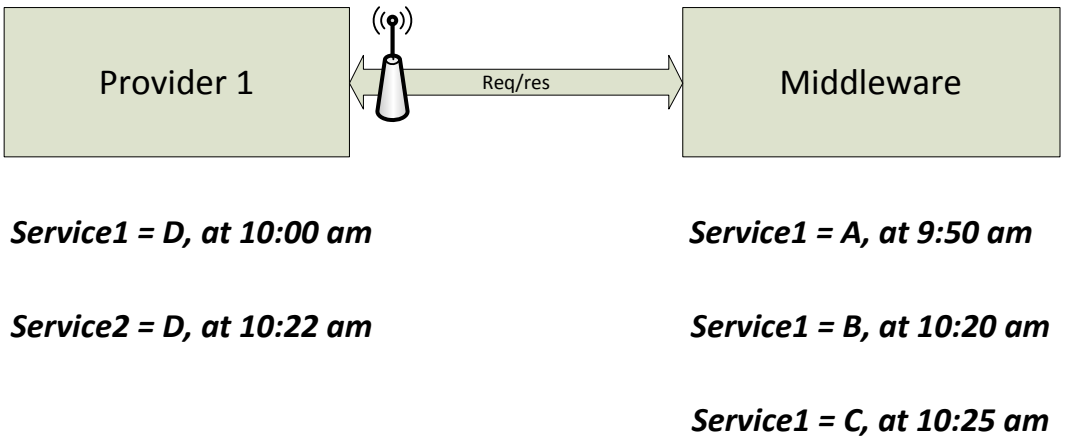


Figure 7.2: Decision making based on timestamps

As illustrated in Figure 7.2, the state of a service on the middleware, say *Service1* is A at 9:50 am. When the provider connects to the middleware 10 minutes later, the provider could make a decision D based on the available service. Assuming *Service1* is updated at 10:20 am to state B on the middleware, then based on the new state, the provider has to make a new decision. In non-monotonic systems, a new service state could potentially lead to totally different decision from earlier decisions. Furthermore, the case could be complicated if for instance, the provider later connects at 10:22 am to make a decision D based a new service say *Servcie2*. Based on

Service2, the provider could request for *Service1* to be updated to C on the middleware. The complexity further increases if multiple services are being created based on inferred data. Since connectivity is not guaranteed, different service states could arrive on the middleware without the provider knowing. In addition, based on a new state of a service, a ripple effect could be triggered where by previous decisions in the system has to be reversed. Also, there could be cascading effect where based on a new service state from the provider, other mobile participants have to update their services as well.

A possible solution to addressing the above system complexities could be the adoption of *autonomic computing* [84] technology. Proposed by IBM, the concept of autonomic computing is to build highly distributed systems that manage themselves. Thus, such systems are expected to make decisions and recover from system failures without human interventions. Regarding mobile provisioning systems, the autonomic computing technique could be employed to enable decision tracking. The system could be built with audit trail mechanisms and timestamps; in which case users' activities could be logged in a file. Rules could also be introduced based on the timings of the resources state changes. A simple rule could state that if resources state change is detected by the provider, all the services in the log file should be re-run.

Furthermore, the adoption of the autonomic computing technique in mobile Web hosting could follow the proposed framework of Rahman et al. [85] as illustrated in Figure 7.3.

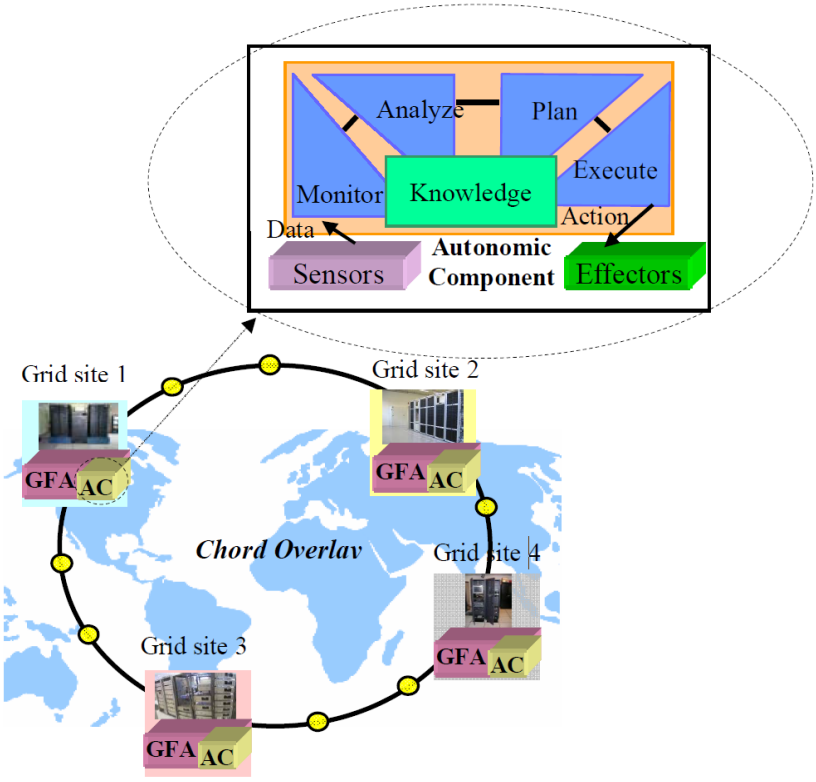


Figure 7.3: Decentralized autonomic workflow management [85]

