ENTERPRISE RESOURCE BUS

AND

VIEWS IN 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

SUNNY PAL SHARMA

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

Over the past few years RESTful Web Services (WSs) have gained immense popularity over the Web Service stack (WS*) to provide WS solutions. The WSs implemented using REST are lightweight and ideally suited for consumption for devices with limited processing capabilities like mobiles and tablets due to its dependence on HTTP. This enables the system architects to leverage the well-known capabilities of HTTP to develop their systems rapidly. However, this introduces the challenges of compromised security and unmanageable systems.

This research presents a novel middleware architecture called Enterprise Resource Bus for Resource Oriented systems to tackle the issues of security, access control and resource management. The middleware architecture put forward in this research is focused on remedying these issues by abstracting the logic of access control and security to the concept of view resources to be managed in the middleware. This research draws inspiration from the middleware architecture implemented in SOA called Enterprise Service Bus. Since, the implementation of ERB is inspired from ESB we have emulated some functionalities of ESB and translated them to resource oriented architecture.

In addition, this research also introduces the idea of views on resources inspired from the concept of virtual relations in relational databases to provide customized view resources based on user privilege/ role in the system to control access. The middleware architecture was tested for its overhead, scalability and security features as opposed to a REST Web Service without a middleware. It can be concluded from the evaluation results that with a small overhead a secure and highly manageable REST Web Services are achievable.

ACKNOWLEDGEMENT

First of all I would like to express my sincere gratitude to my supervisor, Dr. Ralph Deters for his advice, patience and assistance throughout my entire duration of study. Also, my sincere thanks to the members of my advisory committee: Dr. Julita Vassileva, Dr. Chanchal Roy and Dr. Chary Rangacharyulu for their valuable advice and suggestions. Furthermore, I would like to thank Dr. Anthony Kusalik and Dr. Mark Eramian for their guidance and support as a grad chair. Also, I would like to appreciate the assistance of Ms. Jan Thompson and Ms. Gwen Lancaster, at the Department of Computer Science, University of Saskatchewan, who have been helpful and assistive throughout my studies.

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

Finally, I would like to thank my father, Narinder Pal Sharma; my mother, Kamla Devi Sharma; and my sisters, Natasha Kamal Sharma and Monika Sharma for the unconditional love and continued support throughout my studies.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation

API ..... Application Programming Interface

CAC ..... Client Access Controller

CACD ..... Client Access Control Database

CAP ..... Consistency Availability Partition tolerance

CM ..... Cache Manager

CPU ..... Central Processing Unit

CRUD ..... Create Read Update Delete

DETS ..... Disk Erlang Term Storage

EAI ..... Enterprise Application Integration

EC2 ..... Amazon Elastic Cloud Computing

ERB ..... Enterprise Resource Bus

ESB ..... Enterprise Service Bus

ETS ..... Erlang Term Storage

HTML ..... Hyper Text Markup Language

HTTP ..... Hyper Text Transport Protocol

ID ..... Identifier

IT ..... Information Technology

JSON ..... Java Script Object Notation

MIME ..... Multipurpose Internet Mail Extensions

MOM ..... Message Oriented Middleware

PM ..... Process Manager

REST ..... Representational State Transfer

RM ..... Resource Manager

RO ..... Resource Orientation

ROA ..... Resource Oriented Architecture

RPC ..... Remote Procedure Call

RR ..... Request Router

SO ..... Service Orientation

SOA ..... Service Oriented Architecture

SOAP ..... Simple Object Access Protocol

SP ..... Service Provider

TXT ..... Text

UDDI ..... Universal Description Discovery and Integration

URI ..... Uniform Resource Identifier

WS ..... Web Service

WS* ..... Web Service Stack

WSDL ..... Web Service Description Language

XML ..... Extensible Markup Language

YAWS ..... Yet Another Web Server

CHAPTER 1
INTRODUCTION

Web Services (WSs) have gained immense popularity [18] over the past few years as a means of providing applications over the internet. There are two popular approaches to implement WSs nowadays: REST and SOA/ WS-*. In parallel to this development mobile client markets have grown exponentially and more features are being added to these devices every day [29, 34]. It can be said that we are in the midst of a computing paradigm shift from large bulky desktop machines to smaller and smarter mobile devices. Previous research by Hamad et al. [18] and AlShahwan et al. [2] on RESTful WS have shown that REST is ideal for delivering applications on mobile devices which are resource constrained and have poor network connectivity. However, this unprecedented growth could lead us to systems which are unmanageable, sluggish and have critical security issues. To solve this problem, this research draws inspiration from Service Oriented Architecture (SOA) which introduced a new architectural element called Enterprise Service Bus (ESB) to keep the scalability and security of SOAbased systems intact. Therefore, this research proposes to introduce a similar architecture for Resource Oriented Architecture (ROA) called Enterprise Resource Bus (ERB).

To put things in perspective one can think of an Enterprise Information System implemented as a RESTful WS. The main purpose of the WS is to provide access to the data stored in the system. However, in an Enterprise there are different levels of hierarchy of employees. Therefore, it's desirable in such a system to restrict access to certain information from certain employees. This can be a serious challenge for large enterprises which may need complete data-restructuring to accommodate these features. One possible way of handling this challenge is to abstract the access-control logic to a middleware which sits in-between the WS and the client.

Therefore, this research introduces an architecture which could emulate such behavior. The main highlights of this research are:

1. Introduction to ERB, a novel middleware architecture for ROA based systems, and
2. The application of database concept of Views to resources in REST to control access and provide more secure WSs.

The rest of the thesis is organized as follows: Chapter 2 discusses the problem definition. Chapter 3 reviews the literature associated to Web Services, middleware, and Views in Databases. Chapter 4 describes the system architecture and discusses various approaches to define views. Details about the implementation of the architecture are presented in Chapter 5. Chapter 6 describes the experiments and evaluation results. Chapter 7 summarizes the research contributions and finally Chapter 8 concludes this dissertation with possible future research.

CHAPTER 2
PROBLEM DEFINITION

This research focuses on dealing with consuming RESTful Web Service (WS) in a secure, scalable and simplified manner. RESTful WS bears resemblance to databases due to their dependence on state [31, 39]. Moreover, due to the growing pervasiveness of RESTful WSs new techniques for managing security and scalability of WS are being introduced. This research draws inspiration from the concept of views in databases and introduces the view concept on resources in Web Services. Therefore, the key question that needs to be answered is: how do we define and manage views on resources in a RESTful system?

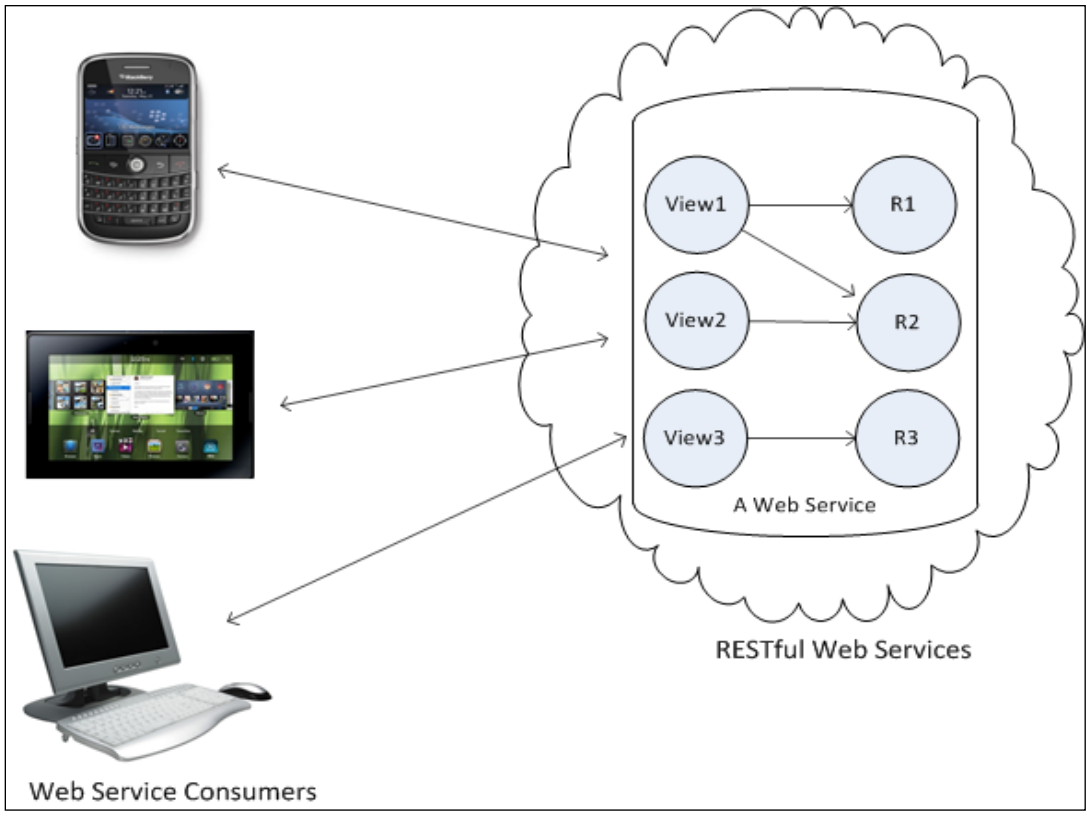

Figure 2-1 A WS exposing Views on Resources

In order to answer this question, the following four challenges must be addressed:

1. **View resources**: The resources are the building blocks of a Resource Oriented system. These resources are exposed to the client through a URI who can invoke simple operations like Create, Read, Update, or Delete (CRUD) on a resource using Hyper Text Transport Protocol (HTTP) methods. We call such resources as atomic resources. However, sometimes based on service requirements certain atomic resources should not be accessible OR be partially accessible to certain clients. For instance, in a hospital information system it may not be desirable to expose all patient data resource to the nurses but at the same time doctor may need to have access to all the patient information. Therefore, here the challenge is how to generate and serve view resources (defined on atomic resources) based on client access privileges/ roles.
2. **Access Control:** In a RESTful system anyone with the URI of a given resource can access that resource; this can result in unanticipated changes to the underlying resources. Therefore, the challenge here is to limit user access on resources based on their privileges.
3. **Load Balancing:** The response time is critical for the success of a Web Service. Moreover, the WS should automatically adapt to the changing needs of clients. E.g. the number of concurrent clients requesting access to a given resources at different time of the day may be different on different days. Therefore, the challenge here is to provide a timely response to the service consumer with minimum consumption of server side resources.
4. **Resources Management**: The proposed idea of implementing views on resources introduces new resources which are similar to the set of resource they are defined on. The main management issue faced after defining view resources is to keep track of

changes to the linked set of resources to reflect the latest state of the resources. Therefore, this raises the challenge of how these resources be stored and kept up-to-date.

The goal of this research is to develop a middleware architecture (Enterprise Resource Bus) which will be responsible for managing and defining views, and to improve security by exposing resources selectively to consumers. It will also help to improve scalability by spawning concurrent activities for each requested resource whenever a new request arrives without overloading the underlying WS. Lastly, it will help simplify the consumption of Web Service as the view resources can be user-specific and changed according to personal preferences.

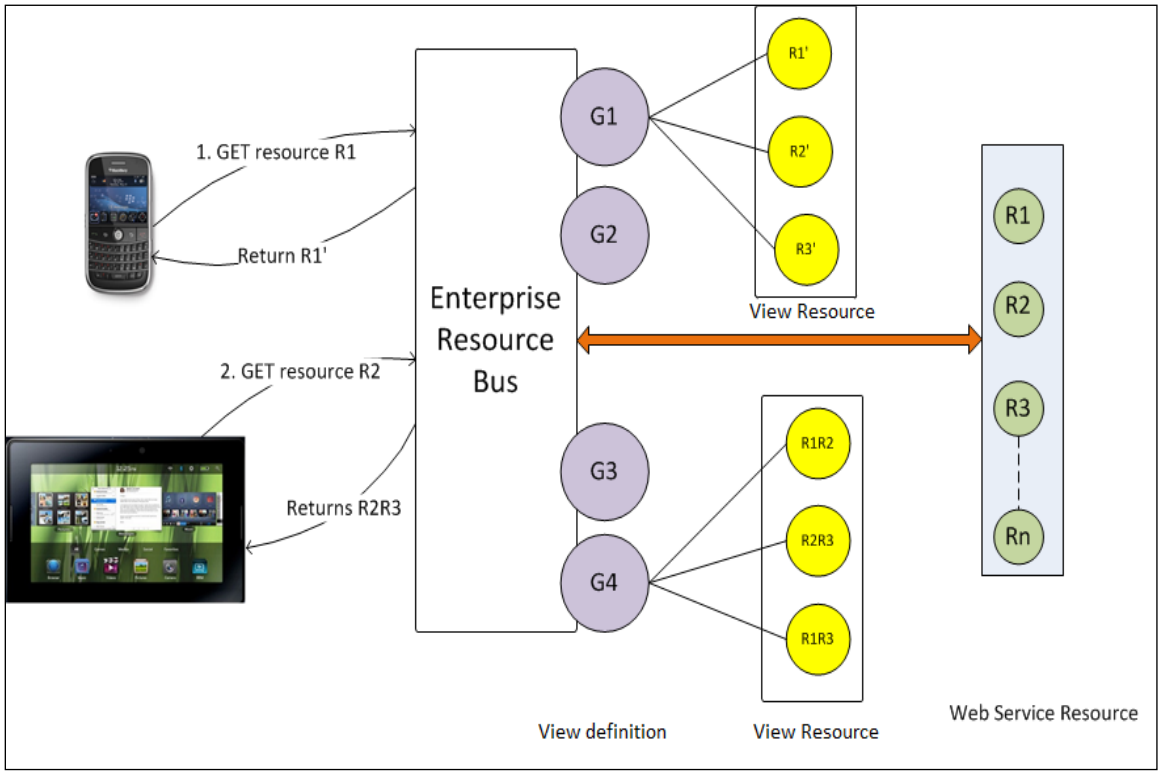

Figure 2-2 Introduction of Enterprise Resource Bus in a RESTful environment

To achieve these goals, the following key issues are to be considered:

1. How to define the views on resources?
2. How to manage client access to different resources or views in a secure manner?
3. How to manage consistency of the view data when updates are applied on the base resources?

CHAPTER 3
LITERATURE REVIEW

The literature review is organized as follows: To understand the semantics of Web Services (WSs), the literature related to Service Oriented Architecture (SOA) and Representational State Transfer (REST) was reviewed in section 3.1, 3.2, 3.3, 3.4 and 3.5. These sections also identifies why REST is a potential choice for implementing the WSs for the future when most of the consumers will be mobile devices. To increase the scalability and performance of the WS the literature related to Cloud computing was reviewed in section 3.6. This section also outlines how a cloud hosted service can enhance the performance of resource-constrained mobile devices. Section 3.7 reviews the literature related to Enterprise Service Bus to understand its various functionalities and how they can be applied to a middleware (Enterprise Resource Bus) for REST. The issue of having a consistent state of the data on the client nodes led to the review of CAP theorem in section 3.8. Furthermore, defining views on resource in REST has never been attempted before, so section 3.9 (views in relational databases) and 3.10 (MapReduce) reviews past successful approaches of defining views in other systems.

3.1 Web Services

A Web Service (WS) is a software system that supports machine-to-machine interaction over a network [41, 47]. A WS primarily comprise of an interface (e.g. WSDL) which describes the methods and services accessible to clients. A client can then initiate a connection by sending XML messages to the service according to the specifications provided in the Interface document over a network protocol like HTTP [18, 41, 47]. WSs have achieved immense popularity over the years due to its technology independent features [18]. Interoperability in distributed systems

has been made possible due to the open XML standards used by WSs [18]. An XML document being text based can be consumed across multiple computing platforms ensuring the availability of the service. There are other Web standards like WSDL and UDDI which ensures access control and availability of the service at all times [18, 43, 47].

According to Radview [47] there are three common styles in which a WS can be implemented: *Remote Procedure Call (RPC), Service Oriented Architecture (SOA) and Representational State Transfer (REST)*. RPC is less popular as compared to SOA and REST, therefore this research focuses only on the latter two in this thesis. There has been long debate from the proponents of both REST and SOA as to which is the best implementation style for a WS [5, 17]. Pautasso et al. [5] draws out comparison between REST and WS-* (SOA compliant WSs) on a multidimensional scale ranging from conceptual to technological aspects of both the styles. The paper [5] argues that both the styles are very similar on a conceptual level but a choice between the styles should be based on the requirements of the service one want to implement. For example, SOA may be a good choice for enterprise level applications integration where *transactions, reliability, and message level security* is of paramount importance and REST can be used for building *tactical ad hoc integration over the web (Mashups)* [5].

The tremendous growth of the WSs can be attributed to ubiquitous support for HTTP on various platforms [2]. Many technology vendors like Google, Amazon, eBay and Microsoft provide WS support by providing APIs to their service endpoints [47].

3.2 Service Orientation (SO)

Due to the growing integration problems in distributed systems, the concept of "Service Orientation" was introduced by Gartner [32] in 1996. SO focuses on the concept of decoupling IT resource from underlying system implementation [17]. According to Liu et al. "SO is a design

and integration paradigm that is based on the notion of well defined, loosely coupled services". SO helps to describe, publish, discover and orchestrate service in a platform independent manner [51]. To achieve SO, Evdemon [21] in his article "*Four tenets of Service Orientation"* outlines the best practices [21]:

1. *Services have clear boundaries:* The implementation of the service is hidden from the consumers by the use of service endpoints which act as an entry point for consumers
2. *Services are autonomous:* Changes to the service implementation should have no side effect on service boundaries thereby shielding consumers from the changes within the service.
3. *Services shares schema and contracts:* The communication allowed between consumers and service providers is pre-defined in a document following which service provider understand consumers request and provide appropriate response.
4. *Service compatibility is based on policy:* Semantic compatibility in services can be achieved by defining policies.

3.3 Service Oriented Architecture (SOA)

SOA [38] was first introduced by Gartner [32] in 1996 and is based on the principles of SO described in the previous section. SOA is a conceptual WS architecture where the focus is mainly on services and action [30, 42]. In SOA the service providers can publish the service in a registry, which in turn can then be discovered and consumed by consumers [32, 51].

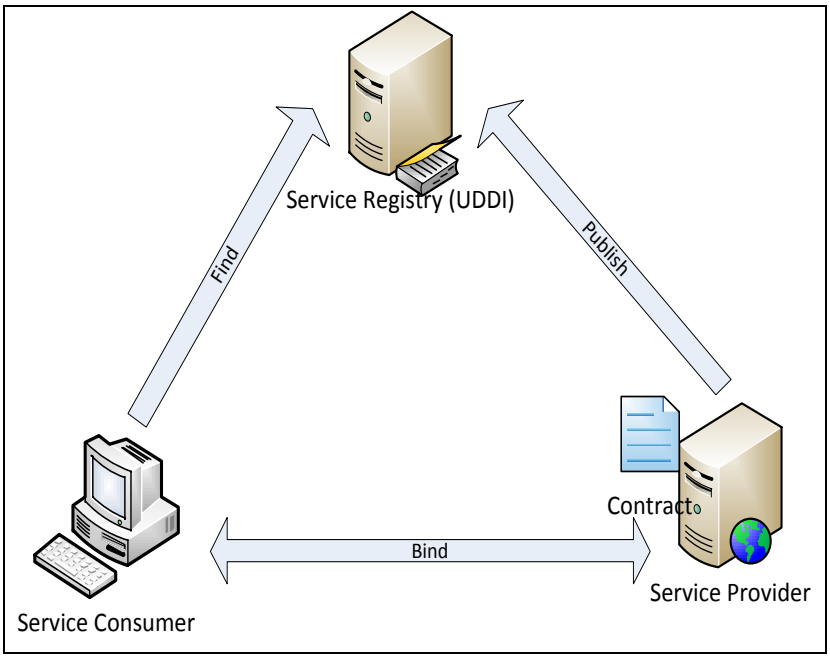

Figure 3-1 SOA

SOA advocates re-usability of Service components by packaging the functionality of IT components into interoperable services [40]. Each component in SOA is associated to a particular action and a particular service may involve combination of many components working together [40, 51]. SOA gained immense popularity due to its packaging ability which provided quick web implementations of the legacy systems in business organizations [15].

There are three important WS technologies which make the implementation of SOA possible [14]:

1. *Universal Description Discovery and Integration (UDDI):* It is a registry where SPs publishes the services that they are offering. The registry contains the URL for different SPs which can be downloaded by consumers to contact the SPs.
2. *Web Service Description Language (WSDL):* This is the contract which defines the service endpoints of a WS. A consumer can retrieve a WSDL document by contacting the SP to generate client stubs to consume the service

3. *Simple Object Access Protocol (SOAP):* SOAP is an XML based protocol which enables communication between WSs in a platform independent manner primarily over HTTP [44]. After retrieving the WSDL document from the SP a consumer can immediately start consuming services by sending SOAP messages to the SP [14].

Although, SOA helped developers achieve interoperability among services it had certain limitations. One of the key limitation of SOA is the use of highly verbose XML based SOAP messages to communicate which consumes lot of CPU affecting overall system performance. [12, 50] Another limitation of SOA is its standard based approach which required developers to follow many standards to achieve platform interoperability between WSs [12, 50].

The limitations of SOA to build scalable systems were outlined by Lee et al. [50]. Their research identifies that SOA implementation at Telco has reduced performance under heavy system loads [50]. To counter these problems an alternative approach called ROA (Resource Oriented Architecture) was introduced [17, 50].

3.4 Resource Oriented Architecture (ROA)

Resource Orientation (RO) was first introduced as a guideline for transition from HTTP/1.0 to HTTP/1.1 [17]. In RO, the focus shifts from identifying IT components as services to identifying each entity explicitly while still preserving the decoupling concepts of SO [17]. According to RO any entity which can be defined as a noun can be considered as a resource [17].

