Mobile Cloud Computing

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the degree of Masters of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Qian (Andy) Wang

PERMISSION TO USE

ABSTRACTION

As mobile network infrastructures continuously improve, they are becoming popular clients to consume any Web resources, especially Web Services (WS). However, there are problems in connecting mobile devices to existing WS. This thesis focuses on three of the following challenge: loss of connection, bandwidth/latency, and limited resources. This research implements and develops a cross-platform architecture for connecting mobile devices to the WS. The architecture includes a platform independent design of mobile service client and a middleware for enhancing the interaction between mobile clients and WS. The middleware also provides a personal service mashup platform for the mobile client. Finally, the middleware can be deployed on Cloud Platforms, like Google App Engine and Amazon EC2, to enhance the scalability and reliability. The experiments evaluate the optimization/adaptation, overhead of the middleware, middleware pushing via email, and performance of Cloud Platforms.

ACKNOWLEDGEMENT

First of all I would like to express my sincere thanks to my supervisor, Dr. Ralph Deters for his support, guidance, patience, and financial assistance throughout my entire two and half years of study. I would like to thank the members of my advisory committee: Dr. Julita Vassileva, Dr. John Cooke, and Dr. Anh van Dinh for their valuable advices and insightful suggestions. I also would like to thank Ms. Jan Thompson, Graduate Correspondent at the department of Computer Science, who has been very helpful throughout my study at University of Saskatchewan and very kind. Finally, I would like to thank my parents and friends for their unconditional love and selfless support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ART	Average Response Time
AWS	Amazon Web Service
BPEL	Business Process Execution Language
CP	Cloud Platforms
CS	Cloud Services
DC	Distributed Computing
DSL	Domain Specific Language
EC2	Amazon Elastic Cloud Computing
FIFO	First In First Out
GAE	Google App Engine
HCM	Hybrid Cloud Middleware
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	integrated development environment
JDO	Java Data Objects
JPA	Java Persistent API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MCC	Mobile Cloud Computing
MOM	Message Oriented Model
MS	Mashup Service
MWS	Mobile Web Services
OS	Operating System
OWL-S	Semantic Markup for Web Services
PaaS	Platform as a Service
POM	Policy Oriented Model
PSMP	Personal Service Mashup Platform

QoS	Quality of Service
REST	Representational State Transfer
ROM	Resource Oriented Model
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SA	Service Action
SaaS	Software as a Service
SED	Standard Deviation
SLA	Service Level Agreement
SOA	Service Oriented Computing
SOAP	Simple Object Access Protocol
SOM	Service Oriented Model
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VC	Virtual Client
VM	Virtual Machine
VN	Virtual Network
WS	Web Services
WSDL	Web Services Description Language
XML	Extensible Markup Language

Chapter 1 INTRODUCTION

Mobile handsets (phones) are expected to increase from the current 4.3B (billion) subscriptions to over 5.8B in 2013 [1] and thus dwarf the numbers of PCs (desktop, laptop, netbook) that are expected to rise from the current 1.1B to 2B in 2015 [2]. As mobile network infrastructures continuously improve, their data transmission becomes increasingly available and affordable, and thus they are becoming popular clients to consume any Web resources, especially Web Services (WS). Today, mobile devices like iPhone, Blackberry, Android, have included applications that consume WS from popular websites, such as Google, Facebook, and Twitter.

However, there are problems in connecting mobile devices to existing WS. Firstly, WS need to provide optimization for mobile clients. For example, the size of the WS messages needs to be reduced to fit the bandwidth of mobile clients. Secondly, mobile clients have to adapt to different kinds of WS, for example, SOAP and RESTful WS. The growing number of mobile clients and availability of WS also drives the needs of customizing and personalizing service mashups. This thesis investigates how Cloud Computing can help mobile clients connect to existing WS.

1.1 Web Services

WS is a technology linked to the idea of Service Oriented Computing (SOA) [3]. A Web Service [4] is

"A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (e.g. WSDL [5]*). Other systems interact with the WS in a manner prescribed by its description using messages* [4],

typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

There are two WS protocols standards, SOAP WS and RESTful WS. *Figure 1.1* shows SOAP WS in a service-oriented architecture.