Rahman et al. [85] in their paper "*An Autonomic Workflow Management System for Global Grids*", put forward the use of autonomic computing framework that manages workflows in dynamic network environments. To achieve this, the authors design a Grid Federation Agent (GFA) at every node (called Grid site) that logs the users' execution of the application's workflow. Furthermore, each GFA has its own autonomic component (AC) that self-manages the activities on the node. The AC performs actions such as workflow execution monitoring, data analyzing, planning against possible failures, and application execution. As a result, the paper reported the success of building autonomic system that is able to react according to changes in the dynamic network, self-heal from failures, and discover services.

Although Rahman et al. [85] implemented the framework in Grid environment, future studies on mobile provisioning could adopt the idea to overcome the challenges of decision

making. Since there are challenges of network loss in mobile networks, the provider could be modeled to have an autonomic component (AC) with the properties shown in Figure 7.3.

7.3 Data Security on the Mobile Host

Now that Web services can be hosted on the mobile devices, another concern that will be addressed in our future studies is the security and privacy of the data. Mobile devices are personal assets and when used in hosting enterprise applications can contain non-disclosure information. Hence, it is important that best security practices and techniques are explored to ensure data safety. In addition, the issue that our further studies will investigate is how to determine the access levels of users on the mobile hosts. In most enterprises, information and data accessibility is maintained in a hierarchical order. It is therefore a challenge to determine the access levels across multiple mobile hosts.

Atluri [86] presents a security outline for workflow systems in Enterprise Information Systems. According to Atluri [86], security concerns that must be addressed in Workflow Management Systems include the following.

- *Confidentiality*
- *Integrity*
- *Availability*
- *Authentication*
- *Authorization:*
- *Audit*
- *Anonymity*
- *Separation of duties*

Based on the security factors outlined above, further studies on mobile Web services provisioning could possibly adapt the Workflow Authorization Model (WAM) proposed by Atluri [86]. In the model, authorized users or agents (called subjects) are given certain access levels (called privileges) to perform operations (called objects). Also the WAM framework gives access to a subject only at run-time by employing Authorization Templates (AT) which is attached to each activity. By this approach, authorization is granted at the beginning of an activity execution and it is revoked when the activity is complete.

Future exploration of security issues in mobile hosting of enterprise information could employ the idea of Authorization Templates to enforce access level privileges. For example, in the E-health domain, health care professionals could have ATs attached to individual clinical activities though they may be using the same application interface. A potential clinical workflow process is graphically depicted in Figure 7.4 based on the WAM framework of Atluri [86].

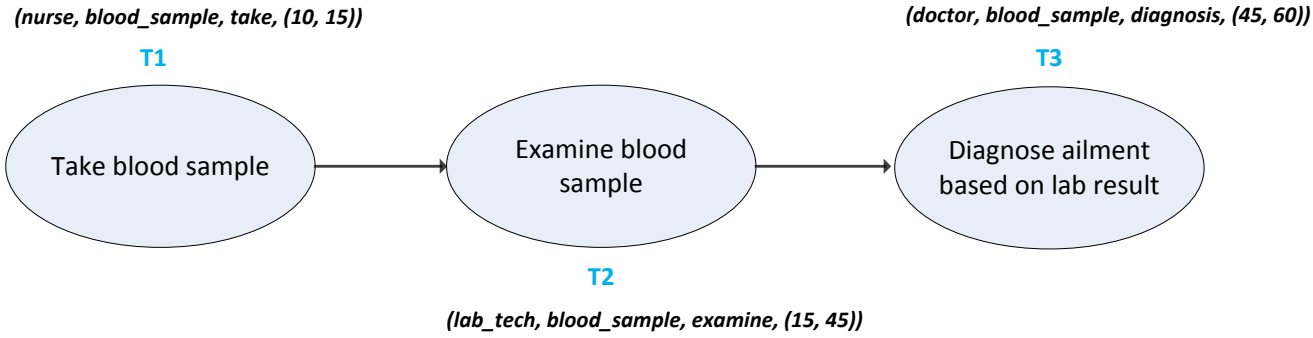


Figure 7.4: Clinical Workflow Authorization Model

From Figure 7.4, assuming an activity, *T1*, is defined as taking a patient's blood sample by a nurse, the system could give an access level (privilege) that allows for the information regarding the patient to be registered and possibly notify the laboratory technician of the activity. The nurse's activity could start from say time 10 and finish at time 15, after which the nurse's privilege should be revoked. A rule could be introduced into the system which states that the

next activities say *T2* and *T3* could be performed if and only if the first activity, which is *T1*, is successful. Based on this rule, after the successful completion of *T1*, the laboratory technician could be given the privilege to record the results, *T2*, after examining the blood sample. The activity of the laboratory technician could start from time 15 and end at time 45 after which the privilege could be revoked. The successful completion of *T2* could lead to the initialization of *T3*, which is the activity of the doctor, which involves recording the diagnosis of the ailment based on the lab result. The doctor could also have additional privileges of writing prescriptions for the patient as well.