According to Overdick [17] RO is merely an additional set of constraints enforced over SO. The ROA as explained by Overdick [17] expose resources instead of services to consumers with a fixed number of allowed operations on them, sometimes also called as *uniform interface*. Each resource has a URI associated with it, following which a consumer can access the resource [17].

Lee et al. [50] report increased adaptability, openness and performance of existing SOA systems in Telco by implementing a Resource Oriented Architecture based on Overdick ROA style [17]. Their research establishes that scalability issues of SOA which were mainly caused due to the parsing of heavy SOAP messages can be overcome by ROA based implementation [50].

RESTful WSs are an ideal example of a ROA implementation [30].

3.5 REST

The term Representational State Transfer (REST) [31] was first introduced by Roy Fielding in his doctoral dissertation. Fielding [31] described it as a set of architectural principles which can be used to develop highly distributed hypermedia systems over the Web.

It is sometime referred to as a *simpler alternative to SOAP and WSDL based WSs* [1]. Due to its simplicity to use and deploy, REST has gained immense popularity among WS developers and is fast becoming a *de-facto* for developing Web based WSs [1, 27].

The concept of REST has its roots in RO where anything which can be described as a noun is treated as a resource, has a particular state at a given time and can have multiple representations [37]. A Restful system is characterized by the following six design principles [27, 37]:

1. *Everything is a Resource:* Anything in the system which can be named as a noun should be considered a resource and given an ID.
2. *Associate URI with Resources:* Every resource is associated with a URI to facilitate access and communication.
3. *Uniform Interface:* The resources could be accessed and manipulated using a predefined set of operations described by HTTP verbs [19]. For example, a GET on a resource returns a requested representation of the resource and a DELETE will destroy the

resource permanently etc. Other allowed operations are POST, PUT, PATCH, HEAD, and OPTIONS.

4. *Stateless Communication:* The communication between consumer and the service should be kept stateless. Although resources can themselves be stateful, no state data about the previous interactions with the client is preserved on the server side. Therefore, making REST system highly scalable.
5. *Allow multiple representations:* A resource could have multiple representations. This allows the flexibility for client to access data in the format they can process. For example, depending on the request from the client a resource can be returned either as an HTML, XML or a TXT document.
6. *Link things together:* Sometime referred to as HATEOS (Hypermedia as an engine of application state) requires a resource representation to contain URI hyperlinks to other resources which outline the steps a consumer can take next [28].

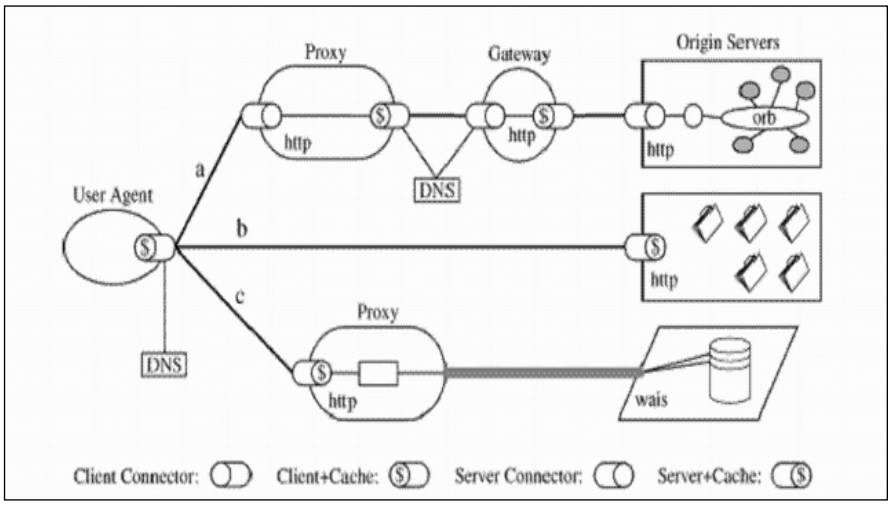

Figure 3-2 REST process view architecture [31]

RESTful WSs follows the principles of REST to build WSs over HTTP. The increased popularity of RESTful WSs can be attributed to its use of well-established W3C/IETF standards like HTTP, XML, URI and MIME [2]. Most programming languages have built-in support for designing HTTP client and server applications, so REST being HTTP dependent is automatically supported by default [5]. The small development and testing cost of RESTful WS is another factor for its increased popularity. Developers can test their applications using an ordinary web browser without the requirement for special software. The use of URIs to represent resources also eliminates the cost of publishing service description into a registry as done in SOA to discover them. The stateless communication between server and consumer eliminate performance bottlenecks and provide a much more scalable system where more clients than a SOA implementation can be served at the same time [5].

RESTful WSs also have in-built support for *caching, clustering and load balancing* which make this particular style an ideal alternative to SOAP based WSs [5].

3.6 Cloud Computing & Mobile Devices

According to the Cloud Computing website [7], "Cloud Computing" is defined as "Internet-based computing whereby information, IT resources, and software applications are provided to computers and mobile devices on-demand". Qian et al. [23] identifies five major technical features of Cloud computing: *large scale computing resource, high scalability and elastic, shared resource pool, dynamic resource scheduling and general purpose.* Although, Cloud is primarily used for storage (Amazon S3), it can also be used to host WSs for ubiquitous access (Google App Engine, Amazon EC2) [48].

In parallel to the growing popularity of Cloud computing and WSs over the past few years, the consumer market for handheld devices including feature phone, smartphone and tablets has

also grown rapidly [29, 34]. Due to this more consumers will access WSs using their handheld devices in future. However, the resource constrained mobile device with limited battery life, processing capabilities and intermittent wireless connection is a major obstacle to achieve ubiquitous WS access [2, 3].

Recent years have attracted lot of interest from researchers to come up with the best possible implementation approach for WSs on mobile devices. In his work, Christensen [6] uses cloud computing to develop context aware mobile RESTful WSs. He outlines that the Cloud not only provides storage space for resource constrained mobile devices but also serves as a processing platform [6].

Hamad et al. [18] concludes that performance of WSs on mobile devices using RESTful implementation is superior to SOAP based WS because of smaller message size and high response time. Similarly, Alshahwan et al. [2] compares service response time, memory consumption and number of concurrently serviced requests for both SOAP and RESTful based WS and found RESTful model to be far superior for mobile hosts.

Therefore, applications hosted on cloud can help develop *light client* models which can then be ran on devices with limited processing capabilities.

3.7 Enterprise Service Bus (ESB)

There are many conflicting definitions for an ESB and people are still confused when they are asked the question what exactly an ESB is. The first formal definition was provided by analysts from Gartner in 2002. They described it as a *universal integration backbone* with in-built capabilities for reliable message passing, routing and message transformation [11]. Mule, an open source project is an example implementation of ESB concepts [25].

Schmidt et al. [35] describes ESB as an essential component for realization of the functionalities offered by a SOA. In their research they mention that SOA's triangular model of "publish-find-bind" as depicted in Figure 2 is incomplete without the introduction of an ESB [35]. Schmidt's model of a SOA with ESB is shown in Figure 3-3. They consider ESB from an Enterprise perspective where multiple applications having different semantics communicate through the use of message passing [35]. Therefore, their view of ESB is an infrastructure that interconnects distributed applications and services in an enterprise [35].

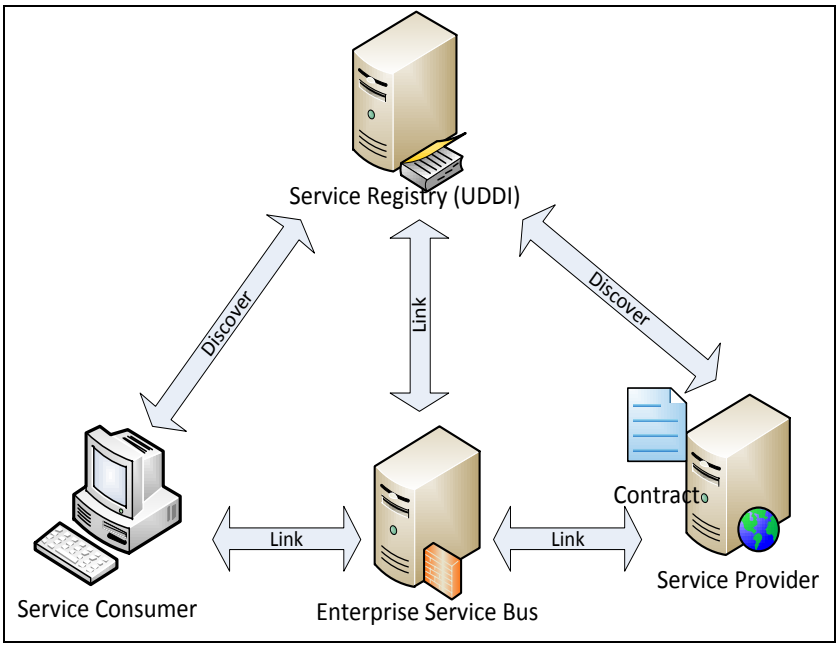

Figure 3-3 SOA with ESB [35]

According to Menge [13], the ESB combines the power of Enterprise Application Integration (EAI), Message-Oriented Middleware (MOM) and SOA to provide seamless communication among service in an enterprise.

However, Richards [24] in his presentation "The Role of Enterprise Service Bus" uses a different approach to define ESB in terms of its capabilities. According to Richards a software

product to be called an ESB could provide any or all of the following core capabilities, namely: *Routing, Message Transformation, Message Enhancement, Protocol transformation, Service Mapping, Message Processing, Process Choreography, Service Orchestration, Transaction Management, Security* [24].

Although, the concept and definition of ESB varies from vendor to vendor [13, 24], ESB is instrumental in its role to realize the full potential of SOA by [8, 13, 35]:

1. **Providing Easy Management of WSs:** Changes to the underlying services can be managed by adapting ESB to those changes rather than changing the clients.
2. **Reducing Client code complexity:** Client doesn't need to know exactly where the service is hosted; ESB routes client's request to the particular SP.
3. **Abstracting implementation details of the service:** A client doesn't need to know how a service is implemented; rather it focuses on what service does.
4. **Providing extra layer of security:** ESB shields services from Denial of Service attacks by acting as a filter for such requests.
5. **Managing Workflows:** ESB manages the flows of control in a service request which involves invocation of more than one service endpoint.

3.8 CAP Theorem

The challenge to handle state was considered a minor issue by the WS community till 2000. However, Brewer's [10] introduction of CAP theorem in 2000 transformed this into a major challenge. Before its introduction it was considered that WS dependability and scalability can be enhanced by moving services into cloud, and/or by replicating the service [60]. Brewer's [10] in his CAP theorem mentions that in a physically distributed system it's impossible to achieve

consistency, availability and fault tolerance all at the same time According to Royans [33] these three factors are described as follows:

- *Consistency* refers to the guarantee of the similar state at all client nodes at same time.
- *Availability* refers to the guarantee that if a node in the distributed system become inaccessible due to failures; other nodes must continue to operate.
- *Fault tolerance* refers to the guarantee that the distributed system is built to "arbitrarily allow loss of messages sent from one node to another".

Brewer [10] also stated that at best any two above mentioned factors are achievable at the same time but only at the cost of the third factor. A formal proof of the CAP theorem was provided by Gilbert et al. [16] in their work where they discovered that it is necessary to relax the consistency constraints to achieve a WS which is highly available and partition tolerant.

3.9 Relational Database Views

According to relational databases, Views are a well understood concept. According to Connolly [39], "a view is the dynamic result of one or more relational operations on the base relations to produce another relation". Therefore, a view is a virtual relation produced as the result of executing a SQL query which doesn't actually exist in the database.

Date [4] defines views as stored queries which are executed on the base relations to generate a table which provides a window to the base relation. A view is a powerful mechanism which can be used for improving security in database system by limiting user's access of the data in the

base relation [4, 39]. Other benefits of views include: Data independence, reduced complexity, data integrity etc. [39].

According to Connolly [39] views are dynamic, which indicates that any changes made to the base relations automatically updates any views defined on those base relations. In the same manner, any updates made on the view should translate these changes to the base relations. However, according to Connolly [39] and Date [4] most relational database systems impose the following restrictions on the allowed modification operations:

- An update on a view is allowed if the view is implemented on a single base relation and contain the primary key or a candidate key of the base relation
- An update on a view is not allowed if the view is defined on multiple base relations
- An update on a view is not allowed if view involves an aggregation or grouping operations in its definition.
- An update on a view is not allowed if it doesn't preserve the symmetry of relation [4]. For example, a DELETE operation on a view involving two base relations must DELETE tuples from both relations.

It is safe to say that databases have a strong resemblance to the RESTful system due to their dependence on state. For example, a REST system is driven by the state of its resources i.e. inserting/ deleting/ modifying the resources will changes the state of the REST system. Similarly, a database is changes its state when the data is inserted/ deleted/ updated in its base relation.

Riva et al. [3] identified network latency as a key performance bottleneck while designing WSs for mobile devices using REST. In their work they suggest the use of views to counter the negative side effects of network latency. According to Riva et al. [3] a view can be created by

creating new URIs which provides application specific data by augmenting or bypassing the core API. To the best of our knowledge this was the first time when the concept of views was introduced in WS domain implemented RESTfully.

3.10 MapReduce

"MapReduce" was first introduced by Google in 2004 [9, 36]. It is a "programming model and an associated model for processing and generating large data sets" [9]. Google uses it in their web index to process large clusters of data sets by distributing the load of processing over multiple machines [20]. The concept of "MapReduce" draws inspiration from functional programming domain where Map and Reduce function are commonly used [9]. However, these functions holds different semantic in a "MapReduce" framework [22]. According to Lammel [22], a map step consists of the following:

- *iteration over the input*
- *computation of key/value pairs from each piece of input*
- *grouping of all intermediate values by key*

And a reduce step consists of the following [22]:

- *iteration over the resulting groups*
- *reduction of each group*

The biggest advantage of the "MapReduce" model is to support parallelism, helping carry out map and reduce function on multiple machines at the same time [20, 45]. Dean et al. [9] in their implementation of "MapReduce" was able to demonstrate the efficiency to this technique to be employed over a large cluster of machine. Based on the results they [9] concluded that

"MapReduce" can easily support load-balancing, provide fault-tolerance and local-optimization to achieve highly scalable systems.

Since its introduction this model has gained immense popularity leading to development of libraries in many different programming languages including C++, C#, Erlang, Java etc.

3.11 Summary

After reviewing the literature, it can be seen that a shift to delivering WSs RESTfully over SOA/ WS* style is taking place. Moreover, considering the growing number mobile consumers of WSs suggests that the WSs will likely be developed RESTfully in the near future [6, 18, 28]. The communication model of REST uses standardized CRUD operations which helps builds systems which are simple yet highly scalable [1, 27, 31]. Moreover, due to the pervasiveness and support for HTTP across all major platforms it's easier for a WS designer to target its audience independent of their devices. However, as the system client base grows security and scalability of the system becomes a major issue. Based on the current literature findings a middleware has not been implemented for ROA systems. However, the use of middleware architecture like ESB in SOA is well-known and proven to build system which are scalable and secure [13, 24, 35]. To further enhance the scalability and elasticity of a system under peak loads the service along with the middleware can be hosted in a Cloud which provides large scale computing resources.

The concept of views in relational databases is well-known for controlling access to databases [4, 39]. After a closer look at the database and REST system one can say that they bear close resemblance to each other due to their dependence on state. An idea to use views in REST was suggested by Riva et al. [3] to help remedy network latency issues. However, the application of view concept has never been tested with resources in REST to control access based on client privileges. The concept of views seems to work well with systems which are implemented using

"MapReduce". Apache's CouchDB [] is an ideal example of a highly scalable, fault-tolerant system with optimum load balancing features which embraces MapReduce at its core. Therefore, to achieve a middleware which is highly scalable with load balancing capabilities "MapReduce" can be a solution to define views on resources in REST.

According to the reviewed research the findings to the challenges identified in problem statement are summarized in Table 3-1.

Table 3-1 Solutions to the challenges from reviewed literature

Challenges	Findings
Load balancing	REST is scalable, preferred choice over SOA, best suited for mobile devices with limited processing capabilities [2, 3, 6, 18, 29]. Cloud Computing provides rich pool of computing capabilities to provide seamless service which are scalable with low downtimes [2, 6, 18, 29, 23, 34, 48]. According to CAP theorem, only two features among Consistency, Availability and Partition tolerant is achievable when looking to design a distributed system [10, 16, 33].
Access Control	Views in database are known for implementing client specific view on a table to control client access [4, 39]. To provide access control for SOAP based WSs a middleware like ESB is a preferred option to provide access control [8, 11, 13, 24, 25, 35].
Virtual Resources	A view in database creates virtual tables and control user access [4, 39]. There is a similarity between Database and REST due to their dependence on state [4, 27, 37, 39].

Resource Management	MapReduce is a well know technique to querying large sets of data by distributing the work across multiple machines e.g. Google uses it to produce their search results [9, 20, 22, 36].

However, there are still **open questions** related to the development of middleware architecture for REST and defining and serving views to clients:

1. How to define view resources programmatically?
2. How to manage the state change in view resources when an underlying resource on which the view is defined has changed?
3. Should the views be stored in memory as a separate resource or generated on the fly?
4. How and where to define what CRUD operations are allowed on the view resource.
5. How different CRUD operations will propagate to the base resource when a CRUD operation is executed on a view resource?
6. How can the access of client be controlled to restrict it to viewing only certain resources?
7. Where the logic of defining views resources should be maintained?
8. How the middleware will handle and distribute loads?

According to the knowledge gathered based on the reviewed literature, this is the first known attempt to apply the database concept of views onto resources in a REST.

CHAPTER 4

DESIGN AND ARCHITECTURE

4.1 Example scenario

To understand the relevance and functionalities of Enterprise Resource Bus, I will be using the example of a hospital information system (Figure 4-1) throughout this chapter.

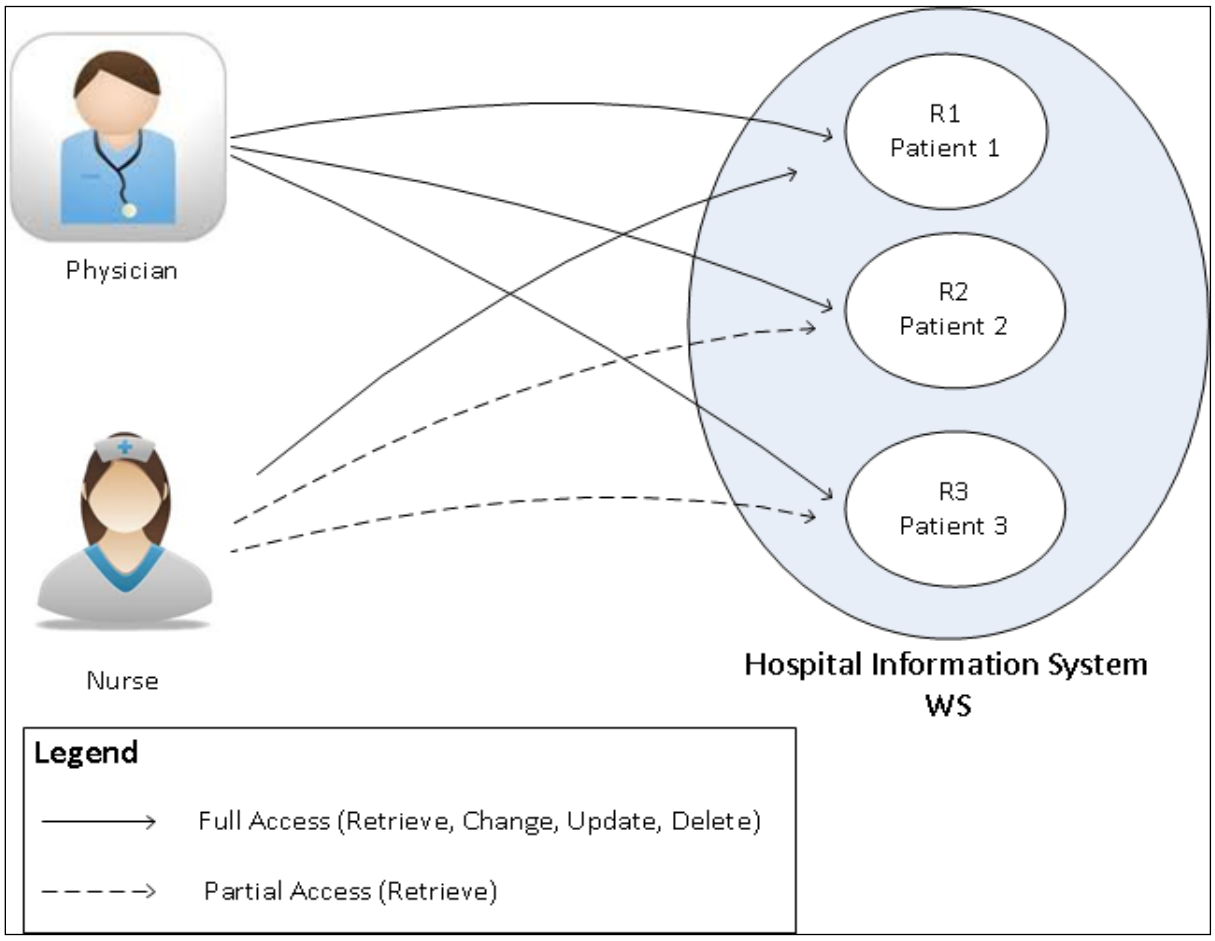

Figure 4-1 Hospital information system- an example WS

1. **Problem Space Description:** As shown in Figure 4-1, let us assume there are two different type of consumers of our hospital WS, namely: the physician and the nurse. For simplicity, let us assume this WS manages the data of three patients in three different

resources. Let R1 contains information related to patient 1, R2 contains data related to patient 2 and R3 contains the data related to patient 3.