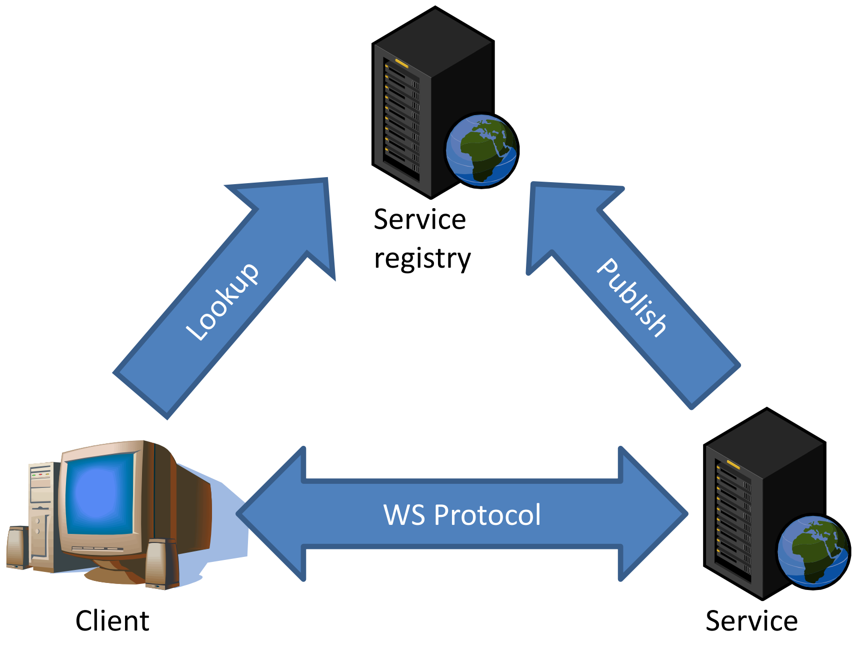
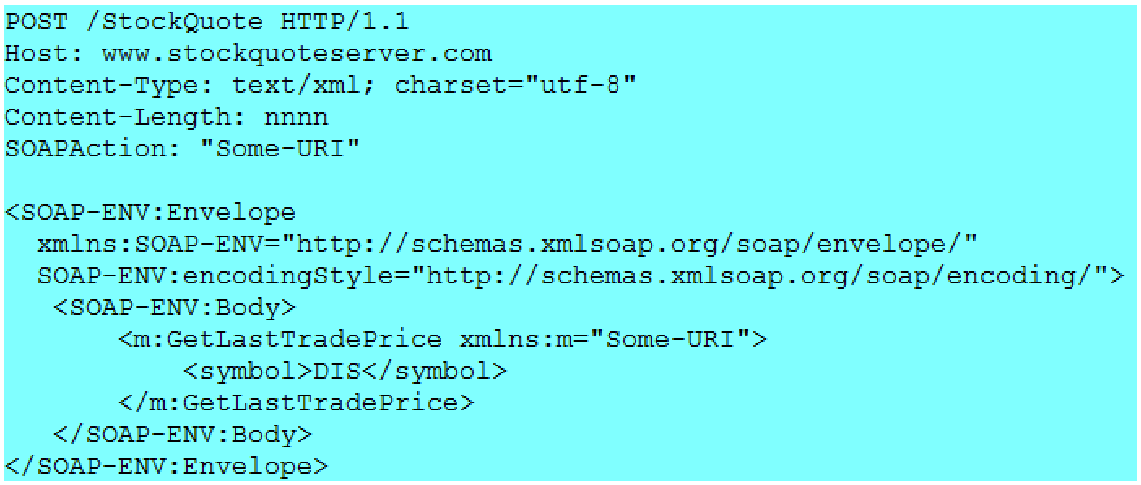