Now, considering the authorization control flow described within the E-health domain, the case of unauthorized disclosure of information and unauthorized changes of data could be avoided. Additionally, the authorization flow leads to clear separation of duties since users are given privileges that are only related to their activities.

However, the WAM framework could become inefficient in mobile distributed environments as a result of unexpected losses in connectivity. There could be cases where the lab technician is unreachable after the completion of *T1* or the doctor could be temporality disconnected at the time *T2* is completed. The challenge therefore is how to initialize the next activity. To overcome such problems, the system could be designed to revisit previous states. For example, if the doctor is not sure that *T2* is completed successfully; the workflow could be reversed to activity *T2* for an acknowledgement (or confirmation).

7.4 Resilient Mobile Hosting

Apart from the security concerns, another research that is worth exploring in the future is building resilient frameworks that support enterprise data hosting on mobile devices. Fault-tolerance (also often called graceful degradation) is an attribute of a computer system that

enforces service delivery to users or sub-system components in case of system failures [87]. In the architecture proposed in this thesis, the Web resources replication technique is employed which is helpful for data recovery in the event that a mobile device becomes faulty. However, issues concerning errors in the decision making workflow are ignored since it is outside the scope of this thesis.

Errors in decision making workflows are bound in mobile environments due to unstable connectivity. As a result, the coordination of activity workflows cannot be guaranteed at all times. For example, in an enterprise system such as E-health, the following scenario could be used to define a workflow for the University of Saskatchewan Health Center.

1. *The medical staff at the front desk records or verifies the patient's demographic record including authenticating the patient's health insurance card and notifies a nurse.*
2. *The nurse meets with the patient and asks basic questions to understand what is wrong with the patient. After which the nurse notifies the doctor (or the expert) on duty. All the interactions between the patient and the nurse are recorded in the computer system as well.*
3. *The doctor meets the patient and interacts with the patient and may conduct physical examination to determine what is wrong with the patient. The doctor also has to log every interaction with the patient in the computer system.*
4. *The doctor could recommend the patient to go for lab test after which the lab technician has to document or record the lab result.*
4. *The lab result is then sent back to the doctor who has to do diagnosis based on the lab result and give prescriptions (this activity has to be recorded as well).*

As illustrated in the above scenario, the doctor can only give prescriptions or diagnose a patient's ailment based on the result from the lab. However, if the process flow is implemented in mobile networks, there could be cases where the lab result will not reach the doctor on time (or at all) due to factors such as unexpected shutdown (due to limited battery life), latency, and hardware failure. Thus, it is essential that the mobile hosting environment is resilient enough to provide a consistent workflow for the health care professionals even in cases of failures.

In order to enforce resilient job execution process flow, Dasgupta et al [87] present a grid environment workflow manager that supports graceful degradation. First, the paper presents the various forms of workflow patterns based on Web Services Business Process Execution Language (WS-BPEL), which are graphically illustrated in Figure 7.5. The workflow could be executed in sequence, in parallel, in a loop, and choice.

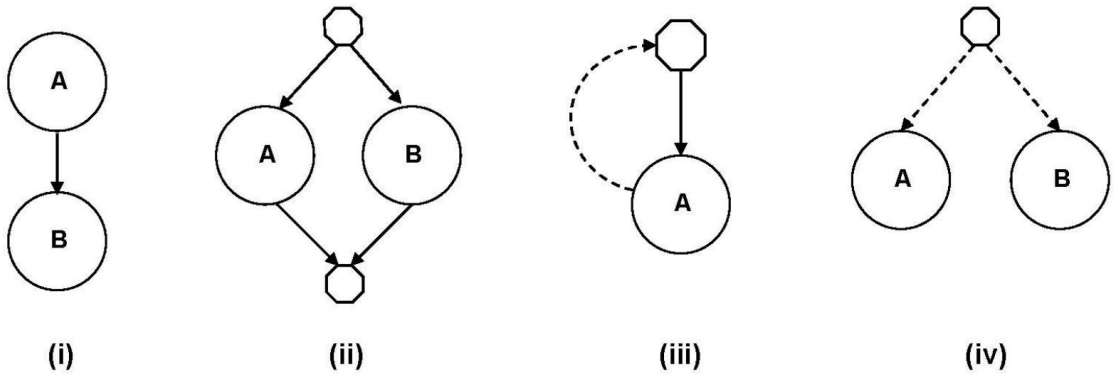


Figure 7.5: Basic workflow patterns described in [87]: (i) sequential, (ii) parallelism, (iii) loop, and (iv) choice.