2. **Service Requirements:** We want our system to provide full access (Retrieve, Change, Update, and Delete) to physician on all three patient resources. On the other hand, we want to restrict the nurse access to patient resources, providing only retrieving capabilities on resource 2 and resource 3. So, the challenge here is how do we provide these functionalities without changing the underlying resources.

4.2 Architecture overview

The goal of the Enterprise Resource Bus (ERB) architecture is to provide a mechanism to explore and implement the idea of views on resources. Figure 4-2 shows an overview of ERB and where it fits in the context of a RESTful environment. The architecture consists of three main components: the clients, the resources and the ERB.

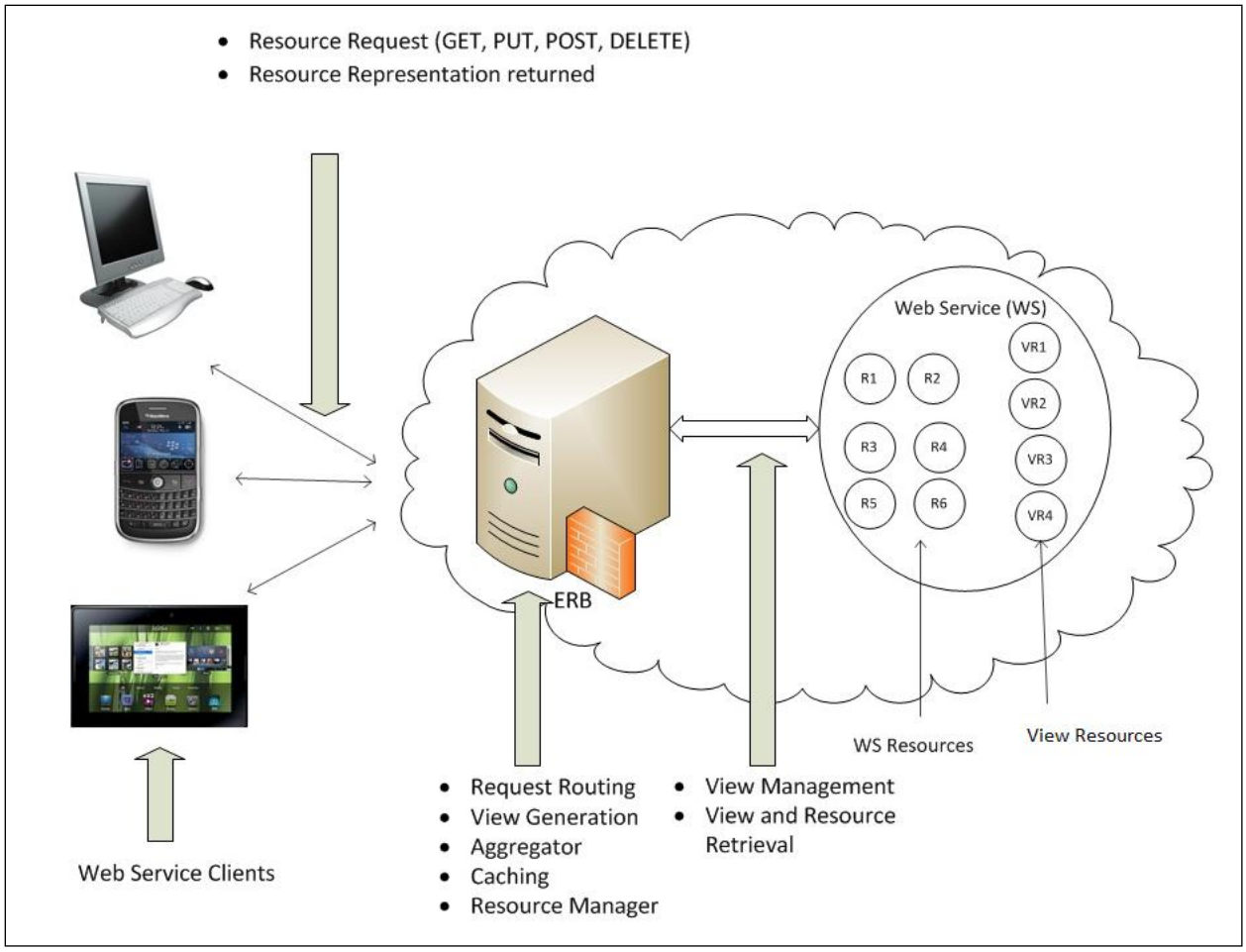

Figure 4-2 Overview of ERB and RESTful WS

When a request for a resource is received from a client it is first encountered by the ERB, which then routes the request to the appropriate resource. In response to the request a representation of the resource is returned by the WS. A representation of the resource is the format in which the data is encoded in, e.g. XML, text/html, JSON etc. The returned representation of the resource can be an atomic resource or a view resource. Whenever an update is made to the WS resources, ERB pulls the new updated resources state into its cache to maintain data consistency. A client is required to issue a HEAD request to check if the resource state has changed, if so an updated copy of the resource can be sent to the client.

4.2.1 Web Service Clients

Web Service clients are implemented using the thin client model. In a thin client model, a client does very limited computations and most of the logic to handle the heavy processing is outsourced to another machine. In our architecture, we want most of the processing related to the WS be outsourced to the ERB. Thin client model is ideal for providing WSs to mobile consumers due to their limited processing capabilities and high power consumption

4.2.2 Resources

A RESTful WS exposes resources to the client which can be accessed using a uniform interface (i.e. CRUD operations of GET, PUT, POST and DELETE). Resources are accessed by issuing an HTTP request to the WS, which in response returns a representation of the resource (Figure 4-3).

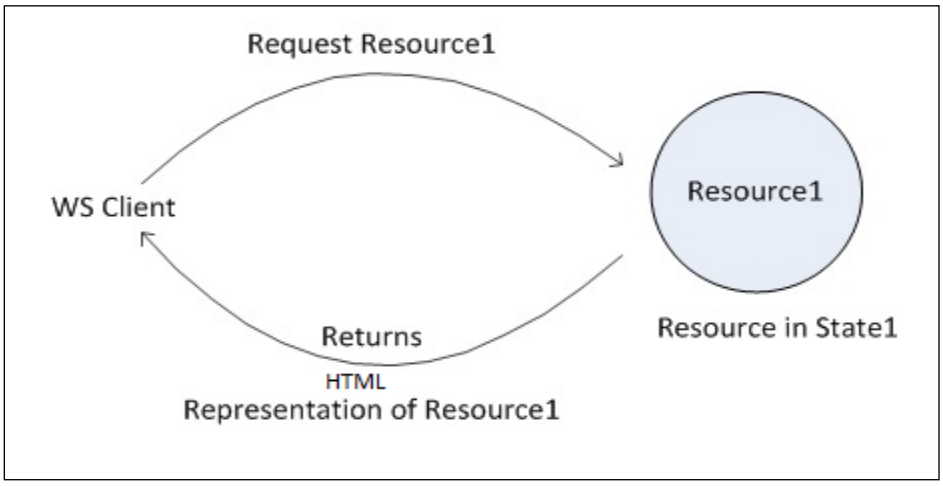

Figure 4-3 Accessing resource in a RESTful WS

According to our architecture, there will be two different types of resources in a REST, namely:

a. **Atomic Resource:** An atomic resource is anything in our system which can be described as a noun and has a unique Identifier (27, 37). Furthermore, it's a standalone entity in a REST environment on which certain CRUD operations such as GET, PUT, POST, DELETE are allowed. For instance, a PUT request from a client on an atomic resource modifies the resource and returns a confirmation with HTTP response code to the client indicating whether the resource has been updated or not.

b. **View Resource:** Similar to the concept of views in relational databases, a view resource in REST can be defined on a set of atomic resources. The virtual relations in database are stored as SQL procedures which are executed when a request to access the virtual relation arrives [39]. The view resource will also be stored as a view definition with the logic to generate the resource when a request to access it arrives. So, whenever a new view resource is defined in our WS the administrator will have to define the set of resources on which the view is defined alongside with the logic to generate the view resource.

   The definition of view along with its associated set of resources is managed by the Resource Manager (RM). It will be the responsibility of the administrator to update the view definition if necessary.

Example Reference: Revisiting our example scenario of a Hospital information system, the atomic resources in our WS will be resource R1, R2 and R3 (Figure 4-4). These are the resources of our WS on which any HTTP operation (GET, PUT, POST, DELETE) can be invoked. These operations can be invoked by anyone (nurse or a physician) if they know the URIs of the resources. This automatically allows the physician to have full access to these resources. However, we want the nurse to have only retrieving functionality on resource R2 and R3. This is where our concept of view resource can come in handy.

Therefore, for nurse clients a customized view resource is defined on a set containing resource R2 and resource R3 (Figure 4-4). This view resource has its unique URI and is an aggregation of resource data from R2 and R3. Furthermore, we want the nurses to have only retrieving access on the view resource and not change it; the allowed operation on the view resource is set to GET for that particular nurse. Therefore, whenever the nurse needs to access patient resources R2 and R3, he/ she will send a GET request to the URI of view resource rather than issuing a GET on R2 and R3 separately.

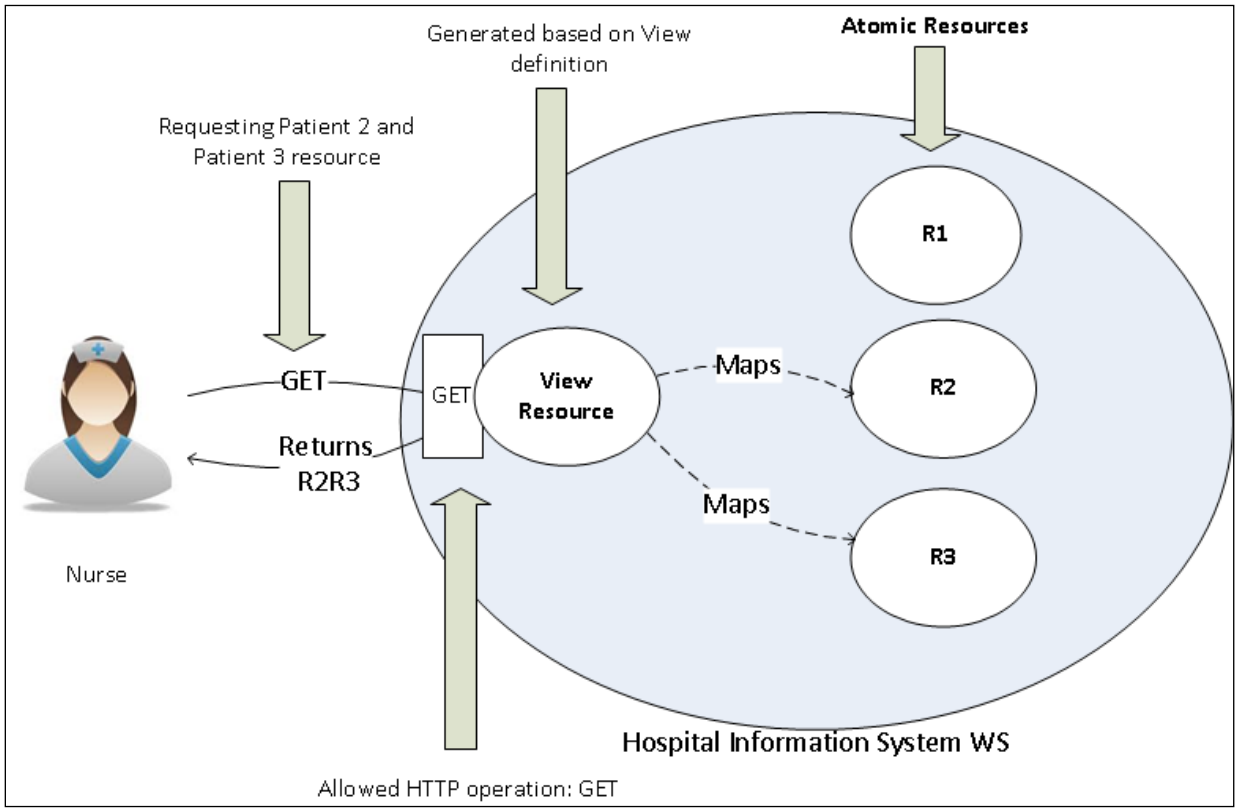

Figure 4-4 Example of a view resource in a Hospital information system

4.2.3 Enterprise Resource Bus (ERB)

The Enterprise Resource Bus (ERB) is a pattern that builds on the model of the Enterprise Service Bus (ESB). In addition to emulating some features provided by ESB, the ERB is

designed to provide better support for RO Systems. Hence, the name Enterprise Resource Bus. Figure 4-5 shows the various components of ERB.

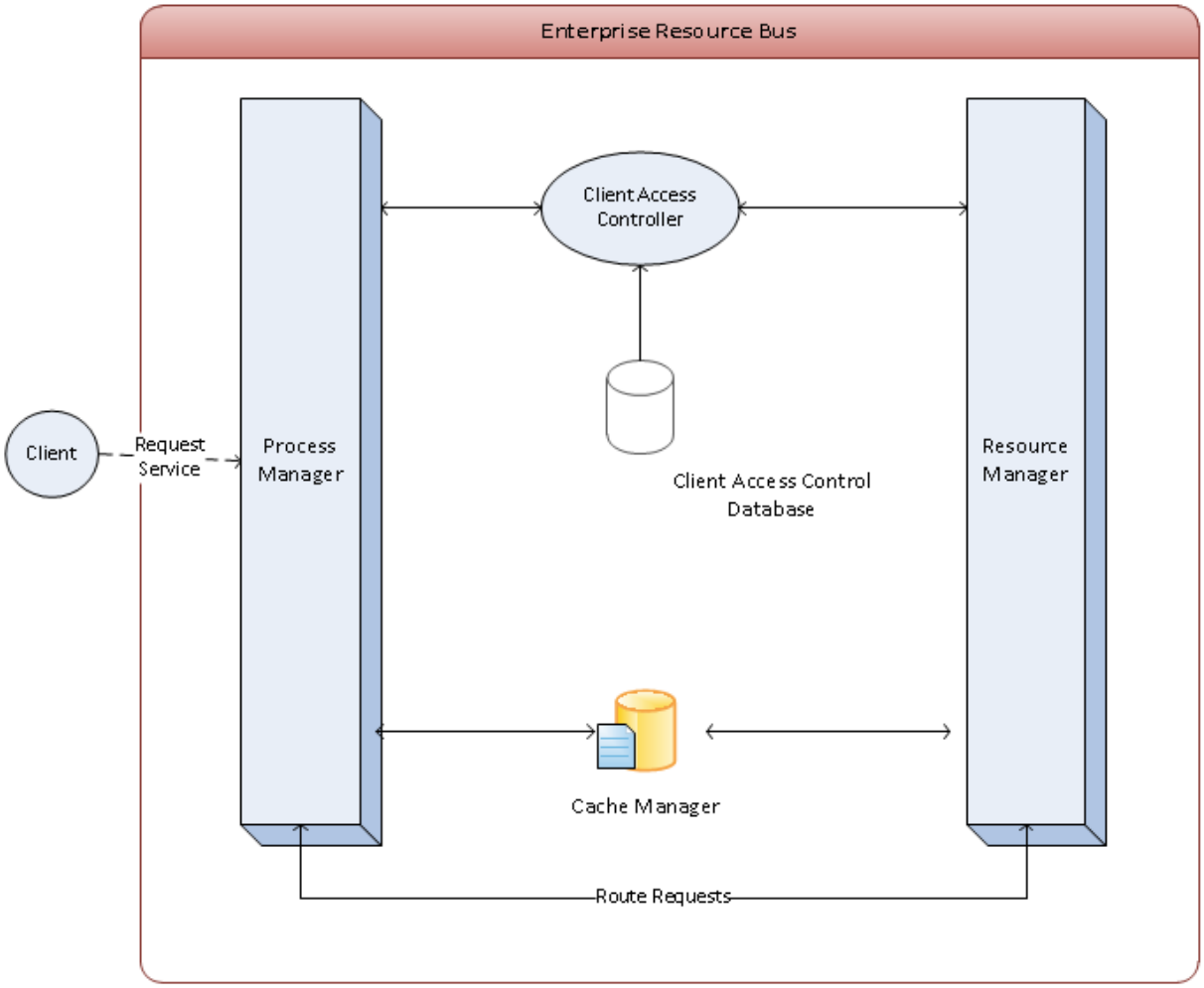

Figure 4-5 Components of ERB

A brief description of each component of ERB is described as follows:

1. **Process Manager (PM):** This component acts as a first point of contact for the clients requesting the WS.

   The PM is also responsible for handling all the communications among other components of the ERB. Based on the type of request from the client, PM contacts the

necessary components within ERB. For instance, a client requesting access to a particular resource will be checked by PM against Client Access Controller (CAC) to verify that the client has appropriate rights to access the resource. All other components replies back to the PM, which then forward an appropriate response back to the client.

2. **Client Access Controller (CAC):** The CAC makes sure that the client requesting access to the service is actually an authorized client. The CAC has access to Client Access Control Database (CACD) which maintains a list of all the clients of the WS. CACD handles the logic that can create a new client, update or delete an existing client.

   Another important job of CAC is to identify what resources are accessible to a given client. It also makes sure that the client is trying to invoke correct HTTP operations on the resources. The information regarding allowed operation on resources is stored in CACD. If the client tries to invoke an operation which is not allowed on a given resource, a response message of "403 Forbidden" is sent back to the client.

3. **Client Access Control Database (CACD):** This component maintains a client database with information like client id, list of accessible atomic and view resources with allowed HTTP operations on each resource.

   The database is implemented as a sequence of tuples with each tuple containing information specific to a client. The motivation behind maintaining a client database is to increase flexibility and security in the system. Depending on each client profile more or less access can be granted to the resources by adding/ removing HTTP operations from the list of allowed operations thereby achieving more flexibility.

On the other hand, security of the system is increased by limiting access of the client to certain resources. This in turn reduces cases of inadvertent or deliberate changes to the service as each client is only liable to the resources accessible to her.

Example Reference: Revisiting our example of a Hospital information system described at the beginning of the chapter, let's imagine our WS maintains 3 patient resources: P1, P2 and P3 with two clients, nurse N1 and nurse N2. Let P1 and P2 be the patients of geriatric division while P3 is a patient of psychiatry division (Figure 46). Similarly nurse N1 is a nurse devoted to geriatric division while nurse N2 be a nurse in psychiatry division. Now, we want the nurses to be able to access only the patients in their own division. This can be a challenging task in a traditional WS where all the data is served by a single WS. However, according to the proposed architecture the logic of access control to different resources in the system will be handled by CAC component of the ERB. As you can see in the Figure 4-8, Nurse N1 is returned back a representation of the resource P1 because the CACD contains an entry related to N1 with P1 being an accessible resource. However, when a request for P1 is issued by nurse N2, the access to the resources is denied and the client gets a response of 405 (Not allowed). This is because the entry associated to N2 in CACD has only resource P3 as an accessible resource.

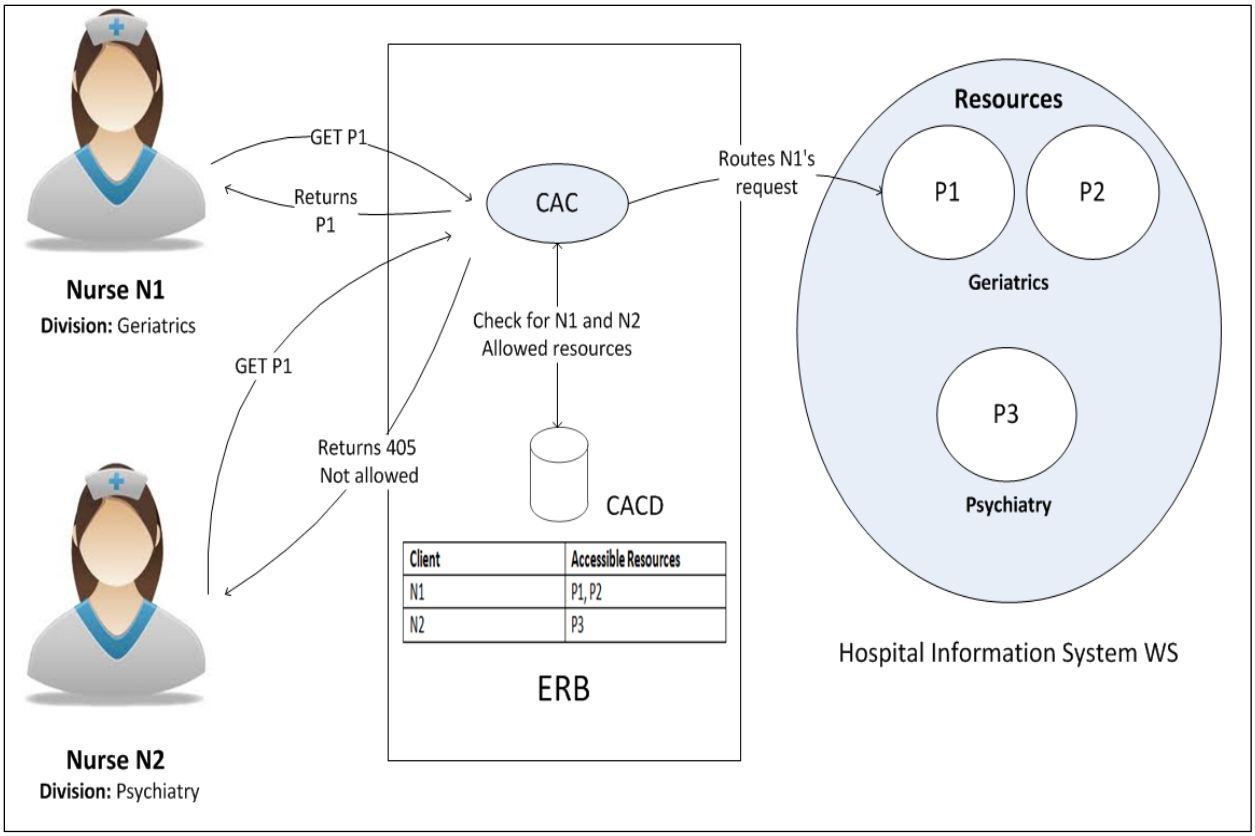

Figure 4-6 Controlling access of clients for different resources in the WS

4. **Cache Manager (CM):** In order to reduce the network traffic and to compensate for infrequent loss of messages, a CM component is included into ERB. To improve the cache effectiveness an event-based callback model is implemented. With the introduction of requirement on each resource to issue a state-change event to ERB, it's possible to keep the most updated version of the resource in the cache.
5. **Resource Manager (RM):** A RM is a link between the Resources offered by the WS and the other components of ERB. The RM is responsible for handling the resource calls from the PM and then to reply back to the clients with a representation of the resource. The view definition are also stored and managed by the RM. The logic to run complex operations like POST on multiple resources is defined in the RM. RM is also responsible

for managing a state record which keeps track of the changes made to all the resources in WS.