Figure 1.1 Service-oriented Architecture

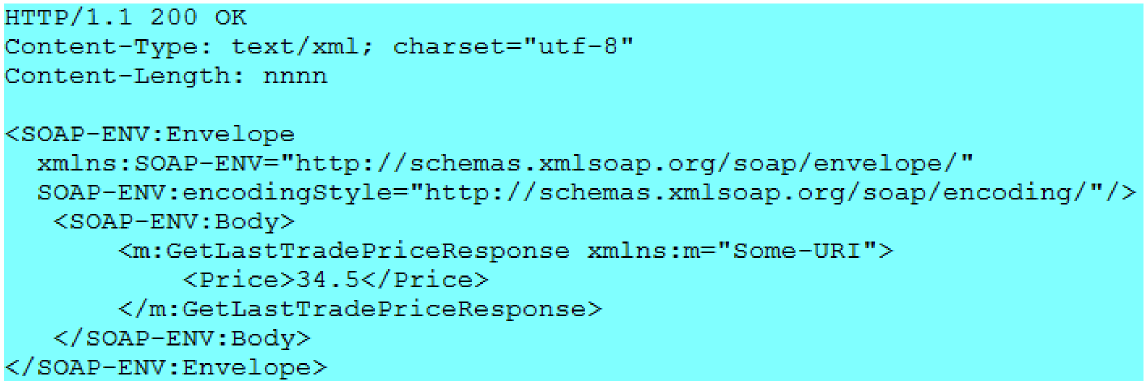
SOAP WS have well-adopted standards. Following is a typical scenario of consuming SOAP WS. Note that service discovery (step 1 and 2) is optional.

1. Service providers publish services to the service registry following the UDDI standard [6].
2. Clients also follow UDDI to discover the service they need.
3. Clients generate code for a specific SOAP WS from the WSDL [5].
4. Clients exchange SOAP messages with the service using the HTTP protocol. *Figure 1.2 & 1.3* shows an example of HTTP POST request and response contains SOAP message.



```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1.2 SOAP Message Embedded in HTTP Request [7]



```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1.3 SOAP Message Embedded in HTTP Response [7]

An alternative to SOAP WS are RESTful WS. RESTful WS were first introduced by Fielding [8] in his doctoral dissertation in 2002. They follow a resource-oriented computing paradigm. RESTful WS are presented as resources which are identified by a Uniform Resource Identifier (URI). Clients communicate with RESTful WS through the HTTP protocol, but the message body can follow any formats, for example XML and JSON, as long as the clients and the service providers agree upon it. RESTful WS also take advantage of the semantics of the HTTP protocol. For example, HTTP GET request is for acquiring a resource and HTTP POST request is for creating a resource. URL query, HTTP header, and request body can all be used as service inputs.

1.2 Cloud Computing

Cloud Computing is the latest addition to the myriad of distributed computing paradigms. Unfortunately, there is no clear definition of the term “Cloud Computing” and its origin is also not clear, e.g. some link it to the standard depiction of the Internet, others to the 2001 New York Times article referring to Dot.Net or the 2006 Eric Schmidt presentation. Vaquero [9] refers to cloud computing as a paradigm, which shifts the location of computing infrastructure to the network in order to reduce the costs associated with the management of hardware and software resources. In this thesis, Cloud Computing is divided into two parts, Cloud Platforms (CP) and Cloud Services (CS).

Cloud Platforms usually refer to application hosts that offer computational power, storage and Web access. Two well-known Cloud Platforms are Amazon Elastic Cloud Computing (EC2) and Google App Engine (GAE) [10]. EC2 is based on virtualization, where each EC2 instance is a Virtual Machine (VM). Users can choose different Operating Systems (OS) and hardware architectures to run on their VMs. Users are charged rental fee hourly for these EC2 instances ($0.085/hour for Linux/Unix usage and $0.12/hour for Windows usage). GAE is mainly a Web application platform. Users upload their Web applications to run on GAE. Currently, it supports only two programming languages, Python and Java. GAE is free as long as the application does not exceed the “free quota”. *Table 1.1* lists some of the limitations of the “free quota”.

Resource	**Free Default Quota**	
	Daily Limit	**Maximum Rate**
Requests	43,200,000 request	45,200requests/minute
Outgoing Bandwidth (HTTP &HTTPS)	1 GB	56MB/minute
Incoming Bandwidth (HTTP & HTTPS)	1 GB	56MB/minute
CPU Time	6.5 CPU-hours	1.5 CPU-minutes/minute

Table 1.1 GAE “free quota”

Cloud Services refers to software functions exposed as WS on the Internet, also called Web API. For example, services that provide information about the closest city based on geo-coordinates. According to “Programmable web” [11] (Mar 23rd, 2010), there are over 1798 available Web APIs related to shopping, blogging, data storage, music and so on. Many of them provide both SOAP and RESTful WS and are free under some limitations, like number of calls per minute and bandwidth. Service mashup is a popular term in Web which means defining a customized service using other services.