Based on the basic workflow patterns, Dasgupta et al [87] proposed a framework that supports system error recovery regardless of the adopted workflow pattern. The job flow execution pattern is illustrated in Figure 7.6 as a state transition diagram.

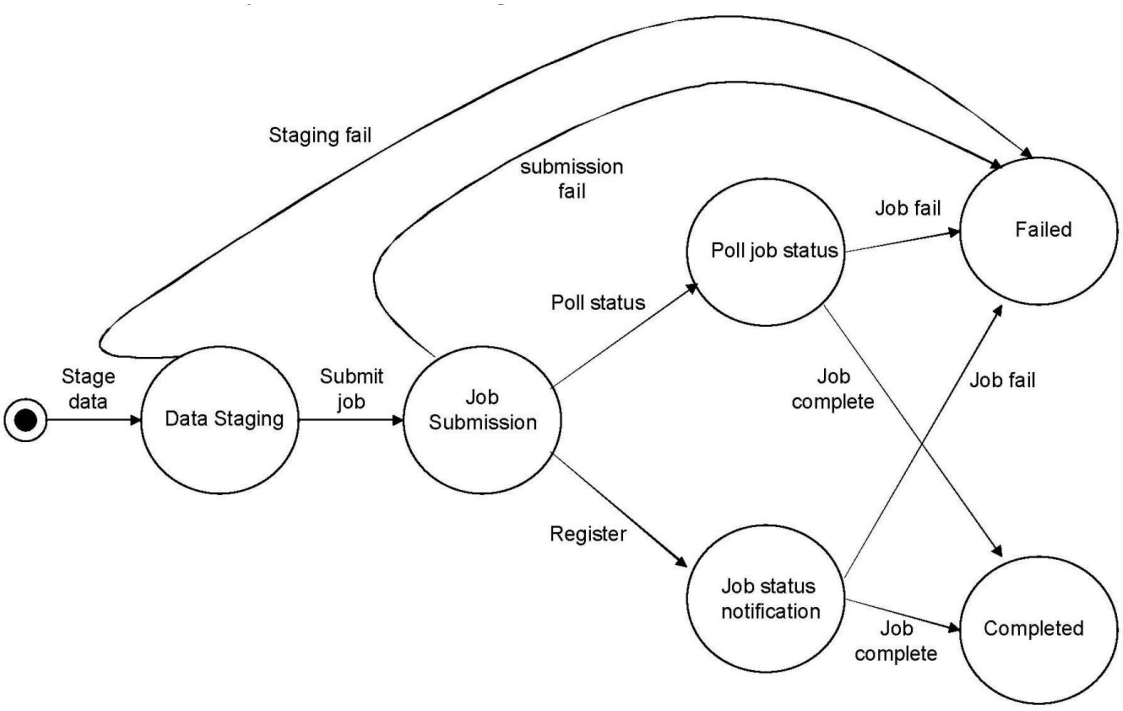


Figure 7.6: Normal job-flow patterns [87]

In the proposed framework of Dasgupta et al [87] as shown in Figure 7.6, every transaction starts with data staging - which is an input from users or sub system component. The staged data is then sent to submitted state for processing. A submitted job could either be polled to determine the status (e.g. is the job completed or not) or notifications could be used to determine the status of the job. All completed jobs are forwarded to the completed state which could become a staged data for the next process or user who needs that information as an input. Also, a failed state is defined where failure attempts at other states are logged. For example, if there is fault at the submission stage, the job will be re-submitted as shown in Figure 7.7. Whenever a job is re-submitted, the status of the job will be re-polled or re-registration has to be done for job status notifications. Forced-fail status means that the transaction is terminated.

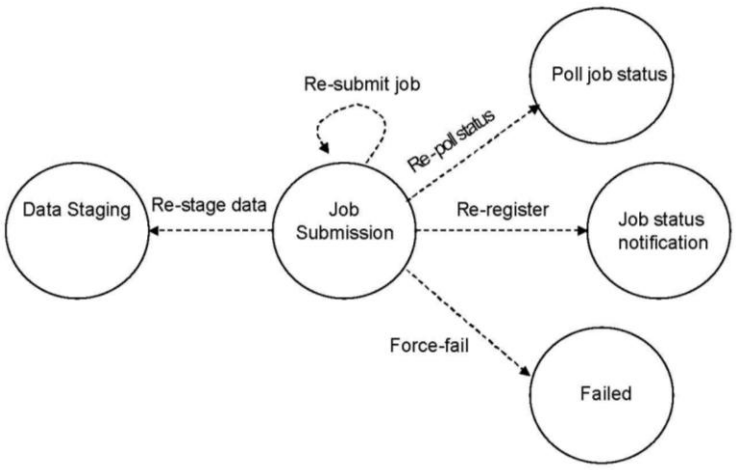


Figure 7.7: Re-submit job pattern [87]

The future studies on resilient mobile hosting of Web services could adopt the fault-tolerance framework of Dasgupta et al [87]. In that case, errors in transactional workflows could be registered for callbacks. Considering the E-health scenario for instance, if there is failure in the submission of the lab result from the lab technician due to system errors, the result could be re-submitted.