4.1 Type of Views

The idea of view resources inspires from the concept of database views. A database view is a stored query which is executed on base relation/s to obtain a new relation called the view. Similarly, a view in a RESTful system is an entity which represents a set of resources. A view can be defined on a single resource or a set of resources. Either way a view is true for a set of resources (single resource being a set of cardinality 1). To define a view in REST this work proposes to use the database approach. Whenever a request for a view arrives, a pre-coded script is executed to produce the view. Based on when the script is executed the views can be one of the following three types:

- **Immediate/ (On the fly generated) View:** As the name suggests the view is generated on the fly whenever a new request arrives. The view exists in memory as long as it's being accessed, otherwise it's destroyed. The script for generating the view is stored in the ERB. This approach is very similar to the view concept in database in which views are only stored as queries and the resulting relation is never stored in the database.
- **Transient View:** In a transient view once the view definition script is executed, a new view resource is created in the system. Similar to other resources in RESTful system, a transient view resource has its own state which is dependent on the state of its set of resources (Figure 4-7). Whenever a new HTTP request for the

resource arrives the view definition script is executed and an updated state of the view resource is returned to the client.

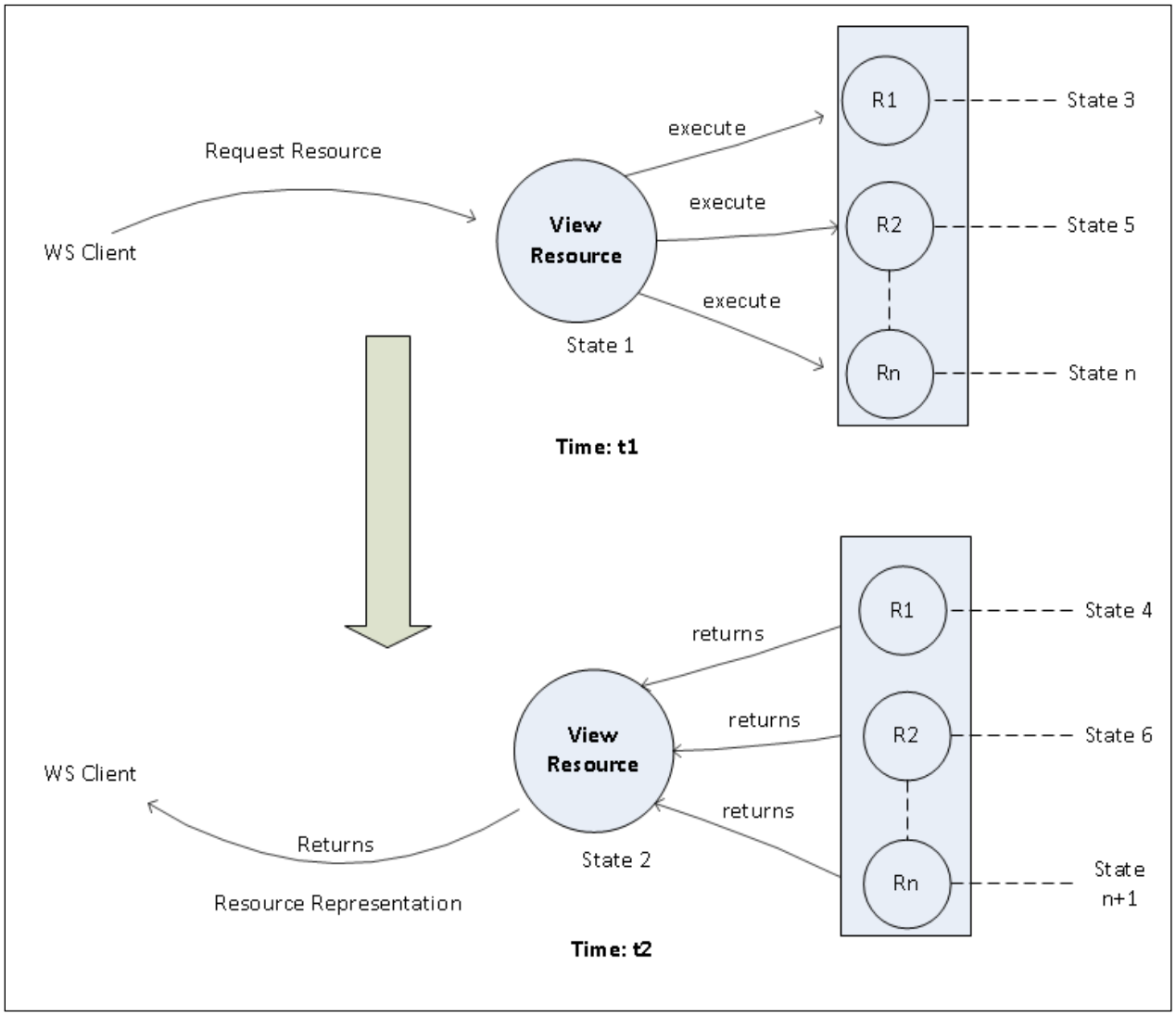

Figure 4-7 Transient View

- **Persistent View:** A persistent view is similar to a transient view as it exists as a new view resource in the system once its definition script has been executed. However, the difference is how the updating of the resource takes place. The view state is always kept updated by following an event-trigger mechanism (Figure 4-

8). Following this model, the set of resource associated with the view pushes their state to the view resource whenever there is a change in their state.

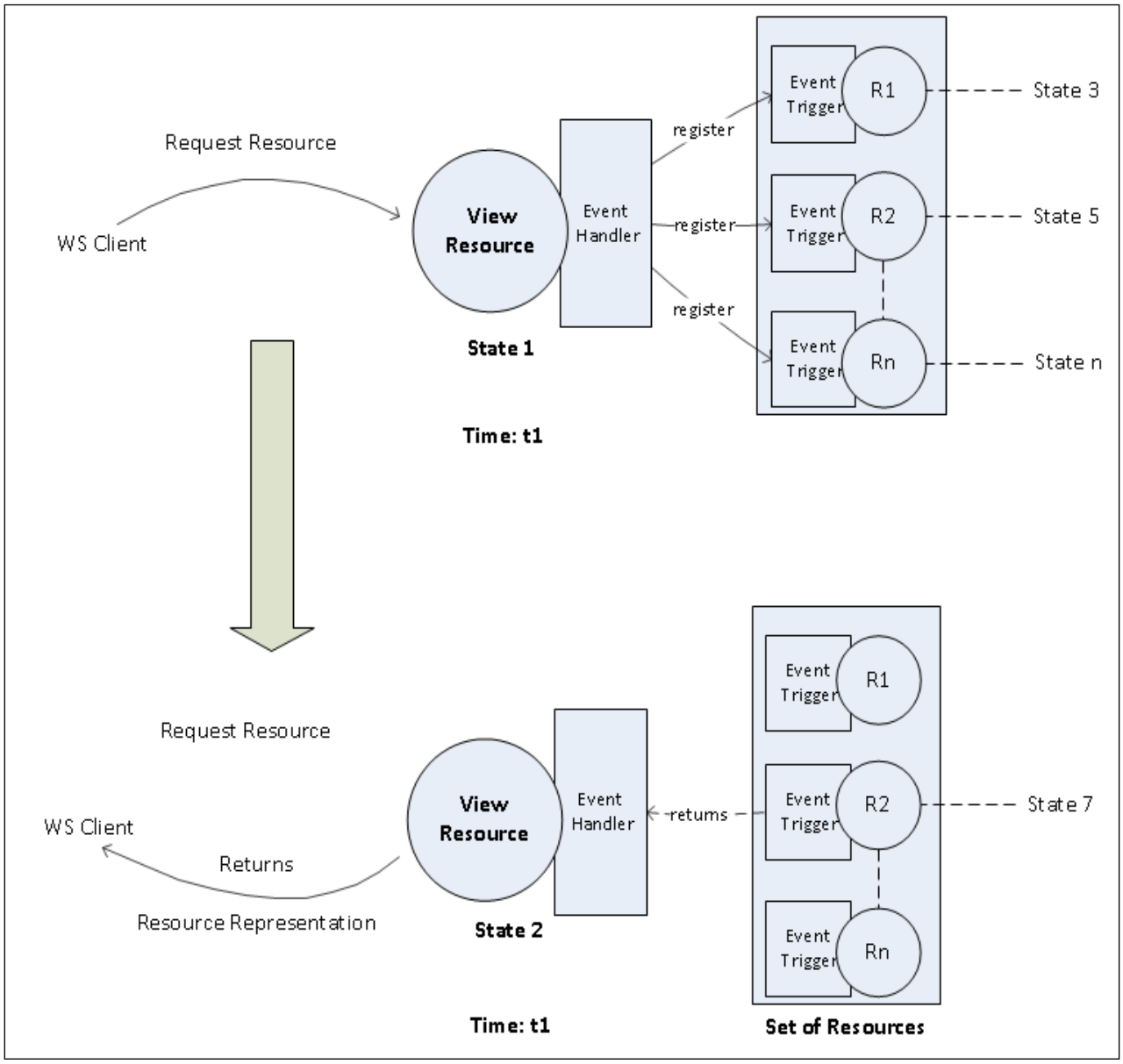

Figure 4-8 Persistent view

4.2 Methodology to generate views

As a proof of concept, this research proposes to use the transient view approach for implementing the views in REST. The view definition will consist of the list of resources on

which it's defined, and the function definitions for handling each allowed HTTP operations on the view e.g. GET function, DELETE function etc. The view upon encountering client requests matches the sent HTTP method to the defined HTTP function definition (Figure 4-9). As the current implementation uses the transient view approach, each view will execute a script to check for any updates to the associated set of resources. The state of view will then be updated if required and the view resource will be returned to the client.

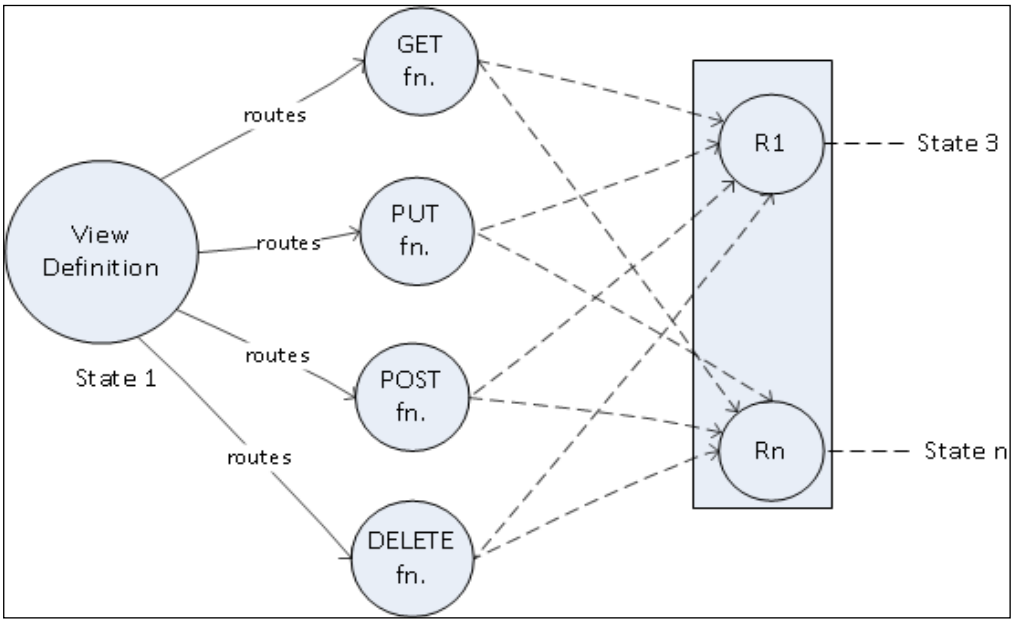

Figure 4-9 Proposed View Definition Model

Now, the question is how do we define these views programmatically? The requirement to generate a view resource is to read all the atomic resource on which the view is based and then combining them together to produce a single output. The semantics of the problem closely matches the methodologies used by "MapReduce" approach. According to "MapReduce", a Map function iterates over a set of input and produces N results and a Reduce function iterates over

the resulting group from Map step to produce a single result that complies with the supplied condition (Figure 4-10).

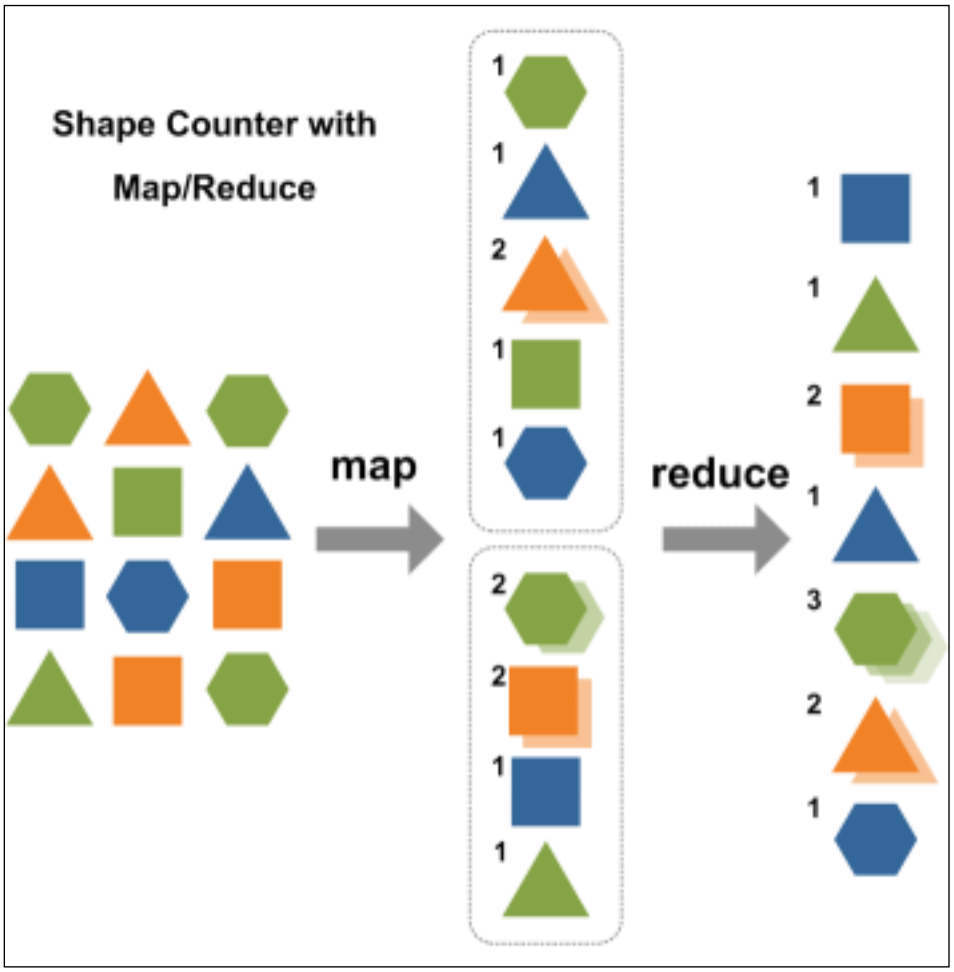

Figure 4-10 MapReduce [50]

As described earlier, the proposed approach defines views on a set of resources which are to be queried and the result is aggregated and returned to client, this research adopts the "MapReduce" approach to define views. Following the "MapReduce" approach, whenever a view receives a client request, the Map function defined in the view will be executed to map the request to all the resources in the set. As a result, a set of resource representation will be returned (Figure 4-11).

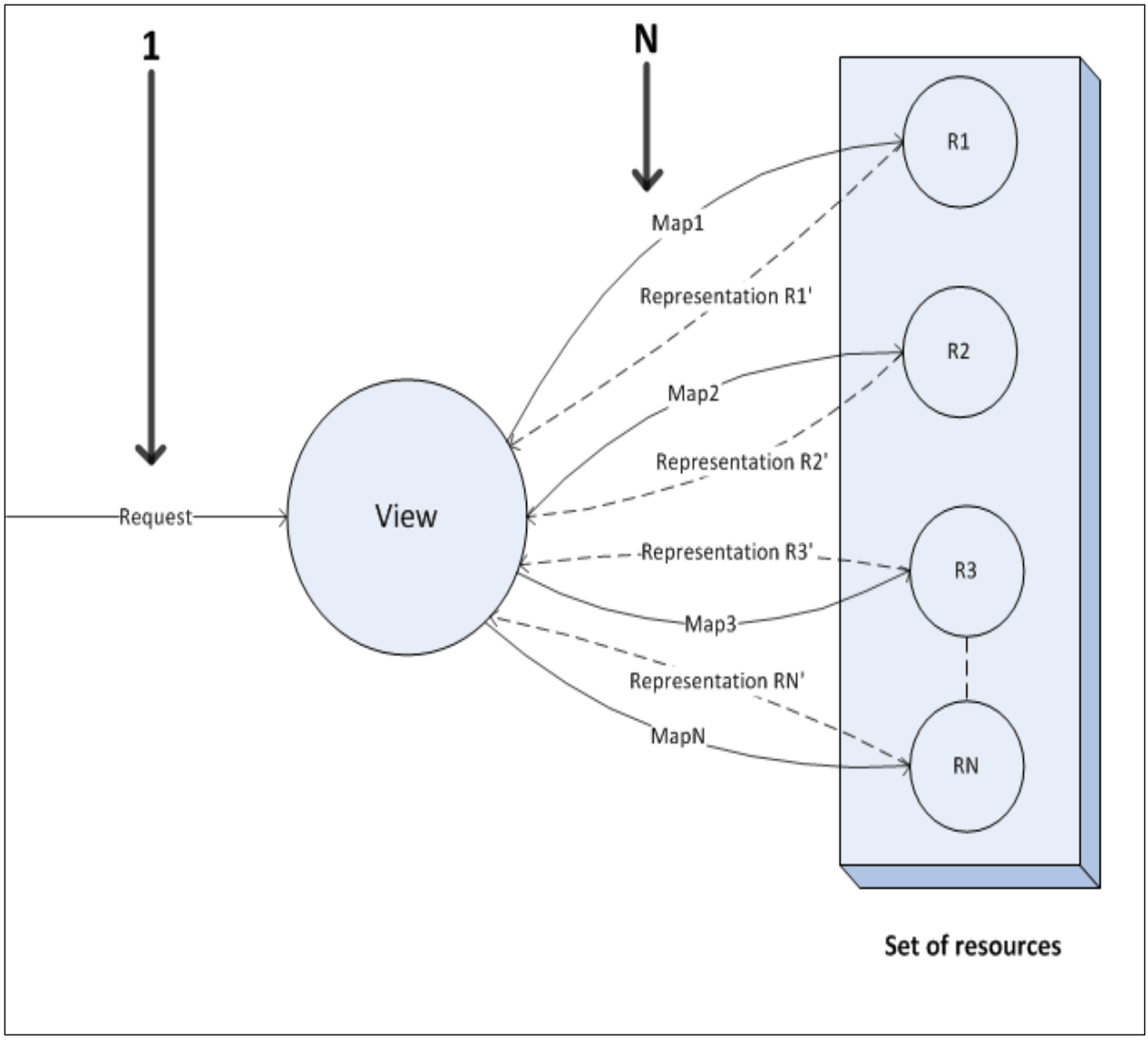

Figure 4-11 Mapping (Step 1)

Once the mapping is done, a reduce function is executed on the set of resource representations returned from the Map step to one aggregate resource. However, as discussed earlier, the views are defined on a set of resources, a Reduce step may be unnecessary if the Map is executed only on a set consisting of a single resource.

In other cases, a Reduce function can be supplied by the administrator to run on the result set of the Mapping the request (Figure 4-12).

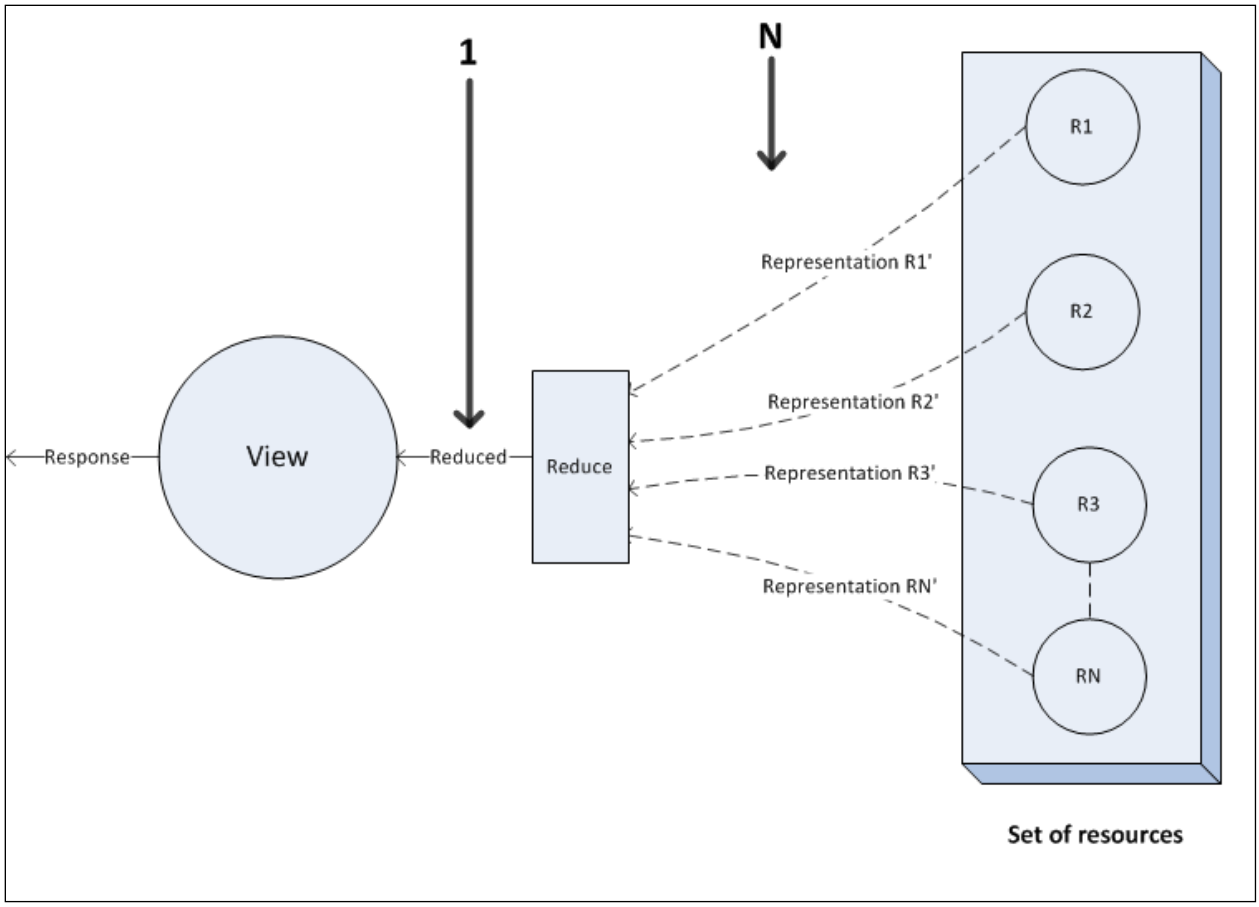

Figure 4-12 Reducing (Step 2)

4.3 Keeping view resource updated

To ensure data consistency, it's important to keep the view resource state updated when the state of the atomic resources changes. However, how do we keep track of these state changes that happen to the atomic resources. There are couple of approaches which can be used, namely:

1. Run the Map function each time a new request arrives and then reduce the resulting set to produce the view; or
2. Use a mechanism which requires only mapping those resources which have changed since last map function call.

The former approach is the simplest approach and can be easily implemented but it will consume more CPU cycles. This can ultimately affect the overall performance of the system when the no. of requests for a view increases exponentially. The second approach draws inspiration from CouchDB's [49] implementation where the view uses a mechanism to only map those documents which have changed since last retrieved, thereby considerably reducing the response time. As this technique has been tried and shown to work well, this research use similar approach to keep track of the changed resources in the set. To achieve it programmatically, an update Erlang record will be implemented and managed by the Resource Manager (RM). Each resource registers its state with the RM if any change in their state happens. The update Erlang record will contain the entry of all the updates that happened to the resource since their last retrieval. So, whenever a new request arrives, the RM will first check for any new entries in the update Erlang record since last map function was executed. If any entries were found, the map function will then be executed against only those resources, the results reduced and sent to the client.

Furthermore, the defined view resource similar to its counterpart virtual relation in databases [39] will have the following restrictions on the update requests (PUT, POST, DELETE) received from the client:

1. Updates on the view resource will be cascaded to its atomic resources only if the view is defined on a single atomic resource.
2. A view defined on a set of resource with cardinality greater than 1 will be read-only.
3. An update on a view defined on a set of resource will be allowed to only change the set of resources on which it's defined.
4. A DELETE request will delete the view but its atomic resources will not be deleted.

4.4 Summary

This chapter provides a detailed explanation of the architecture of our middleware. The middleware will consist of five main components: Process Manager, Client Access Controller, Client Access Control Database, Cache Manager and Resource Manager. The functionality of each component was explained with the help of an example reference to a hospital information system Web Service.

Furthermore, the idea of view resources in REST is introduced and discussed by defining a view resource. We then classified the view resource into three different types: immediate view, transient view and persistent view. As a proof of concept this research implements the transient view approach due to its simplicity. In order to define views programmatically in REST our architecture will use “MapReduce” approach to iterate over a set of resource. Lastly, the key issues related to updating the views are presented and discussed.

CHAPTER 5

IMPLEMENTATION

The various components of Enterprise Resource Bus (ERB) were implemented using Erlang [55]. Erlang is preferred over other languages because of its ability to handle peak loads with high efficiency, low downtime and quick response time. Moreover, the entry point for our RESTful Web service is handled by Yet Another Web Server [52] (YAWS) which is also written in Erlang. This ensures code compatibility with our implementation.

5.1 YAWS integration

Enterprise Resource Bus consists of various components with their own functionality. The requests made to our REST web service are first handled by a YAWS server running on port 9091, which have access to all the components (modules) of the ERB. This approach helps us achieve an easy to implement secure entry point for the WS and encapsulating the implementation details of various components from the WS clients. The received request from the client is parsed by YAWS and then passed onto the Process Manager component of the ERB. In order to setup YAWS to handle and parse incoming client request "Appmod" feature of YAWS is used. By specifying an Appmod in YAWS.config file the URI path will be directed to the specified Erlang module that will handle/ parse the incoming requests. To route all incoming client request URIs containing "hospitalWS" in their path to the Erlang module- "myWSEntryPoint" the configuration shown in Figure 5-1 is used.

```
<server hospital>
        port = 9091
        listen = 0.0.0.0
        docroot = "C:\Program Files (x86)\Yaws-1.95/www"
        appmods = <hopsitalWS, myWSEntryPoint>
</server>
```

Figure 5-3 YAWS configuration file

The Erlang module- "myWSEntryPoint" is shown in Figure 5-2. The functionality of this module is to parse the incoming request and then start the execution of the request in a new thread by spawning "processRequest' function contained in "processManager' module.

```
-module(myWSEntryPoint).
%%
%% Include files
-include("yaws_api.hrl").

%%
%% Exported functions
-export([out/1]).
%%
%% Local Functions

out(A) ->
       %% Extract information from the record Arg
       Request= A#arg.req,
       EntryPid= A#arg.pid,
       Uri=yaws_api:request_url(A),

       %% Extract Information about the HTTP Request
       Method= Request#http_request.method,
       Uri_path= Uri#url.path,
       Path= string:tokens(Uri_path, "/"),

      [_|Tail]= Path,
       case Tail of
              [] ->
                     %% If the URI is malformed
                     {status, 403}; %% HTTP code 403 means Forbidden
              [_|_] ->
                     %% start execution of the request in a new thread
                     erlang:spawn(processManager, processRequest, [A, EntryPid,
Uri, Method, Path]),
                           %% wait for reply
                           receive
                              Reply->
                                        %% send reply to the client
```

```
                                Reply
                        after 5000->
                                %% timeout of 5 seconds
                          {status, 408} %% HTTP 408 means request timeout
                        end
                end.
```

Figure 5-2 The module parsing the client requests and calling ERB

5.2 Process Manager

Once the request has been passed to the "processManager", it's responsible for further processing it by contacting other components of the ERB. A skeleton structure of processManager is shown in Figure 5-3.

```
-module(processsManager).
%%
%% Include files
-include("yaws_api.hrl").%%
%% Exported functions
-export([processRequest/5, search_resource/2]).
%%
%% Local Functions

processRequest(Arg, Pid, Uri, 'GET', Path)->
       read_resource(Arg, Pid, Uri, Path); %% Read resource
processRequest(Arg, Pid, Uri, 'POST', Path)->
       post_resource(Uri, Path, Arg); %% Create resource
processRequest(Arg, Pid, Uri, 'PUT', Path)->
       put_resource(Uri, Path, Arg); %% Update resource
processRequest(Arg, Pid, Uri, 'DELETE', Path)->
       delete_resource(Uri, Path, Arg); %% Delete resource
processRequest(Arg, Pid, Uri, _, Path)->
       {status, 501}. %% HTTP 501 means method not implemented

read_resource(Arg, Pid, Uri, [_,"nurse", Nuuid, "patient", Puuid]) ->
       %% handle read request
post_resource(Uri, ["hospitalWS",Role, Id], Arg)->
       %% handle post request
put_resource(Uri, ["hospitalWS",Role, Id], Arg)->
       %% handle put request
delete_resource(Uri, ["hospitalWS",Role, Id], Arg)->
       %% handle delete request
```

Figure 5-3 Process Manager Skeleton

The functions contained in the "processManager" module are responsible for exhibiting the following functionalities:

1. Controlling client access to various resources: The process manager after receiving the request contacts the client access controller to retrieve a list of allowed resources for a client as shown in Figure 5-4.

```
get_user_resources(Id)->
	{ok, Result}= clientAccessController:read_user(Id),
	ResourceList=case Result of
		none ->
			[];
		{_, _} ->
			Result
	end,
	ResourceList.
```

Figure 5-4 Getting list of allowed resource for a client

The resource that user requested is then checked against the list as shown in Figure 5-5. If the resource is found in the list then an "allowed" value is returned otherwise "not_allowed".

```
search_resource(RequestedRId, ResourceList)->
	case sets:is_element(RequestedRId,sets:from_list(ResourceList)) of
		true->
			allowed;
		false->
			not_allowed
	end.
```

Figure 5-5 Search for resource in list of allowed resources

If the search_resource returns "allowed", then the request is checked for the type of HTTP method allowed on that particular resource. The code in Figure 5-6 checks whether 'GET' method is allowed to be invoked on the requested resource with id

'RequestedRId'. If method is allowed then a "method_allowed" value is returned otherwise "method_not_allowed".

```
is_method_allowed(ResourceList, RequestedRId)->
	[Check|_]=[sets:is_element('GET', sets:from_list(L))||{Id, L}<-
ResourceList, Id=:= RequestedRId],
	case Check of
		true->
			method_allowed;
		false->
			method_not_allowed
	end.
```

Figure 5-6 Check if invoked HTTP Method is allowed

2. Routing request to appropriate resources: Once the process manager has established that the client requesting a particular resource is allowed to access that resource with the particular HTTP method, the request is spawned to the "resourceManager" to retrieve the resource representation shown in Figure 5-7.

```
read_resource(Arg, Pid,  Uri, [_,"nurse", Nuuid, "patient", Puuid]) ->
	{ResourcesList,_} = get_user_resource(Nuuid),
	Resources= [Val||{Val, _}<-ResourcesList],
	%% Check the HTTP allowed methods
	case search_resource(Puuid, ResourcesList) of
		allowed ->
		case Resources of
			[]->
				Pid!{status, 404};
			[_|_]->

				R= is_method_allowed(ResourcesList, Puuid),
				Case R of
					method_allowed->
				  %% Spawn the resource manager
						erlang:spawn(resourceManager,
get_resource, [Puuid, Pid]);
					method_not_allowed->
						Pid!{status, 405}
					end
			end;
		not_allowed->
			Pid!{status, 403}
	end;
```

Figure 5-7 Route the request to the Resource Manager

3. Handling HTTP requests other than GET: The other HTTP methods implemented by our architecture are POST, PUT and DELETE. The Process Manager identifies these requests and after parsing the body of the request it spawn it to the "resourceManager", which handles the creation, updating and deletion of resources. Figure 5-8 shows the implementation for handling POST request to create new resources

```
post_resource(Uri, ["hospitalWS",Role, Id], Arg)->
      case Role of
%% Add a new nurse
            "nurse" ->
                  {ok, AResources}= yaws_api:getvar(Arg, "aRlist"),
                  {ok, VResources}= yaws_api:getvar(Arg, "vRlist"),
                  erlang:spawn(clientAccessController, create_user,[Id,
Role, AResources, VResources]);
%% Add a new doctor
            "doctor"->
                  {ok, AResources}= yaws_api:getvar(Arg, "aRlist"),
                  {ok, VResources}= yaws_api:getvar(Arg, "vRlist"),
                  erlang:spawn(clientAccessController, create_user,[Id,
Role, AResources, VResources]);
%% Add a new patient
            "patient"->
                  {ok,Fname}= yaws_api:getvar(Arg, "first"),
                  {ok,Lname}= yaws_api:getvar(Arg, "last"),
                  {ok, Diagnosis}= yaws_api:getvar(Arg, "diagnosis"),
                  Data= [Fname, Lname, Diagnosis],
                  erlang:spawn(resourceManager,create_resource,[Id, Data]);
%% Add a new view resource
            "view"->
                  {ok, Resources}= yaws_api:getvar(Arg,"aRlist"),
                  erlang:spawn(resourceManager,create_vresource, [Id,
Resources])
      end;
```

Figure 5-8 Handling POST requests

5.3 Client Access Controller

The Client Access Controller is implemented as a *gen_server behavior* [57] in Erlang. Behaviors in Erlang provide templates to implement certain functionalities. The *gen_server behavior* in particular provides a template to implement a server which can respond to its clients

synchronously (by invoking call function) or asynchronously (by invoking cast function). CAC receives request from the processManager, after processing them it return the results back to the processManager. In order to increase performance all the calls to CAC are made asynchronously. There are mainly three callback functions implemented in CAC (Figure 5-9):

1. Create client.
2. Update client.
3. Read client.

```
%% create new user
create_user(Id, Role, Resources, ViewResources, Pwd) ->
       gen_server:cast({global, ?MODULE}, {create_user, Id, Role, Resources,
ViewResources, Pwd}).

%% read the user
read_user(Id) ->
       gen_server:cast({global, ?MODULE}, {read_user, Id}).

%% update the user
put_user(Id, Role, Resources, ViewResources) ->
       gen_server:cast({global, ?MODULE}, {put_user, Id, Role, Resources,
ViewResources}).
```

Figure 5-9 CAC Callbacks

CAC talks to the Client Access Control Database (CACD) to read/ update/ create new users of our Web service.

5.3.1 Client Access Control Database

This database is implemented as a DETS (Disk Erlang Term Storage) [56] table and consists of records with the following fields:

1. Client ID
2. Client role in the system
3. List of atomic resource
4. List of view resources
5. Password

Table 5-1 shows the structure of CACD database. The list of resources consists of tuples with resource id as their key and list of allowed HTTP methods as its value. Following this approach the methods allowed on a particular resource for a particular client can be updated without affecting other clients of the WS.

Table 5-1 CACD table structure

```
-record(client, {
                    uuid, %% Client Id

                    role, %% Client role as Nurse/ Doctor

                    aresources, %% List of Atomic resources

                    vresources, %% List of View resources

                    pwd
                    }).
E.g.
A nurse with id ‘1’ with access to atomic resource ‘1’ and ‘2’ and view
resource ‘51’ will be stored as:
{
client, '1', "nurse",
[{'1', ['GET', ‘POST’]}, {'2',['GET']}],  %% List of Atomic resources
[{'51',['GET']}], %% List of View resources
“madmuc” %% Password
}
```

5.4 Resource Manager

The Resource Manager is implemented as a *gen_server behavior* in Erlang. It receives requests from the processManager, and after processing them asynchronously it returns the results to the client. Resource Manager is responsible for handling the creation/ update/ deletion/ reading of resources of our Web Service. RM implements the following callback functions for both atomic and view resources (Figure 5-10):

1. Read resource
2. Create resource
3. Delete resource

4. Update resource

```
%%% Handle GET Requests %%%%

get_resource(Id, RPid) ->
       gen_server:cast({global, ?MODULE}, {get_resource, Id, RPid}).

get_vresource(Id, RPid) ->
       gen_server:cast({global, ?MODULE}, {get_vresource, Id, RPid}).

%%% Handle POST Requests %%%%

create_resource(Id , Data, RPid) ->
       gen_server:cast({global, ?MODULE}, {create_resource, Id, Data, RPid}).

create_vresource(Id, LResources, RPid) ->
       gen_server:cast({global, ?MODULE}, {create_vresource, Id, LResources,
RPid}).

%%% Handle PUT Request %%%%

put_resource(Id, Data, RPid) ->
       gen_server:cast({global, ?MODULE}, {put_resource, Id, Data, RPid}).

put_vresource(Id, Resources, RPid)->
       gen_server:cast({global, ?MODULE}, {put_vresource, Id, Resources,
RPid}).

%%% Handle DELETE Request %%%%

delete_resource(Id, RPid) ->
       gen_server:cast({global, ?MODULE}, {delete_resource, Id, RPid}).

delete_vresource(Id, RPid)->
       gen_server:cast({global, ?MODULE}, {delete_vresource, Id, Resources,
RPid}).
```

Figure 5-10 Resource Manager Callbacks

5.4.1 Resource Database

This database is implemented as a DETS table which consists of records with the following fields:

1. Resource Id
2. Resources list

3. A tuple to store module and function name to handle GET requests
4. A tuple to store module and function name to handle POST requests
5. A tuple to store module and function name to handle PUT requests
6. A tuple to store module and function name to handle DELETE requests
7. A list to store state.

Table 5-2 shows the structure of the Resource Table. The resource list in the resource database helps differentiate between an atomic and a view resource. In case of an atomic resource, the list contains only the id of the atomic resource, and in the latter case the list contains ids of all resources on which the view resource is defined.

Table 5-2 Resource table structure

```
-record(resource,{
                            id, %% List of View resources

                            resourcelist=[], %% List of  resources

                            get={},%% GET resource function

                            put={},%% PUT resource function

                            post={},%% POST resource function

                            delete={},%% DELETE resource function

                            state=[] %% Resource state
              }).

E.g.

1. An atomic resource with id '1' will be stored as:

{resource,
       id='1',
       resourcelist=['1'],
       get= {?MODULE, getFunction},
       put={?MODULE, putFunction},
       post={?MODULE, postFunction},
       delete={?MODULE, deleteFunction},
       state=[AtomicState]
}

2. A view resource with id '51' defined on resource '5', '6' is stored as:
{resource,
       id=Id,
       resourcelist=['5', '6'],
       get= {?MODULE, getViewFunction},
       put={?MODULE, putViewFunction},
       post={?MODULE, postViewFunction},
       delete={?MODULE, deleteViewFunction},
       state=[ViewState]
}
```

5.4.2 Data Store Database

This database is responsible for storing the core data related to our resources. In our web service of a hospital information system, this database will store the information regarding the patients. This database is also implemented as a DETS table and consists of records with the following fields:

1. Id
2. First name
3. Last name
4. Diagnosis

Table 5-3 shows the structure of the Data store table. Whenever an atomic resource is created, the above fields are required to be supplied by the administrator which results in creating an entry in our data store database. No entries are added when a view resource is created as the view resource is only a definition which is executed when someone requests it.

Table 5-3 Data store table structure

```
-record(data, {
                        id,
                        firstname,
                        lastname,
                        diagnosis
                        } ).

E.g.

When an atomic resource with id ‘1’ is created then the data store will add
the following entry:

{data,
       id=‘1’,
       firstname=“Sunny”,
       lastname=“Sharma”,
       diagnosis=“Headache”
}
```

5.5 Map Reduce

The map/ reduce technique is used when a view resource is generated. When the request for getting the view resource is received, the Resource Database is queried to retrieve the record associate with view resource id (Figure 5-11). The value of get field of the record is then extracted to find the module and the function name to handle GET request. The execution to handle mapping/ reducing is starting in a new process by spawning the Get function.

```
handle_cast({get_vresource, Id, RPid}, State)->
      case R = dets:lookup(?ResourceDB, Id) of
            []->
                  RPid![{status, 404}];
            [_]->
                  [{_Resource, _Id, LResources,
Get,_Put,_Post,_Delete,_Patch, L}|_] = R,
                  {Mod, Fun}= Get,
                  erlang:spawn(Mod, Fun, [Id, LResources, RPid])
      end,
            State1= #state{method="GET", table=?ResourceDB, rid=Id},
            {noreply, State1};
```

Figure 5-11 View Resource Callback

The spawned function then extracts all the resources from the list of resources and spawns new processes to do a map on each resource (Figure 5-12).

```
getViewFunction(Id, List, RPid)->
	ListTemp1=[dets:lookup(?ResourceDB, X)||X<-List],

	ListCombine=sets:to_list(sets:from_list(lists:flatten(ListTemp1))),
	Count= lists:flatlength(ListCombine),

	%% Call Map
	MapResults= getViewFunction(ListCombine, Count, 1,[]),
	%% Call Reduce
	erlang:spawn(?MODULE, reduceFun, [MapResults, RPid, 0]).

getViewFunction(List, Count, TimeCalled, Result)->
	if
		TimeCalled==1 ->
			%% Spawn new process for each resource
			[erlang:spawn(?MODULE, mapFun, [X, self()])||X<- List];
		true->
			ok
	end,

	receive
		""->
			getViewFunction(List, Count, TimeCalled+1,[[]|Result]);

		L->
			if
				TimeCalled==Count->
					[L|Result];
				true->
					getViewFunction(List, Count, TimeCalled+1,
[L|Result])
			end
	end.
```

Figure 5-12 Get View Function callback

Figure 5-13 shows a Map function which reads a resource from the dets table and returns its JSON representation.

```
%% Map Function
mapFun(X, ReturnPid)->
	case Results= dets:lookup(EDBStore, X) of
		[]->
			ReturnPid!"";
		[_]->
			Body= erlang:spawn(?MODULE, compile_json, [Results]),
			ReturnPid!Body
	end.
```

Figure 5-13 Map function

The results from the map are then fed into a reduce function. Here, the reduce function is responsible for combining the JSON representation of 'n' resource to return a single JSON document representing the view resource.

```
%% Reduce Function
reduceFun(MapResults, RPid, _)->
	Body=erlang:apply(?MODULE, reduceFun, [MapResults, ""]),
	ViewBody="{\"patients\":["++Body++"]}",
	RPid![{status, 200},{allheaders,[

		{header,["Content-Type: application/json; charset=utf-8"]},

			{header,["Pragma: no-cache"]},

			{header,["Cache-Control: no-cache"]},

			{header,["Connection: Close"]}]

			},{html, ViewBody}].

reduceFun([], Body)->Body;
reduceFun([H|T], Body)->
	case T of
		[]->
			reduceFun(T, Body++H);
		[_|_]->
			reduceFun(T, Body++H++",")
	end.
```

Figure 5-14 Reduce Function

5.6 Caching

In order to provide faster response time, caching techniques are employed by our middleware. The caching is achieved by performing lookup operation on ETS (Erlang Term Storage) [58] tables instead of DETS tables. ETS tables are stored in the RAM of the computer thereby providing faster lookup times.