APPENDIX

Asynchronous Messaging: This is a message exchanging system where participants don't have to wait for a response from the recipient, because they can rely on the messaging infrastructure to ensure delivery. The alternative is synchronous messaging.

Composite Web services: An aggregation of multiple other elementary and composite Web services, which interact according to a given process model.

Distributed Systems: Multiple computers that interact among themselves via a computer network.

Distribution Transparency: Hiding the technical and network level details from the user and presenting the entire system as a single component rather than showing all functioning components.

E-Health: A study that use information and communication technology mostly with the aid of the Internet; to support the safe delivery of health care services.

Heterogeneous Web services: The description of the generic Web services that centered on SOAP, REST, WSDL and UDDI.

Middleware: Software consisting of a set of services that enables multiple processes running on one or more machines to communicate.

Mobile participants: All the mobile service requesters and mobile service providers that have subscribed to services on the middleware.

Proxy: Intermediary server that acts between the client and the server.

Web server: Hardware or software that holds and deliver Web content to clients via the Internet.

REFERENCES

[1] Gibbs, C. (2011). The Rise of Tablets in the Enterprise. GigaOM Pro, June 2011.

[2] Definition of Enterprise Information System. Norman's Java (TM) Glossary, Last accessed: January 15, 2012. http://dictionary.babylon.com/enterprise%20information%20system/

[3] RADVIEW, RadView Software Whitepaper, Load Testing Web 2.0 Technologies: Ajax-RIA-SOA-Web Services, Last accessed: January 15, 2012. http://www.radview.com/files/Load%20Testing%20Web%202.0%20Technologies%20%5BWhitepaper%5D.pdf

[4] Beal, V. (2010). Understanding Web Services. September 2010. Last accessed: January 15, 2012. http://www.webopedia.com/DidYouKnow/Computer_Science/2005/web_services.asp

[5] Ranck, J. (2010). The Rise of Mobile Health Apps. October 2010.

[6] Vital Wave Consulting, mHealth for Development: The Opportunity of Mobile Technology for Healthcare in the Developing World, The United Nations Foundation and Vodafone Foundation Technology Partnership, February 2009. Last accessed: January 10, 2012. http://www.vitalwaveconsulting.com/pdf/mHealth.pdf

[7] Burt, J. (2010). UPMC Creates E-Health App for BlackBerry Smartphones. March 2010. Last accessed: January 15, 2012. http://www.eweek.com/c/a/Health-Care-IT/UPMC-Creates-EHealth-App-for-BlackBerry-Smartphones-820733/

[8] Wicks, G., Van Aerschot, E., Badreddin, O., Kubein, K., Lo, K., & Steele, D. (2009). Powering SOA Solutions with IMS, Pg. 9 Publisher: IBM Redbooks Pub., Date: March 30, 2009, Part Number: SG24-7662-00, Pages in Print Edition: 410.

[9] Pautasso, C., Olaf, Z., & Leymann F. (2008). RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08, p 805-814, 2008, Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08.

[10] 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, 2011.

[11] Rodriguez, A. (2008). RESTful Web Services: The basics, Software Engineer, IBM. November 2008. Last Accessed: January 15, 2012. http://www.ibm.com/developerworks/webservices/library/ws-restful/

[12] Christensen, J. (2009). 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, p 627-633, 2009, OOPSLA 2009 Companion - 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 2009, Orlando, Florida, USA.

[13] Uribarren, A., Parra, J., Uribe J. P., Zamalloa, M., & Makibar, K. (2006). Middleware for distributed services and mobile applications, InterSense '06 Proceedings of the first international conference on Integrated internet ad hoc and sensor networks, May 30-May 31 2006, Nice, France.

[14] Beltran, V., & Paradells, J. (2008). Middleware-based solution to offer mobile presence services, MOBILWARE '08 Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications,
ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) ICST, Brussels, Belgium.

[15] van Halteren, A., & Pawar, P. (2006). Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning. Wireless and Mobile Computing, Networking and Communications, 2006 (WiMob'2006). IEEE International Conference, 19-21 June 2006, page(s): 292 – 299, Montreal, Que.

[16] Sheng, Q. Z., Benatallah, B., Maamar, Z., Dumas, M., & Ngu, A.H.H. (2004). Enabling personalized composition and adaptive provisioning of web services. In Proceedings of the International Conference on Advanced Information Systems Engineering 2004, LNCS 3084, pages 322-337, 2004.

[17] Al-Turkistany, M., Helal, A. S., & Schmalz, M. (2009). Adaptive wireless thin-client model for mobile computing. Wirel. Commun. Mob. Comput., vol. 9, 2009, pp. 47–59.