To ensure data consistency, ets tables are forced to copy the data from its DETS counterpart whenever there is an HTTP request other than GET is received. Figure 5-15 shows the use of ETS tables when a POST request is received.

```
handle_call({create_vresource, Id, LResources}, _From, _State) ->
      %% Check whether the view already exists in the system
      Reply= case Results= ets:lookup(?EResourceDB, Id) of
            [] ->
                  RData= #resource{id=Id, resourcelist= LResources, get=
                        {?MODULE, getViewFunction1}, put={?MODULE,
                        putViewFunction}, post={?MODULE, postFunction},
                        delete={?MODULE, deleteFunction}, patch={?MODULE,
                        patchFunction},state=[LResources]},
                  dets:insert(?ResourceDB, RData),
                  io:format("~n View Resource: ~p created!~n", [RData]),
                  {status, 201};
            [_] ->
                  [{_Resource, Id, List, Get,Put, Post,Delete,Patch,
                  State}|_] = Results,
                  NewState= #resource{id=Id, resourcelist=List, get=Get,
                        put=Put, post=Post, delete=Delete, patch=Patch,
                        state=[LResources|State]},
                  {Mod, Fun}= Post,
                  erlang:spawn(Mod, Fun, [Id, NewState])
               end,
      dets:to_ets(?ResourceDB, ?EResourceDB),
      State1= #state{method="POST", table=?ResourceDB, rid=Id},
      {reply, Reply, State1};
```

Figure 5-15 Caching with ETS

5.7 Summary

In Chapter 5, the implementation details of the various components of the Enterprise Resource Bus are provided. The functionality of each component was also described in conjunction to other components of the ERB.

YAWS is used as an entry point for our WS. When a request arrives, the YAWS server parses it and forwards it to the middleware's Process Manager (PM) component. The PM then contacts the Client Access Controller (Erlang *gen_server behavior*) to retrieve a list of resources accessible to the requesting client. If the requested resource is found in the list of accessible resources the PM checks the HTTP method invoked by the client on the resources. The client request method is then compared to the allowed HTTP methods on the requested resource. If the method is allowed then the PM sends an asynchronous request to the Resource Manager (Erlang

gen_server behavior). The Resource Manager performs a Map/ Reduce to return the response to the client.

CHAPTER 6

EXPERIMENTS

The experimental setup to run the experiments (Figure 6-1) uses a load generator hosted on a desktop machine to emulate client requests. The middleware and the WS are hosted on separate machines which are connected with each other through a gigabit Ethernet cable.

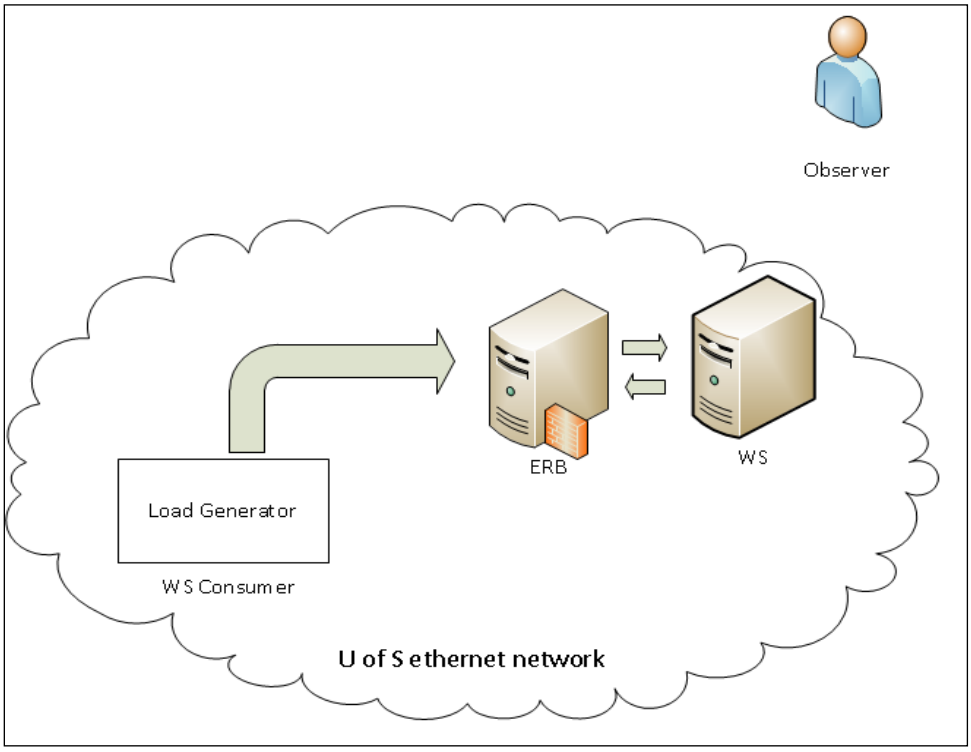

Figure 6-1 Experimental Setup

6.1 Experiment Goals

The experiments focused on evaluating the implemented architecture based on the following four goals:

Goal 1: How much overhead is introduced to the system due to use of ERB?

Goal 2: How scalable is the middleware architecture? This will test the ability of the architecture to support multiple clients, and the amount of time it take to provide a response.

Goal 3: How long does it take to generate and serve views to clients? This will test the performance of using the Map Reduce approach to generate resource.

Goal 4: How does the middleware respond to the requests which are not allowed for a client? This will test the middleware implementation for its responses to the activities which are not allowed for a client. Furthermore, the HTTP response codes will be tested for its proper use of HTTP protocols.

6.2 Experimental Setup

The setup of experiment will include hosting the ERB, the WS and the load generator on separate desktop machines connected to the University of Saskatchewan network through a gigabit Ethernet connection (Figure 6-1). The empirical data related to these tests was recorded by the load generator.

In order to test the system the loads were generated using Apache Jmeter [54] and Apache bench [53]. The load generators were setup on a computer with the following specifications:

Windows Edition:

Windows 7 Enterprise

Service Pack 1

System:

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

CPU Q9450@ 2.66 GHz

2.67 GHz (4 processors)

RAM: 4GB

System: 64-bit operating system

The ERB was hosted on a computer with the following specifications:

Windows Edition:

Windows 7 Enterprise

Service Pack 1

System:

Processor: Intel (R) Xeon (TM)

CPU E5410@ 3.20 GHz

3.20 GHz (2 processors)

RAM: 10GB

System: 64-bit operating system

The Web server was hosted on a computer with the following specifications:

Windows Edition:

Windows 7 Enterprise

Service Pack 1

System:

Processor: Intel (R) Xeon (TM)

CPU E5410@ 3.20 GHz

3.20 GHz (2 processors)

RAM: 5GB

System: 64-bit operating system

6.3 List of Experiments

Following is the list of experiments that were conducted to evaluate the middleware architecture based on each of the goal outlined in section 6.1:

6.3.1: Evaluation of overhead

The reason for conducting this experiment is to check the amount of latency introduced to the responses due to the middleware architecture. This test will help us evaluate how fast/ slow is the average response time to requests when routed through our middleware.

6.3.1.1 Experiment Setup

In order to test the system a two-phase approach was used:

1. First, the request was generated from the client and routed directly to the REST WS. The time from when the request was generated to the time a response was received was measured (Figure 6-2).

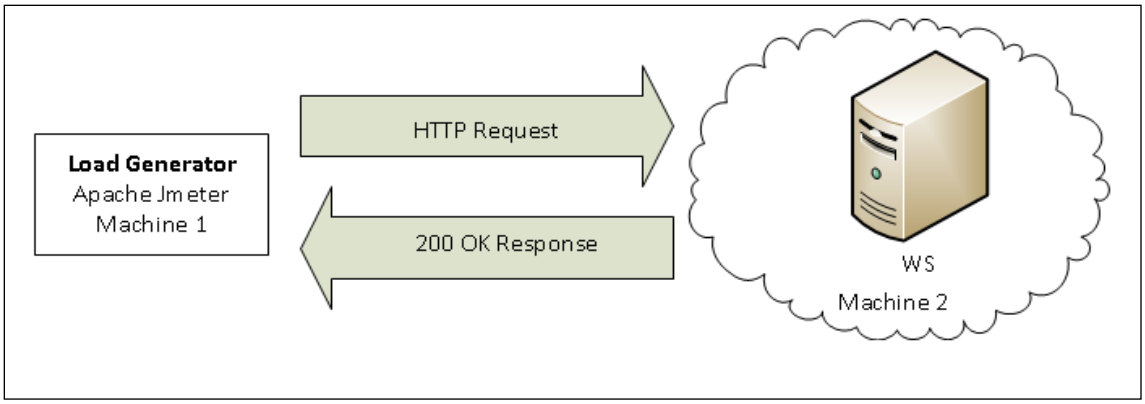

Figure 6-2 Client requesting resource directly from WS

2. Secondly, the request generated from the client was routed to the WS through the ERB. Again, the turn-around time for the request response was measured (Figure 6-3).

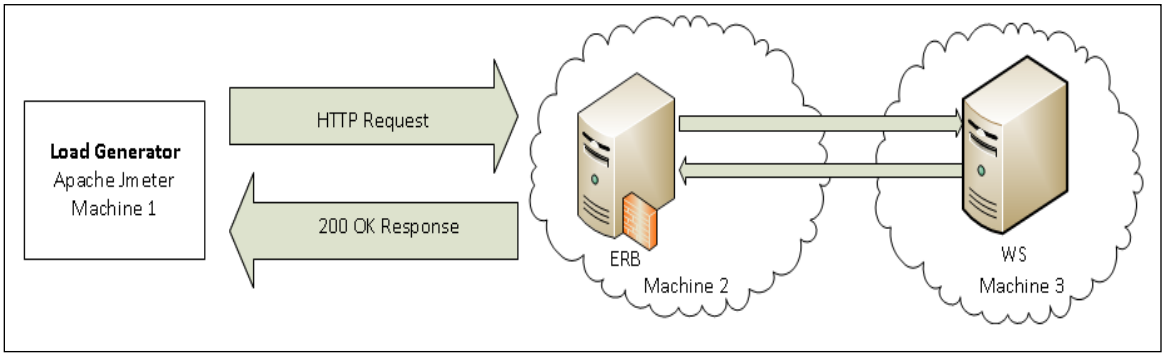

Figure 6-3 Client request routed through ERB

To conduct this experiment Apache Jmeter was used to simulate 50 concurrent clients which sent a total of 5000 read requests over a period of time. A factor of concurrency was introduced to simulate real-world conditions where multiple clients connect to the WS at the same time. This simulation was repeated 15 times to generate average response time over a total of 75000 requests. For each simulation average response time and throughput values were measured.

6.3.1.2: Results and discussion

Table 6-1 shows that the average response time taken to receive a resource directly from the web service is 208.8 ms, and to retrieve the same resource through the middleware is 218 ms.

Table 6-2 Average response time for 500 requests

Experimental Setup	**REST WS**	**REST WS with ERB**
Average response time for 5000 read requests (50 concurrent clients) in ms	208.8	218

Similarly, the throughput (number of requests/seconds) was measured for each setup. The REST WS with the middleware was able to serve on average 200.84 requests/ second as compared to an average of 208.46 requests/s by the REST WS (Figure 6-4).

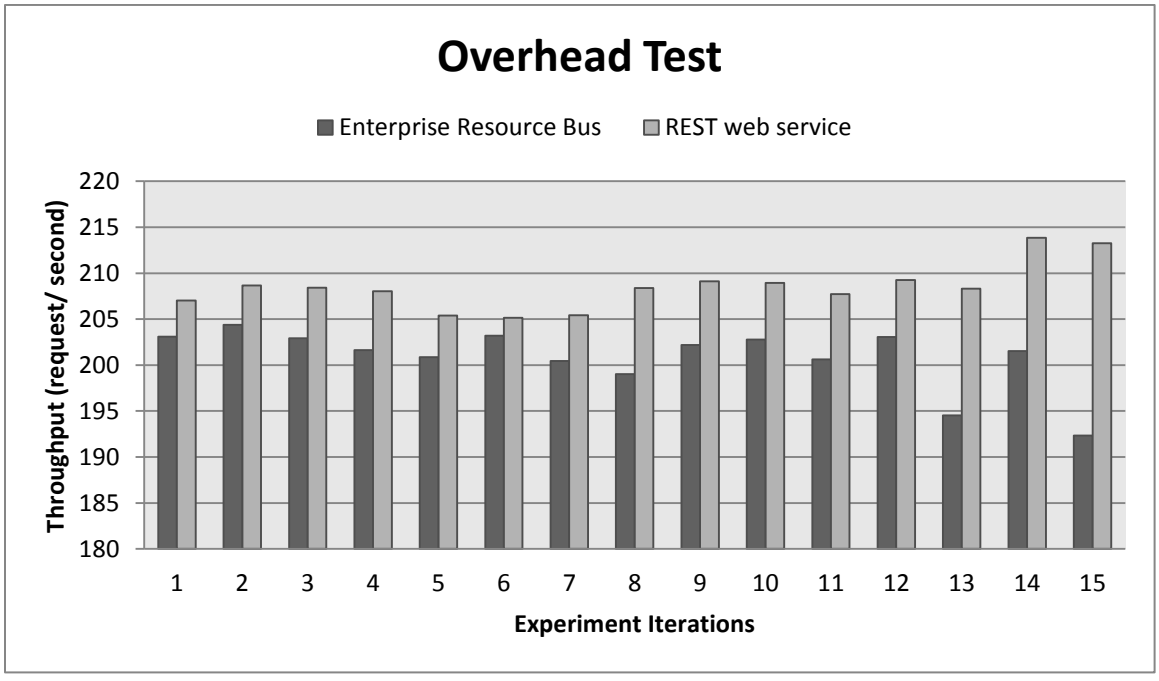

Figure 6-4 Throughput of middleware and REST WS

The results reveal that the average response time increases by approximately 10ms, i.e. by a factor of 4%. This was expected because the middleware execute additional code to authenticate the client before the resource is returned. Similar trend was observed when the throughput of the two setups was measured. The REST WS can process approximately 8 more requests/ second as compared to the REST WS using the ERB. The drop in throughput can again be attributed to the additional processing done by the middleware before returning a response.

6.3.2 Evaluation of Scalability

The reason for conducting this experiment is to check the performance of the middleware when the number of request and number of user increases. The test was conducted by simulating clients request using the Apache Bench load generator.

6.3.2.1 Experiment Setup

To evaluate the scalability of the middleware architecture setup shown in Figure 6-5 was used. Apache Bench was used as a load generator to send concurrent HTTP requests to the middleware. The load generating tool was installed on a machine in the laboratory with the specification given in Section 6.2.

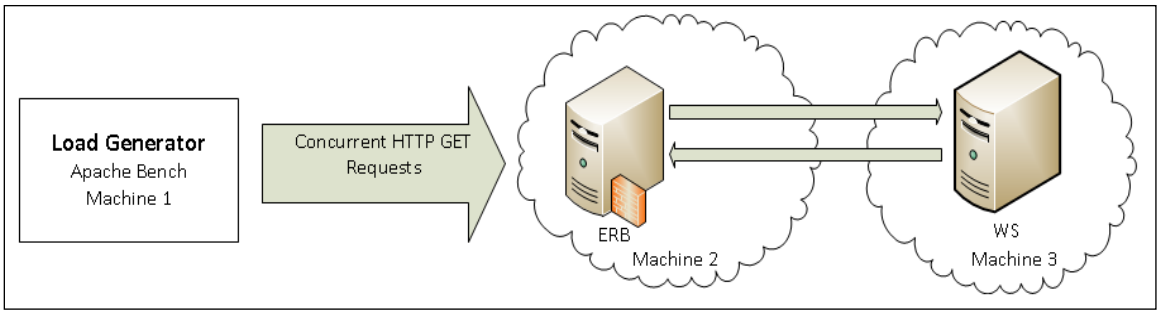

Figure 6-5 N-clients connecting to ERB

The load generator is configured to simulate the activities of 500 concurrent users of the WS. The number of concurrent requests that a user can send is varied from 500 to 10000. The

RESTful resources are represented as patient information records, illustrated as JSON data in Figure 6-6. The experiment is repeated for both the REST WS and the REST WS with the middleware.

```
{
“id” : “1”,
“firstname”: “Sunny”,
“lastname”: “Sharma”,
“diagnosis”: “headache”
}
```

Figure 6-6 A atomic web resource

6.3.2.2: Results and discussion

The result of the performance of the REST WS and the middleware are presented in Table 6-2. The minimum response time in both the setup is found to be approximately similar, i.e. 60ms. The mean response time for the middleware was reported to be 0.80 ms faster than the REST Web Service alone. This result suggests that even though the maximum response time to read a resource in a WS using the middleware is higher, over an average both the setups performs almost identically.

Table 6-3 Scalability test results (time per request)

Server Setup	**Minimum Time per request (ms)**	**Maximum Time per request (ms)**	**Mean response time (ms)**
REST WS	60.687	108.701	66.7929
REST WS with ERB	60.495	120.739	66.1950

The results from the experiement are graphically presented in Figure 6-7. The graph suggests a trend of reduced mean response time as the number of requests per user of the WS increases.

Both the setups start with their maximum response times, but gradually the response time decrease as the number of total requests are varied from 500 to 10000.

It was observed that when the number of request per user reaches 4500, both setups behave exactly the same and have identical response times. After this point, the mean response time tends to stabilise at 60ms even though the client requests are increased to 10000. This suggests a lower bound of our implementation was reached. A mean response time of 60ms with 500 concurrent clients sending 10000 requests suggests that our middleware is highly efficient in handling peak loads.

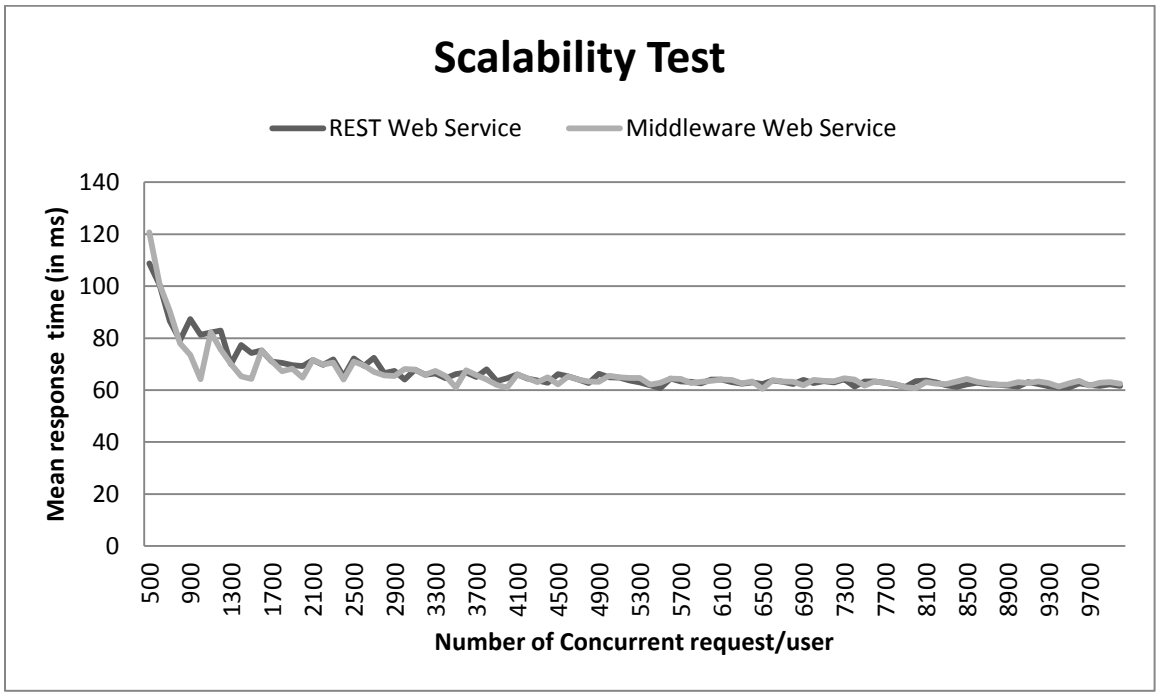

Figure 6-7 Mean response time per request (across all concurrent requests)

6.3.3 Evaluation of view response time

This test was conducted to test the performance of Map/ Reduce approach implemented by the middleware. To measure the performance the time required to generate a view and serve it back to the client was measure. The test conducted the following experiments:

Experiment 1: Time to generate a view for the first time from atomic resources

Experiment 2: Time to retrieve a view which is pre-cached

Experiment 3: Time to generate a view from another view and /or atomic resources

6.3.3.1 Experiment Setup

To conduct the experiments Apache Jmeter was used to simulate client requests. The load generator was configured to simulate 50 concurrent users sending a total of 5000 requests. This setup was repeated 15 times to measure the performance of views over a total of 75000requests.

In order to conduct experiment 1, a client GET request was generated requesting a view resource defined on only atomic resource. The time since the client has sent the request to the time a response was received was measured (Figure 6-8).

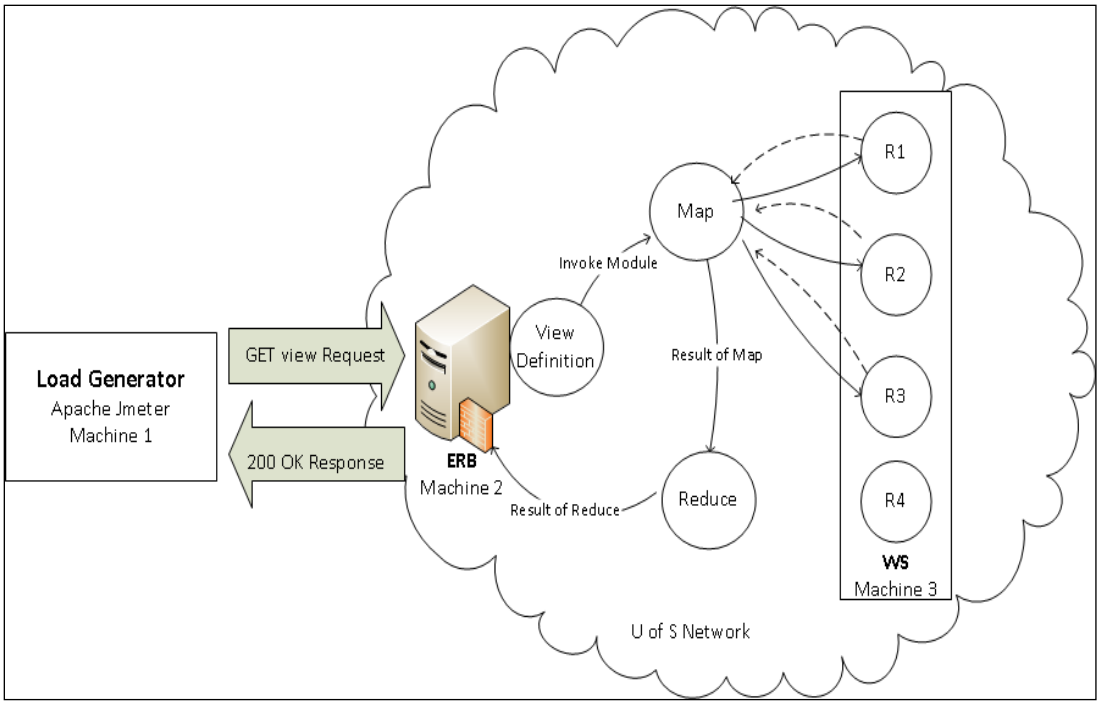

Figure 6-8 GET View resource based on atomic resources

To measure the performance of the Map/ Reduce approach same experiment was conducted for 5 view resource each consisting of a different number of atomic resources. Table 6-3 provides details about the view resource used in the experiment.

View	**No of resources**	**Size (bytes)**
View 1	10	1120
View 2	20	2095
View 3	30	3061
View 4	40	4035
View 5	50	5001

In order to conduct experiment 2, a client GET request was generated to a view resource which is known to be pre-cached. The time for the ERB to find the view in its cache and sending it back to the client was measured (Figure 6-9). This experiment will use the 5 view resources used in experiment 1 to compare the response times.

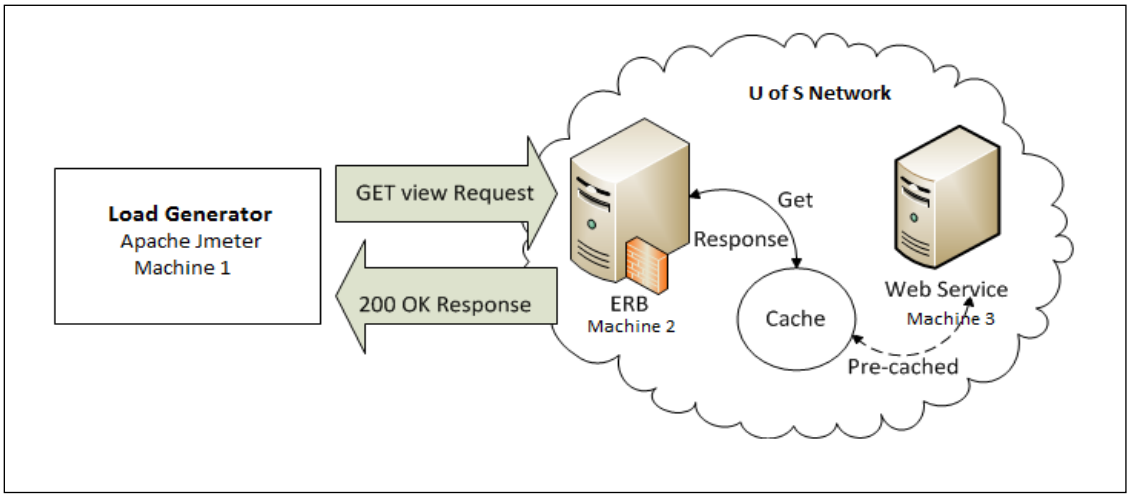

Figure 6-9 GET view resource from cache

In order to conduct experiment 3, a client GET request was generated requesting a view resource which is made up of another view resource and/ or atomic resources (Figure 6-10). The time since the request was received to the time a response was received is measured. The performance of this view resource will then be compared to the performance of a view resource made up of equal number of atomic resources.

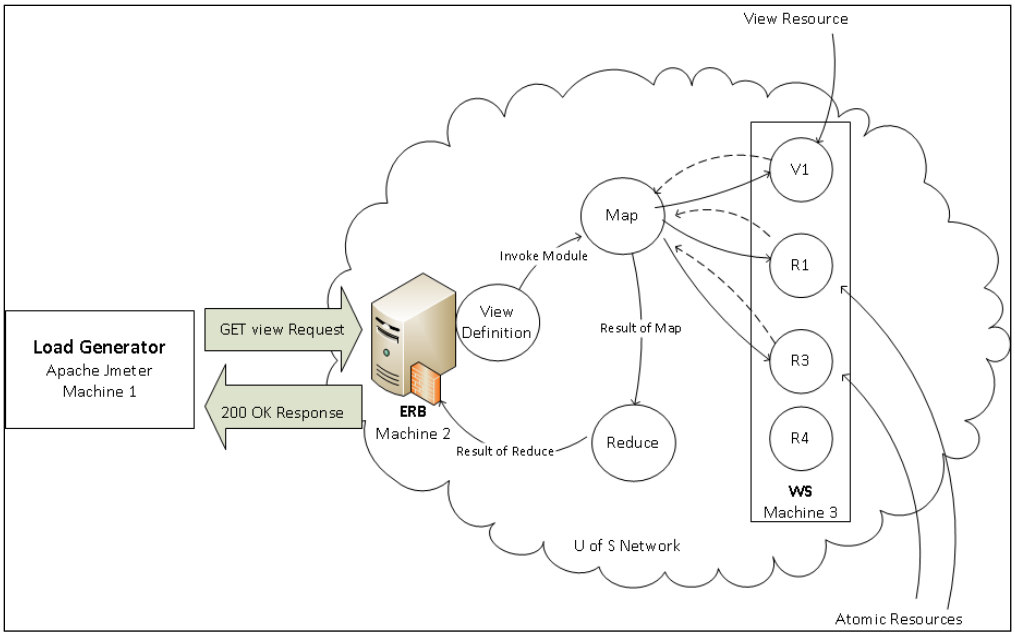

Figure 6-10 GET view resource based on another view resource

6.3.3.2 Results and discussion

The results from experiment 1 reporting the average response time are presented in Table 6-4. A gradual increase in the average response time can be seen with the increase in number of resource. This increase can be attributed to the extra time required to map and reduce the growing number of resources from 10 to 50.

Table 6-4 Average view response times (no-cache)

View	**Average response time in ms [Experiment 1]**	**Average response time in ms [Experiment 2]**
View 1	181.80	141.80
View 2	214.06	155.42
View 3	265.01	162.53
View 4	309.66	172.6
View 5	350.20	172.2

The results from experiment 2 are also presented in Table 6-4. An increase in the average response time can be seen as the number of atomic resource in a view increases. However, the results from experiment 2 suggest that serving a pre-cached view resource to the client is much faster e.g. in case of view resource 5 the time to serve the view from the cache is 178ms less than serving the same resource when it's uncached.

The results from experiment 1 and experiment 2 are graphically represented in Figure 6-11 to Figure 6-15. The graphs shows the average response time for running the map reduce from view 1 to 5.

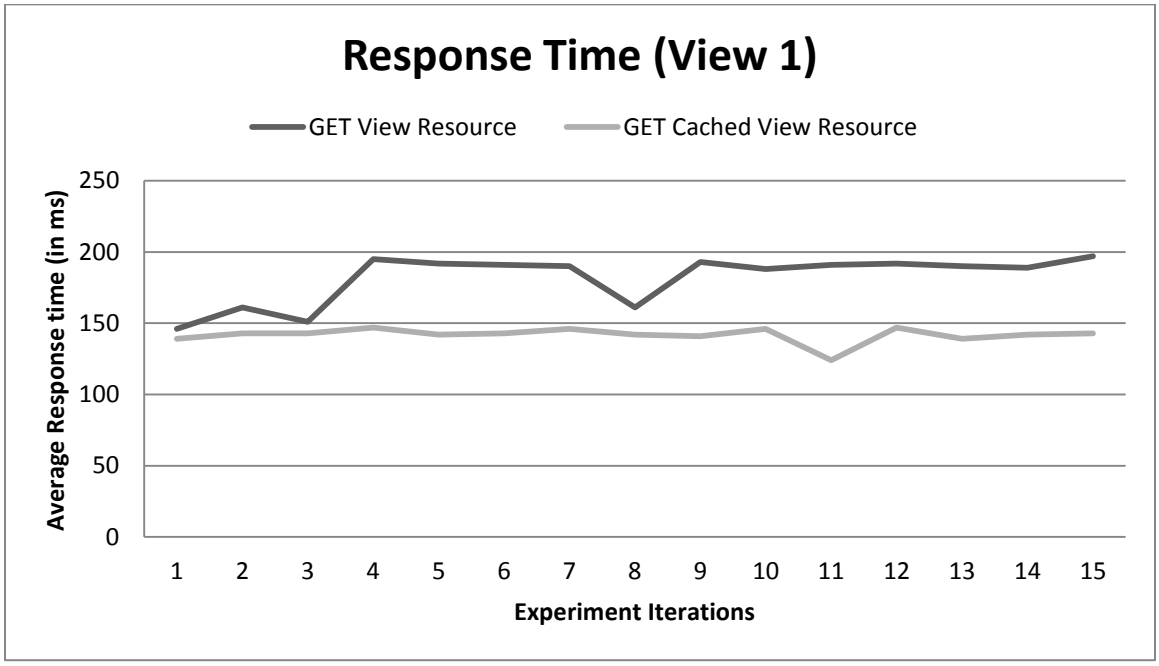

Figure 6-11 Average response time for View 1

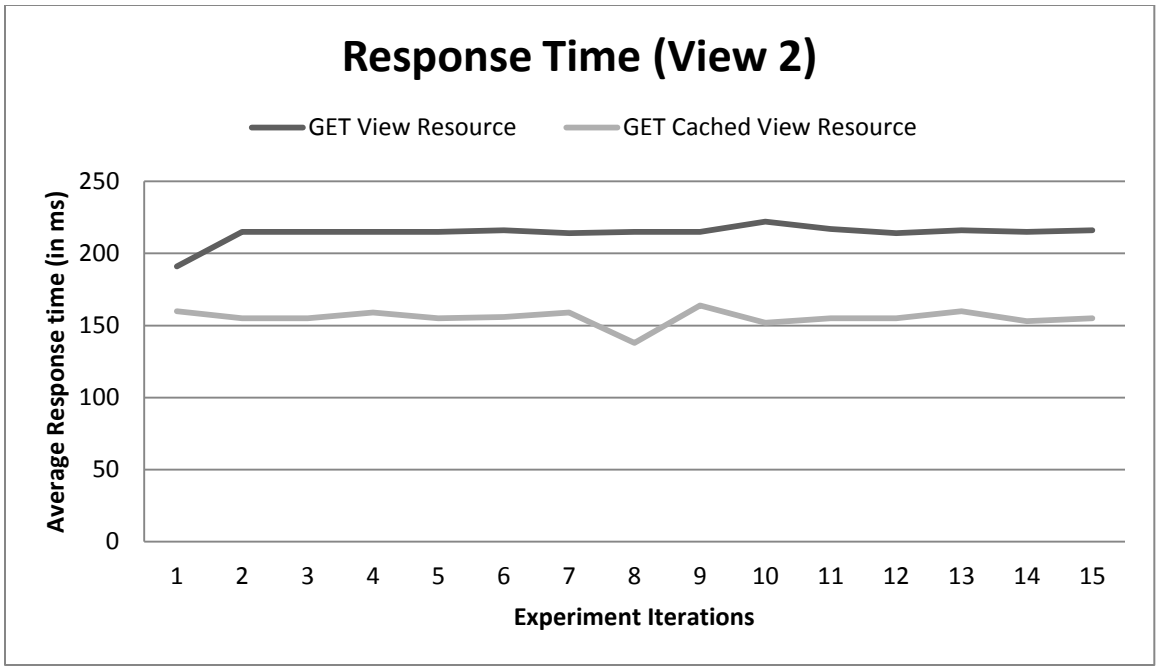

Figure 6-12 Average response time for View 2

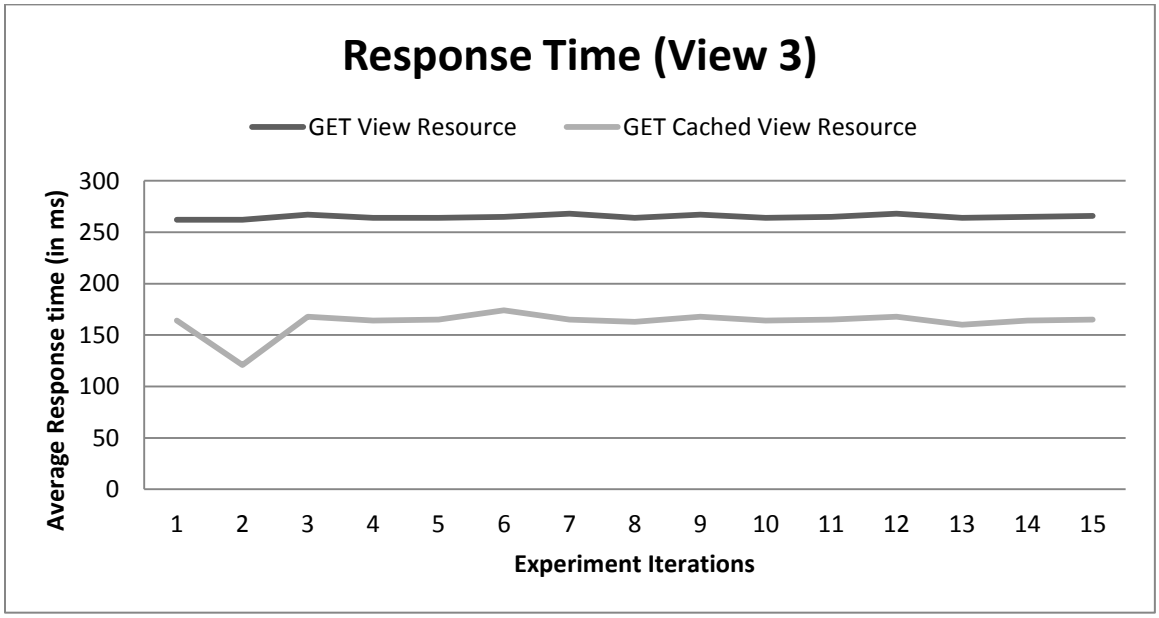

Figure 6-13 Average response time for View 3

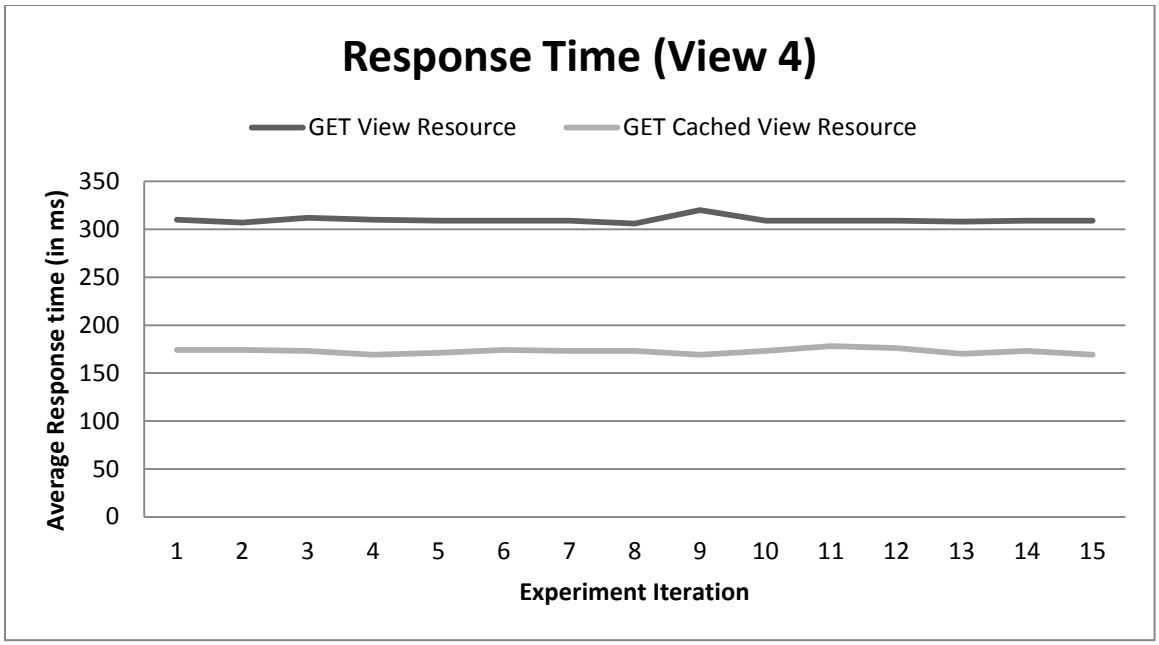

Figure 6-14 Average response time for View 4

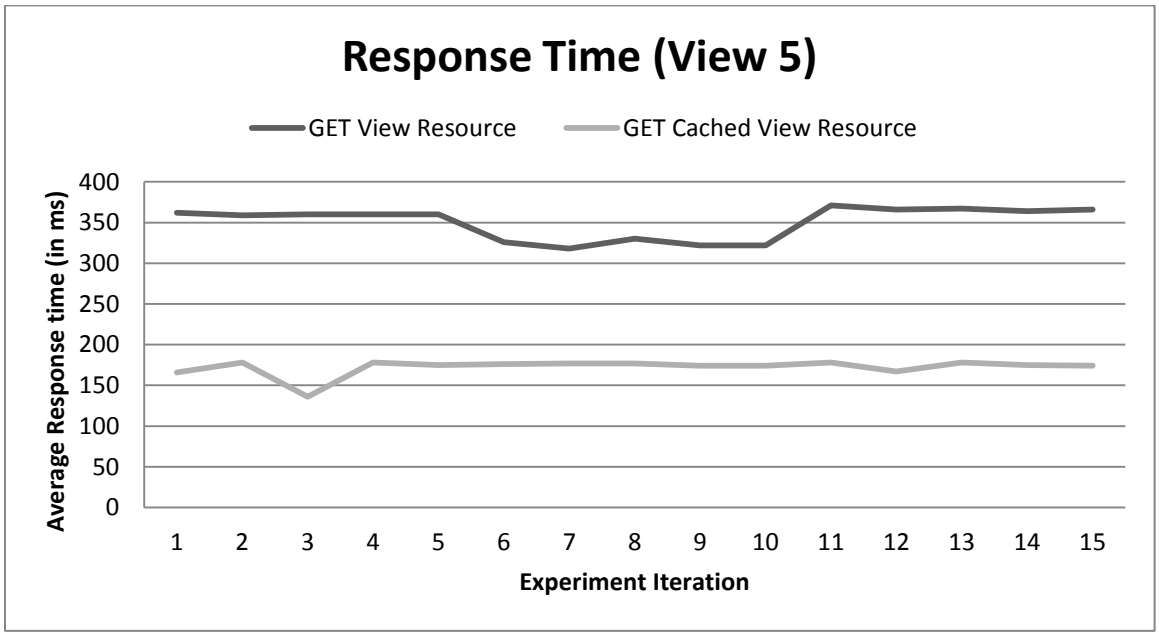

Figure 6-15 Average response time for View 5

The results from experiment 3 are presented in Table 6-5. The experiment records the average response time and the throughput of the middleware. An increase of 9.6% (33.46ms) in response time was observed when a view resource is defined on another view resource.

Throughput was measured to identify the overhead introduced by the Map/ reduce approach when serving view resources. The throughput of the middleware decreases by 4.6% (6.19 request/ sec) when the requested view resource is based on another view resource.