[18] Brewer, E. (2000). Towards Robust Distributed Systems, (invited Talk) Principles of Distributed Computing, Portland, Oregon, July 2000.

[19] Gilbert, S., & Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Service; SIGACT News, v 33, n 2, p 51-9, June 2002.

[20] Tharakan, R. (2010). Brewers CAP Theorem on distributed systems, Scalable Web Architecture, February 14, 2010, Last accessed: January 15, 2012.
http://www.royans.net/arch/brewers-cap-theorem-on-distributed-systems/

[21] Browne, J. (2009). Brewer's CAP Theorem. January 2009, Last accessed: January 15, 2012. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[22] Pritchett, D. (2008). BASE: An Acid Alternative, In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. Queue, v 6, n 3, p 48-55, May 1 2008.

[23] Dean, J. (2009). Designs, Lessons and Advice from Building Large Distributed Systems. keynote talk at LADIS 2009. Last accessed: January 15, 2012. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf

[24] Young, C. (2006). BizTalk Server 2006: The Compensation Model, Sagas. 2006. Last accessed: January 15, 2012. http://geekswithblogs.net/cyoung/articles/100424.aspx

[25] Garcia-Molina, H., & Salem, K. (1987). SAGAS, Department of Computer Science, Princeton University, Princeton, N J 08544, Last accessed: march 23, 2012. http://www.amundsen.com/downloads/sagas.pdf

[26] Vogels, V. (2008). Scalable Web services: Eventually Consistent, Vol. 6 No. 6 – October 2008 . http://queue.acm.org/issuedetail.cfm?issue=1466443

[27] CS Notes. (2009). Consistency Model - A Survey: Part I - What's Data Consistency Model and Why Should We Care?, 5/27/2009, Last accessed: June, 2011. http://xcybercloud.blogspot.com/2009/05/data-consistency-model-survey.html

[28] Menascé, D. A., Nandigama, S. K., & McAndrews, P. Causal consistency. Distributed Shared Memory, Last accessed: January 17, 2012. http://cs.gmu.edu/cne/modules/dsm/orange/causal_con.html

[29] Monash, C. (2010). DBMS2: Read-your-writes (RYW), aka immediate, consistency. A Monash Research Publication, May 1, 2010. Last accessed: January, 17, 2012. http://www.dbms2.com/2010/05/01/ryw-read-your-writes-consistency/

[30] Read-Your-Writes Consistency. (2011). Getting Started with Replicated Berkeley DB Applications, June 10, 2011, Last accessed: January, 17, 2012. http://download.oracle.com/docs/cd/E17076_02/html/gsg_db_rep/C/rywc.html

[31] Golding, R. A. (1992). A weak-consistency architecture for distributed information services. Computing Systems, Concurrent Systems Laboratory; Computer and Information Sciences; University of California, Santa Cruz, JOURNAL Vol. 5, July 6, 1992. http://citeseer.ist.psu.edu/viewdoc/versions;jsessionid=43748F56CBFD9F5843EB13BFCAA52BA3?doi=10.1.1.34.7444.

[32] Serrano-Alvarado, P., Roncancio, C., & Adiba, M. (2004). A Survey of Mobile Transactions. Distributed and Parallel Databases, Publisher: Springer Netherlands, Volume 16, Number 2, 193-230, DOI: 10.1023/B:DAPD.0000028552.69032.f9, 2004-09-24, http://www.springerlink.com/content/h4q47w6573560288/export-citation/.

[33] Web Services Architecture. (2004). Last accessed: March 03, 2011. http://www.w3.org/TR/ws-arch/.

[34] MSDN. (2011). XML Web Services Infrastructure. Last accessed: January 15, 2012. http://msdn.microsoft.com/en-us/library/sd5s0c6d(v=vs.71).aspx.

[35] Farley, P., & Capp, M. (2005). Mobile Web Services. Published in: BT Technology Journal archive, Volume 23 Issue 3, July 2005, Kluwer Academic Publishers Hingham, MA, USA.

[36] Exforsys Inc., SOA Disadvantages, Service Oriented Architecture Disadvantages & Applicability, Last accessed: January 15, 2012. http://www.exforsys.com/tutorials/soa/soa-disadvantages.html.

[37] Smith, R. (2008). Modeling in the Service Oriented Architecture. Intel Software Network, October 31, 2008, Last accessed: January 15, 2012. http://software.intel.com/en-us/articles/modeling-in-the-service-oriented-architecture/.

[38] Rusu, M., Saplacan, G., Sebestyen, G., Todor, N., Krucz, L., & Lelutiu, C. (2010). eHealth: Towards a Healthcare Service-Oriented Boundary-Less Infrastructure, Original Research:Applied Medical Informatics Vol. 27, No. 3/2010, pp: 1-14. http://ami.info.umfcluj.ro/Full-text/AMI_27_2010/AMI(27)_001_014.pdf

[39] Lee, W., Lee, C. M., Lee, J. W., & Sohn, J. (2009). ROA Based Web Service Provisioning Methodology for Telco and its Implementation, Management Enabling the Future Internet for Changing Business and New Computing Services, Proceedings 12th Asia-Pacific Network Operations and Management Symposium, APNOMS 2009, p 511-14, 2009.

[40] Overdick, H. (2007). The Resource-Oriented Architecture. 2007 IEEE Congress on Services, p 340-7, 2007.

[41] Cartwright, A. S. SOAP Soup. Last accessed: January 15, 2012. http://www.xmlfiles.com/articles/adam/soapsoup/default.asp.

[42] MSDN. (2011). Anatomy of an XML Web Service Lifetime. Last accessed: January 15, 2012. http://msdn.microsoft.com/en-us/library/x05s00wz(v=vs.71).aspx.

[43] Sessions, R. (2004). Fuzzy boundaries: Objects, components, and web services. ACM Queue, v 2, n 9, p 40-7, Dec. 2004-Jan. 2005.

[44] Feng, X., Shen, J., & Fan, Y. (2009). REST：An Alternative to RPC for Web Services Architecture, 2009 First International Conference on Future Information Networks, ICFIN 2009, p 7-10, 2009. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05339611.

[45] Tilkov, S. (2007). InfoQ: A Brief Introduction to REST, Community Architecture, Dec 10 2007. http://www.infoq.com/articles/rest-introduction.

[46] Hadley, M., Pericas-Geertsen, S., & Sandoz P. (2010). Exploring Hypermedia Support in Jersey, ACM International Conference Proceeding Series, p 10-14, 2010, Proceedings of the 1st International Workshop on RESTful Design, WS-REST 2010. April 26 2010, Raleigh, NC, USA.

[47] Burke, B. (2009). RESTful Java with JAX-RS, Publisher: O'Reilly Media, Inc. Pub. Date: November 17, 2009, Most Recent Edition Print ISBN-13: 978-0-596-15804-0, Pages in Print Edition: 320.

[48] Hadley, M. (2008). Web Application Description Language. W3C Member Submission 31 August 2009. http://www.w3.org/Submission/wadl/.

[49] Fowler, M. (2010). Richardson Maturity Model: steps toward the glory of REST, March 2010, Last accessed: January 15, 2012. http://martinfowler.com/articles/richardsonMaturityModel.html.

[50] Infosys Microsoft Alliance and Solutions blog, How i explained REST to a SOAP pro. Last accessed: January 15, 2012. http://www.infosysblogs.com/microsoft/2009/08/how_i_explained_rest_to_a_soap.html

[51] Selonen, P., Belimpasakis, P., & You, Y. (2010). Experiences in Building a RESTful Mixed Reality Web Service Platform, Web Engineering, Proceedings 10th International Conference, ICWE 2010, p 400-14, 2010.

[52] Stirbu, V. (2010). A RESTful Architecture for Adaptive and Multi-device Application Sharing, ACM International Conference Proceeding Series, p 62-65, April 26, 2010, Proceedings of the 1st International Workshop on RESTful Design, WS-REST 2010; Raleigh, NC, USA.

[53] Khare, R., & Taylor, R. (2004). Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems, In ICSE '04: Proceedings of the 26th International Conference on Software Engineering, pages 428–437, Washington, DC, USA, 2004, IEEE Computer Society.

[54] Sletten, B. (2009). The REST architectural style in the Semantic Web, JavaWorld.com, April 2009, Last accessed: January 15, 2012. http://www.javaworld.com/javaworld/jw-04-2009/jw-04-rest-series-4.html.

[55] Han, H., Kim, S., Jung, H., Yeom, H.Y., Yoon, C., Park, J., & Lee, Y. (2009). A RESTful approach to the management of cloud infrastructure, 2009 IEEE International Conference on Cloud Computing (CLOUD), p 139-42, 2009 CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing.

[56] Erenkrantz, J. R., Gorlick, M., Suryanarayana, G., & Taylor, R. N. (2007). From Representations to Computations: The Evolution of Web Architectures, Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 255-264, Dubrovnik, Croatia, September 2007.

[57] Erenkrantz, J. R., Gorlick, M., Suryanarayana, G., & Taylor, R. N. (2009). CREST: A new model for Decentralized, Internet-Scale Applications. Institute for Software Research, University of California, Irvine, Technical Report UCI-ISR-09-4, September 2009.

[58] Xu, X., Zhu, L., Liu, Y., & Staples, M. (2008). Resource-Oriented Architecture for Business Processes, 2008 15th Asia-Pacific Software Engineering Conference, p 395-402, 2008.

[59] Cloud Computing Sample Architecture. Last accessed: January 15, 2012. http://en.wikipedia.org/wiki/File:CloudComputingSampleArchitecture.svg.

[60] EBU – TECH 3300. (2005). The Middleware Report, System integration in broadcast environments, Geneva February 2005. http://tech.ebu.ch/docs/tech/tech3300.pdf.