Table 6-5 Performance results for view defined on another view

View	**Average response time in ms**	**Throughput (requests/ second)**
View defined on atomic resources only	350.20	130.34
View defined on view and atomic resource	383.66	124.15

The results from 3 are graphically presented in Figure 6-16. As expected, the graph shows lower average response time for view resource based on atomic resources as compared to view resource based on other view resources

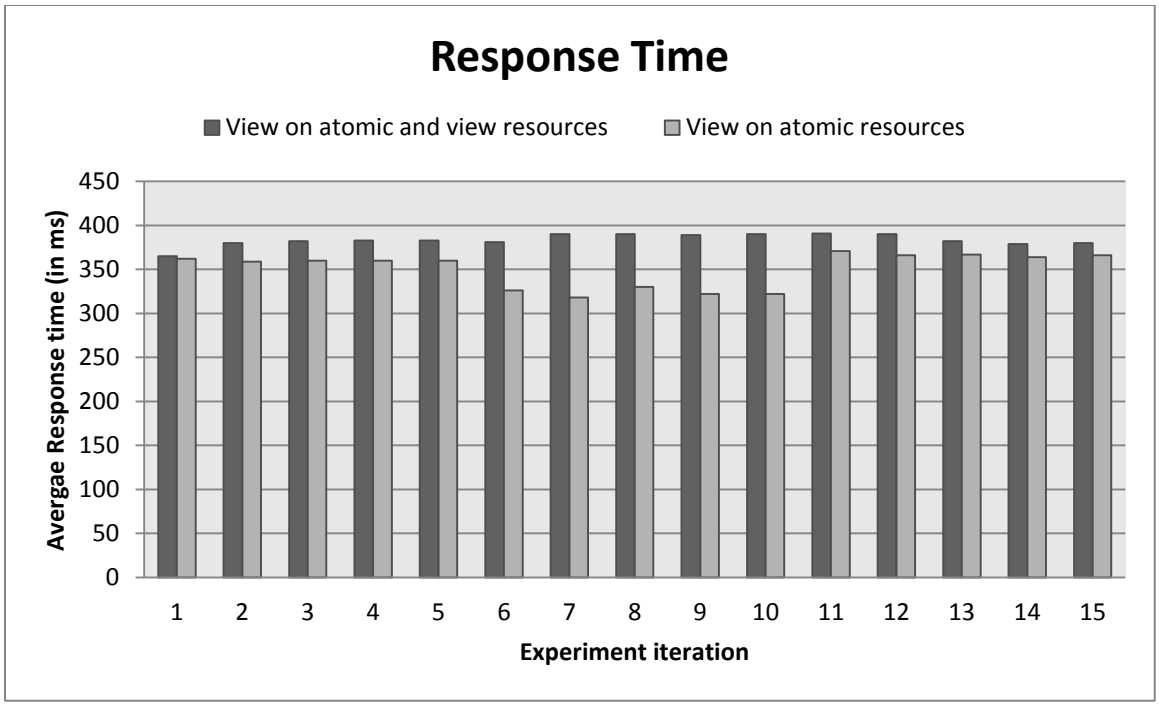

Figure 6-16 Average response time for view resource based on another view

6.3.4 Evaluation of middleware security

This test was conducted to evaluate the middleware's level of security to activities which should not be allowed for certain specific clients. Normally, in a WS a client can access a resource by just knowing its URI. Also, the client can invoke any HTTP methods on a resource e.g. a client can change the state of a resource by invoking a POST request where it should only be sending a GET request. This can cause the resource in the WS to be changed deliberately or inadvertently. These changes can affect other clients of the WS rendering the WS less secure. To test these scenarios when our middleware is employed by the WS the following test cases are considered:

Case1: Client requesting access to a non-allowed resource

This happens when we want the client to have access to only certain resources in our system.

GET /hospitalWS/nurse/1/patient/11 HTTP/1.1

Host: yuting.usask.ca:9091

Connection: keep-alive

Cache-Control: no-cache

Pragma: no-cache

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.64 Safari/537.11

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Figure 6-17 Body of GET request to a non-allowed resource

Let's consider the case of a nurse client with id 1 which is allowed access to patient resources with id 1 to 10. So, whenever a request for patient resource with id 11is sent (Figure 6-17) the middleware respond with the HTTP status response code 403 meaning that the resource is forbidden to be accessed (Figure 6-18).

HTTP/1.1 403 Forbidden

Connection: Close

Date: Thu, 10 Nov 2012 20:48:43 GMT

Server: Yaws 1.92

Figure 6-18 Server response to HTTP GET request for non-allowed resource

Case2: Client invoking an HTTP method not allowed on a resource

This situation is desirable when we want the client to have only certain type of access to the resources.

POST /hospitalWS/nurse/1/patient/1 HTTP/1.1

Host: yuting.usask.ca:9091

Connection: keep-alive

Cache-Control: no-cache

Pragma: no-cache

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.64 Safari/537.11

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Figure 6-19 Body of POST request to a non-allowed method resource

Let's assume that our nurse client with id 1 is allowed to have only read access to patient resource with id 1 to 10. So, whenever an HTTP method other than GET is invoked by the nurse to patient resource with id 1 (Figure 6-19) the middleware respond with HTTP status response code 405 meaning that the method is not allowed to be invoked on patient resource with id 1 (Figure 6-20).

HTTP/1.1 405 Method Not Allowed

Connection: Close

Date: Thu, 12 Nov 2012 21:48:43 GMT

Server: Yaws 1.92

Figure 6-20 Server response to POST request for no-allowed method resource

6.3 Summary

This chapter described the various experiments that were conducted in order to evaluate the ERB middleware architecture against factors such as overhead, scalability, and response/ wait times and security. The summary of the results are listed below based on the goals

Goal 1: How much overhead is introduced to the system due to the use of ERB?

Experiment 1: To calculate the system overhead the load generator was configured to simulate 50 concurrent clients to send a total of 5000 requests. This test was repeated 15 times to calculate the average over a total of 75000 requests. The tests were repeated for the WS running with and without the middleware. The results reported the average response time and throughput for both the setups. The results in Table 6-1 shows an increase in average response time by 10ms (4%) for the WS running with the middleware. The increase was expected as the middleware required additional computation to evaluate client access control. Similar trend was observed when the throughput of the two setups was measured (Figure 6-4). The REST WS using the middleware reported a decreased throughput of 8 requests/second.

Goal 2: How scalable is the middleware architecture?

Experiment 1: To calculate the scalability of the middleware architecture the load generator was configured to simulate 500 concurrent clients with an increasing number of total requests from 500 to 10000 for each client. The results presented in Table 6-2 showed that the mean response time for both the WS using the middleware and the WS not using the middleware were identical (~66ms). The results presented in Figure 6-7 showed a trend of reducing mean response times when the number of total requests for each client is increased. Moreover, it was found that when the number of request per user

reaches 4500, both setups have identical response times. Also, increasing the clients requests further upto 10000 reported a steady average of 60ms.

Goal 3: How long does it take to generate and serve views to the clients?

The load generator was configured to simulate 50 concurrent clients to send a total of 5000 requests. The test was repeated 15 times to calculate the average over a total of 75000 requests for each of the following experiments:

Experiment 1: This experiment focused on calculating the response time for view resource defined on only atomic resources. To measure the performance of Map reduce the experiment was repeated for five different view resources with each view consisting of varying number of resource (Table 6-3). It was observed that the average response time to get a view with 10 atomic resources is 181.80ms. An increased average response time was reported as the number of atomic resources was increased from 10 to 50 (350.20ms).

Experiment 2: This experiment focused on calculating the response time for view resources when they are served from the middleware cache. The experiment was repeated for the same five view resources used in experiment 1. Figure 6-11 to Figure 6-15 presented the graphs for each view resource showing a decreased average response time for serving all the view resource. The caching approach found an improved average response time of 178ms less than the time observed in experiment 1.

Experiment 3: This experiment focused on calculating the response time for a view resource composed of other view resources. The average response time was compared to an average response of a view resource composed of equal number of atomic resources.

The increase of 33.46ms (9.6%) average response time was observed for the getting a view based on another view (Table 6-5).

Goal 4: How the middleware respond to the requests which are not allowed for a client?

To test the middleware ability to keep its resources secure from being changed or accessed by unauthorized users, the following two test cases were considered:

Case 1: The middleware was tested for its response to client's request to a resource which shouldn't be accessible to this client. It was observed that the middleware replied with a HTTP response code of 403, meaning the resource is forbidden.

Case 2: The middleware was tested for its response to client's request using when a non-allowed HTTP method is invoked on a resource. It was observed that the middleware replied with a HTTP response code of 405, meaning that the method is not allowed to be invoked.

CHAPTER 7

SUMMARY AND CONTRIBUTION

This work introduces a novel middleware architecture (ERB) for Resource Oriented Systems to tackle the issues of security, access control and resource management. As REST is gaining more popularity over SOA in recent years, the numbers of consumers as well as the number of resources for REST systems are growing exponentially. This unchecked growth is soon becoming a nuisance for administrator's who wants to control who can see what resources; often leading to frustrating changes to complete systems overhaul. The middleware architecture put forward in this research is focused on remedying these issues by abstracting the logic of access control and security to the concept of view resources to be managed in the middleware. This research draws inspiration from the middleware architecture implemented in SOA called Enterprise Service Bus. It has been shown over time that the use of middleware in SOA systems helped in creating scalable, secure and manageable systems. Since, the implementation of ERB is inspired from ESB we have emulated some functionalities of ESB and translated them to resource oriented architecture. In addition, this work for the first time in the related literature uses the concept of view on RESTful resources inspired from the concept of virtual relations in databases.

This research proposes the middleware architecture to consist of 5 components, namely: Process Manager (acts as an entry point and contact for other components in the ERB), Resource Manager (define and manage resource in our WS), Cache Manager (to provide timely response to client requests), Client Access Controller (checks whether a client can access a particular resource) and Client Access Control Database (stores the data related to the client and the resources accessible to them). The view resources are defined on a set of resources as stored procedures. These stored procedures use Google's "MapReduce" technique to generate a single

resource which can be returned in response to the client's request. To control access to individual resources, each client of the WS has a record stored in the Client Access Control Database. Each record representing one client contains information about the allowed atomic and view resources with allowed HTTP methods for each resource. This technique warrants flexibility as for different client's different resources can be allowed and for each resource different HTTP methods can be allowed. The process manager insures the security of WS resources by using the information from the Client Access Controller to allow/ reject a client's request.

The implementation of the architecture is done entirely done in Erlang. The middleware components like Client Access Controller and Resource Manger are implemented as Erlang's *"gen_server"* behaviors, Client Access Control Database and Cache Manager uses Erlang databases called DETS and ETS respectively. Finally, the process manager resides in a YAWS web server which is also written in Erlang. All the communication within the components of ERB takes place asynchronously through message passing rendering it highly scalable and non-blocking.

The contributions and findings of our work are summarized below:

- A novel middleware architecture for Resource Oriented Architecture is developed.
- The concept of views from relational databases can be applied to resources in REST
- Views on resource can help control client access to various resources in a REST system
- The security to prevent the inadvertent/ deliberate change to the resource can be achieved by using views in conjunction with a middleware like ERB.

Though the goal of achieving access control, security and manageability is reached there are some limitations with our work. In our implementation, a centralized middleware architecture is being used. This can result in long service downtimes if the middleware fails because this is the only entry point to our REST WS.

Furthermore, our implementation puts the responsibility of describing view resources on the administrators who will need a proper knowledge of the all the atomic resources before they can start describing the view resources. With the current implementation the interface to write view resources is entirely programmatic which can be challenging for some as it entails a learning curve in the beginning. These limitations will be considered in our future work.

CHAPTER 8

FUTURE WORKS

8.1 Distributed Middleware

This work uses centralized middleware architecture as a single point of contact for clients with the Web Service. This is desirable in the beginning when the number of clients of WS is relatively small. However, as the number of clients in our system grows the proposed middleware will become overloaded and result in crash of service. In that case, it would be impractical to have a single middleware answering to all client requests. Therefore, to remedy this issue our future work will focus on investigating how multiple middleware's can be set up and connected together to serve the increasing number of clients. The work will also look into techniques to achieve eventual data consistency among all connected middleware's.

8.2 GUI based resource Management

This work requires the administrators to shoulder the responsibility to define view resources. This task can be challenging for administrator's who have a very small prior knowledge of all the resources in the REST. Moreover, the resources in our REST WS are stored in Erlang "dets" tables which have no means to be viewed graphically. The administrators have to write separate code to access the data contained in these tables. Therefore, our future work will investigate the use of a GUI interface to accommodate defining of the views. To achieve this, a web application will be built where administrators can select resource and write Map Reduce function to speed up the process of view generation.

8.3 Providing client customizable views

This work described the concept of using views as a mean to managing resources, security and access control. The concept developed in this research can be further extended to allow each client to tailor their service needs by defining their own views. This way each client can have multiple views of their resources. For example, a doctor accessing a hospital information system while on rounds to see his patient may want to change his view resource to show only patient vitals information on his tablet as compared to the detailed information that he sees when he is sitting in his office on his desktop. Therefore, the future work will investigate the use of currently developed view defining technique to allow individual clients to define their own views.

8.4 Enabling Publish/ Subscribe

This work requires the client to issue a request for the resource each time an updated copy of that resource becomes available. This can be a performance bottleneck due to the increased number of requests as the number of clients and the resources of the Web Service increases. Therefore, the future work will investigate the use of various publish/ subscribe mechanisms [59] to minimize the number of requests for each client, and pushing the updates to the client who subscribe for update events on a particular resource.

LIST OF REFERENCES

1. Rodriguez A. (2008). RESTful Web services: The basics, IBM developerWorks, Last accessed: April 2011. http://www.ibm.com/developerworks/webservices/library/ws-restful/

2. AlShahwan, F., & Moessner, K. (2010, May). Providing SOAP Web services and RESTful Web services from mobile hosts. In Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on (pp. 174-179). IEEE.

3. Riva, C., & Laitkorpi, M. (2009). Designing web-based mobile services with REST. In Service-Oriented Computing-ICSOC 2007 Workshops (pp. 439-450). Springer Berlin/Heidelberg.

4. Date, C.J. An introduction to database systems. Toronto: Addison-Wesley, 2000. Print.

5. Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). Restful web services vs. big'web services: making the right architectural decision. In Proceeding of the 17th international conference on World Wide Web (pp. 805-814). ACM.

6. Christensen, J. H. (2009, October). Using RESTful web-services and cloud computing to create next generation mobile applications. In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (pp. 627-634). ACM.

7. Knorr, E., & Gruman, G. (2010). Cloud Computing: What cloud computing really means? , Infoworld. Last accessed: April 2011. http://www.cloudcomputingdefined.com/

8. Hérault, C., Thomas, G., & Lalanda, P. (2005, December). Mediation and enterprise service bus: A position paper. In Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005) (pp. 67-80).

9. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

10. Brewer, E. A. (2000, July). Towards robust distributed systems. In Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (Vol. 19, pp. 7-10).

11. ESB: Glossary, Loosely Coupled. Last accessed: April 2011. http://www.looselycoupled.com/glossary/ESB

12. Exforsys Inc. (2007). SOA Disadvantages, Exforsys Inc. Last accessed: April 2011. http://www.exforsys.com/tutorials/soa/soa-disadvantages.html

13. Menge, F. (2007, August). Enterprise service bus. In Free and open source software conference.

14. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. Internet Computing, IEEE, 6(2), 86-93.

15. Wicks, G., Aerschot, E.V., Badreddin, O., Kubein, K., Lo, K., & Steele, D. Powering SOA Solutions with. New York: IBM Redbooks Pub. March 30, 2009. Print.

16. Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51-59.

17. Overdick, H. (2007, July). The resource-oriented architecture. 2007 IEEE Congress on Services. (pp. 340-347). IEEE.

18. Hamad, H., Saad, M., & Abed, R. (2010). Performance evaluation of restful web services for mobile devices. International Arab Journal of e-Technology, 1(3), 72-78.

19. Network Working Group. (1999). RFC 2616: Hypertext Transfer Protocol--HTTP/1.1. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T.

20. Lennon, J. Beginning CouchDB. Toronto: Apress publications. 2009. Print.

21. Evdemon, J. (2005). Principles of service design: Service patterns and anti-patterns. MSDN Library. Last accessed: April 2011. http://msdn.microsoft.com/en-us/library/ms954638.aspx

22. Lämmel, R. (2008). Google's MapReduce programming model—Revisited. Science of Computer Programming, 70(1), 1-30.

23. Qian, L., Luo, Z., Du, Y., & Guo, L. (2009). Cloud computing: An overview. Cloud Computing, 626-631.

24. Richards, M. (2006, October). The Role of Enterprise Service Bus, InfoQ. Last accessed: June 2011. http://www.infoq.com/presentations/Enterprise-Service-Bus

25. Mulesoft. What is Mule ESB?, MuleSoft Community. Last accessed: April 2011. http://www.mulesoft.org/what-mule-esb

26. Muzaffar, N. Hadoop, Nadir Muzaffar. Last accessed: April 2011. http://www.cs.berkeley.edu/~ballard/cs267.sp11/hw0/results/htmls/Muzaffar.html

27. Adamczyk, P., Smith, P. H., Johnson, R. E., & Hafiz, M. (2011). REST and Web Services: In Theory and in Practice. Wilde and Pautasso, 35-57.

28. Williams, P. (2007, March). Hypermedia as the engine of Application state, Peter Williams. Last accessed: April, 2011. http://barelyenough.org/blog/2007/05/hypermedia-as-the-engine-of-application-state/

29. Portio Research. Portio Research Mobile Factbook 2009, www.portiaresearch.com: Portia Research Limited. 2009, Print.

30. Wang, Q. A. (2011). Mobile cloud computing (Master dissertation, University of Saskatchewan).

31. Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California).

32. Schulte, W. R., & Natis, Y. V. (1996, April). Service Oriented Architectures Part 1, Gartner.

33. Royans. (2010, February). Brewers CAP Theorem on distributed systems, Scalable web architectures. Last accessed: April 2011. http://www.royans.net/arch/brewers-cap-theorem-on-distributed-systems/

34. Yates, S. (2007). It's Time To Focus On Emerging Markets For Future Growth, Forester.

35. Schmidt, M. T., Hutchison, B., Lambros, P., & Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. IBM Systems Journal, 44(4), 781-797.

36. Shankland, S. (2008, March). Google spotlights data center inner workings, CNET News Blog. Last accessed: April 2011. http://news.cnet.com/8301-10784_3-9955184-7.html

37. Tilkov, S. (2007, December). A Brief Introduction to REST. Community Architecture, InfoQ. Last accessed: April 2011. http://www.infoq.com/articles/rest-introduction

38. Erl, T. SOA Principles of Service Design. Toronto: Prentice Hall. 2007. Print.

39. Connolly, T. M., & Begg, C. E. Database systems: a practical approach to design, implementation, and management. Toronto: Addison Wesely. 1998. Print.

40. Haas Hugo. Designing the architecture for Web services, W3C. Last accessed: April 2011. http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html

41. Haas, H., & Brown, A. (2004). Web services glossary. W3C Working Group Note (11 February 2004).

42. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., & Orchard, D. (2004). Web services architecture: Service Oriented Architecture. http://www.w3.org/TR/ws-arch/#service_oriented_architecture

43. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., & Orchard, D. (2004). Web services architecture: What is a Web Service?. http://www.w3.org/TR/ws-arch/#whatis

44. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. Internet Computing, IEEE, 6(2), 86-93.

45. Yang, H. C., Dasdan, A., Hsiao, R. L., & Parker, D. S. (2007, June). Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data (pp. 1029-1040). ACM.

46. Bieberstein, N., Bose, S., Fiammante, M., Jones, K., &Shah, R. Service-Oriented Architecture (SOA) Compass: Business Value, Planning, and Enterprise Roadmap. New York: IBM Press. 25 Oct 2005. Print.

47. RADVIEW. RadView Software Whitepaper, Load Testing Web 2.0 Technologies: AjaxRIA-SOA-Web Services, Last accessed: August 15, 2012. http://www.radview.com/files/Load%20Testing%20Web%202.0%20Technologies%20%5BWhitepaper%5D.pdf.

48. Creeger, M. (2009). Cloud computing: An overview. ACM Queue, 7(5), 3-4.

49. Apache CouchDB Project. http://couchdb.apache.org/

50. Lee, W., Lee, C., Lee, J., & Sohn, J. S. (2009). ROA based web service provisioning methodology for Telco and its implementation. Management Enabling the Future Internet for Changing Business and New Computing Services, 511-514.

51. Liu, X., & Deters, R. (2007, March). An efficient dual caching strategy for web service-enabled PDAs. In Symposium on Applied Computing: Proceedings of the 2007 ACM symposium on Applied computing (Vol. 11, No. 15, pp. 788-794).

52. Yaws: Yet Another Web Server. http://yaws.hyber.org/

53. ab- Apache HTTP server benchmarking tool. Last accessed: September 2012. http://httpd.apache.org/docs/2.2/programs/ab.html

54. Apache Jmeter. Last accessed: October 2012. http://jmeter.apache.org/

55. Erlang programming language. http://www.erlang.org/

56. dets. Erlang stdlib reference manual version 1.18.3. Last accessed: October 2012. http://www.erlang.org/doc/man/dets.html

57. Gen_Server Behaviour. Erlang OTP design principles user's guide version 5.9.3. Last accessed: October 2012. http://www.erlang.org/doc/design_principles/gen_server_concepts.html

58. ets. Erlang stdlib reference manual version 1.18.3. Last accessed: October 2012. http://www.erlang.org/doc/man/ets.html

59. Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. ACM Computing Surveys (CSUR), 35(2), 114-131.

60. Sharma, S., & Deters, R. (2011). Supporting Interactive IPTV Apps with an Enterprise Resource Bus. In Third International Conference on Software, Services and Semantic Technologies S3T 2011 (pp. 127-131). Springer Berlin/Heidelberg.