[61] Alarcón, R., Wilde, E., & Bellido, J. (2010). Hypermedia-driven RESTful Service Composition, 6th Workshop on Engineering Service-Oriented Applications (WESOA 2010), San Francisco, California, December 2010.

[62] Sumari, P., & Rahiman, A.R. (2010). Caching Scheme for Handheld Device in Mobile Video-on-Demand System. Computer Graphics, Imaging and Visualization (CGIV), 2010 Seventh International Conference, 7-10 Aug. 2010, page(s): 49 – 54, Sydney, NSW.

[63] 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, pg:79 -84.

[64] Pajunen, L., & Chande, S. (2007). Developing workflow engine for mobile devices. Source: 2007 11th IEEE International Enterprise Distributed Object Computing Conference, p 279-86, 2007, Annapolis, MD.

[65] Rosenberg, F., Curbera, F., Duftler, M. J., & Khalaf, R. (2008). Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. Published in: Journal IEEE Internet Computing archive, Volume 12 Issue 5, September 2008, IEEE Educational Activities Department Piscataway, NJ, USA.

[66] da Rocha, R.C. A., Endler, M., & de Siqueira, T. S. (2008). Middleware for ubiquitous context-awareness. MPAC '08 Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing, December 1-5, 2008 Leuven, Belgium.

[67] Li, L., & Chou, W. (2010). Design Patterns for RESTful Communication Web Services. Conference: International Conference on Web Services - ICWS , pp. 512-519, 2010, DOI: 10.1109/ICWS.2010.101.

[68] Adamczyk, P., Smith, P. H., Johnson, R. E., & Hafiz, M. (2011). REST and Web Services: In Theory and In Practice, Book Chapter in REST: From Research to Practice, Springer 2011.

[69] Parastatidis, S., Webber, J., Silveira, G., & Robinson, I. S. (2010). The role of hypermedia in distributed system development. WS-REST '10 Proceedings of the First International Workshop on RESTful Design ACM New York, NY, USA ©2010.

[70] Kelly, M., & Hausenblas, M. (2010). Using HTTP link: header for gateway cache invalidation, WS-REST '10 Proceedings of the First International Workshop on RESTful Design, ACM New York, NY, USA ©2010.

[71] Engelke, C., & Fitzgerald, C. (2010). Replacing legacy web services with RESTful services, WS-REST '10 Proceedings of the First International Workshop on RESTful Design, ACM New York, NY, USA ©2010.

[72] Fernandez, F., & Navón, J. (2010). Towards a practical model to facilitate reasoning about REST extensions and reuse, WS-REST '10 Proceedings of the First International Workshop on RESTful Design ACM New York, NY, USA ©2010.

[73] Hernández, A. G., & García, M. N. (2010). A formal definition of RESTful semantic web services, WS-REST '10 Proceedings of the First International Workshop on RESTful Design, ACM New York, NY, USA ©2010.

[74] Jacobi, I., & Radul, A. (2010). A RESTful messaging system for asynchronous distributed processing, WS-REST '10 Proceedings of the First International Workshop on RESTful Design, ACM New York, NY, USA ©2010.

[75] QCon. (2011). http://qconsf.com/sf2011/tracks/show_track.jsp?trackOID=522.

[76] Pearce, J. (2011). HTML5 and the Dawn of Rich Mobile Web Applications. InfoQ Sections: Development, Architecture & Design, Jun 24, 2011 http://www.infoq.com/presentations/HTML5-Dawn-of-Rich-Mobile-Web-Applications.

[77] JQUERYMOBILE. http://jquerymobile.com/.

[78] JQTOUCH. http://jqtouch.com/.

[79] Couchbase Mobile. http://www.couchbase.com/products-and-services/mobile-couchbase.

[80] Erlang Programming Language. http://www.erlang.org/.

[81] Apache CouchDB Project. http://couchdb.apache.org/.

[82] ab - Apache HTTP server benchmarking tool. Last accessed: January 15, 2012. http://httpd.apache.org/docs/2.0/programs/ab.html.

[83] Nodeload. Last accessed: January 15, 2012. https://github.com/benschmaus/nodeload.

[84] IBM Research. Last accessed: February 11, 2012. http://www.research.ibm.com/autonomic/index.html

[85] Rahman, M., & Buyya, R. (2008). An Autonomic Workflow Management System for Global Grids. Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium, Lyon, page(s): 578 – 583, Issue Date: 19-22 May 2008.

[86] Atluri, V. Security for Workflow Systems. Rutgers University, Last accessed: February 12, 2012. http://cimic.rutgers.edu/~atluri/workflow.pdf

[87] Dasgupta, G., Ezenwoye, O., Fong, L., Kalayci, S., Sad-jadi, S. M., & Viswanathan, B. (2008). Design of a fault-tolerant job-flow manager for grid environments using standard technologies, job-flow patterns, and a transparent proxy, In Intl. Conf. on Software Engineering and Knowledge Engineering, 2008.