This thesis proposes a Mobile Cloud Computing architecture which uses Cloud-hosted middleware to support mobile clients consuming Web Services (Cloud Services). The architecture enhances the interaction between mobile clients and Web Services and provides a personal service mashup platform for mobile clients.

The rest of the thesis is structured as follows. Chapter 2 discusses problems with the current mobile service architecture and the idea of Mobile Cloud Computing. Chapter 3 reviews research related to Mobile Computing, Web Services and Cloud Computing. Chapter 4 describes the Mobile Cloud Computing architecture and the Personal Service Mashup Platform. Chapter 5 focuses on implementations of the mobile client and the middleware. Chapter 6 shows experiments and evaluations of the middleware and the mobile client. Chapter 7 summarizes the research contribution. Chapter 8 presents possible further work.

Chapter 2 PROBLEM DEFINITION

2.1 Consuming WS from Mobile Clients

Consuming WS from a mobile client (see *figure 2.4*) is different compared to the standard WS scenarios, due to the following factors.

- Mobile devices have limited resources (e.g. CPU power, screen size).
- The communication between client and service is established through wireless or cell network.
- Existing WS in the Cloud do not support mobile clients.

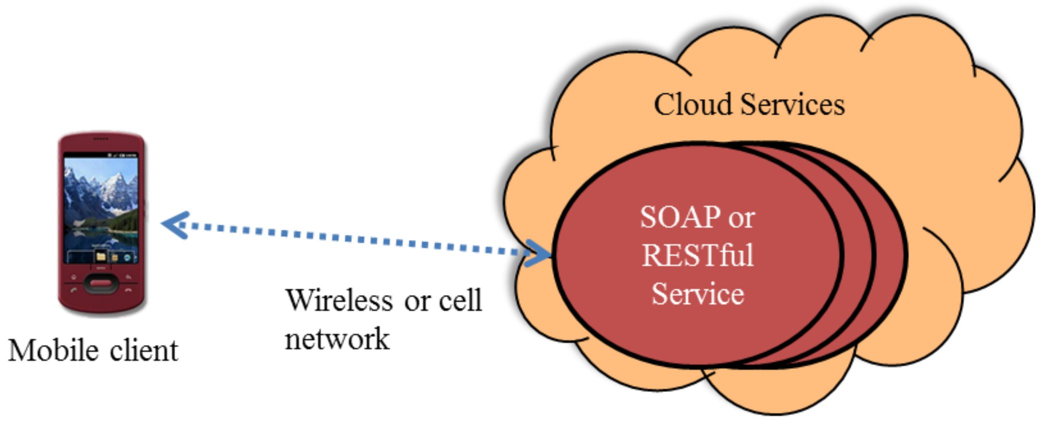


Figure 2.1 Consuming WS from Mobile Client

There are several **challenges** in the process of consuming Web Services from mobile clients. The following two are the focus of this thesis.

Challenge 1. Loss of connection: The interaction between client and service requires a stable connection. However, due to the mobility of the clients and the wireless network setup, mobile clients can be temporarily removed from the previous connected network and later may enter to another network. In such incidents, either service requests or responses may fail to be delivered to their destination.

Challenge 2. Bandwidth/Latency: Cell networks have limited bandwidth and are often billed based on the amount of data transferred. However, even a simple SOAP message often contains a large chunk of XML data, which consumes a lot of bandwidth and the transmission can cause major network latency. In addition, the SOAP message contains mostly XML tags that are not all necessary for mobile clients.

Challenge 3. Limited resources: Mobile clients are "thin clients" [12] with limited processing power. The limitations are intrinsic to mobility and not just the shortcomings of current technology [13]. For example, a service mashup involovs parsing and combining different WS results requires a lot of computation. The challenges are mimimizing the data processing on mobile clients and extending processing power beyond mobile clients. In addition, many mobile platforms do not include necessary libraries for SOAP WS.

2.2 The Idea of Mobile Cloud Computing

To overcome these challenges, I propose a Mobile Cloud Computing (MCC) architecture (see *figure 2.2*) which connects mobile devices to the Cloud Computing. The MCC architecture includes a mobile client and a middleware design.

There are two approaches to implement the mobile client: native applications and embedded browser applications. Native applications are built with specific programming languages supported by the mobile platforms. However, embedded browser applications can run HTML and Javascript in the embedded browser and use interfaces exposed by native application.

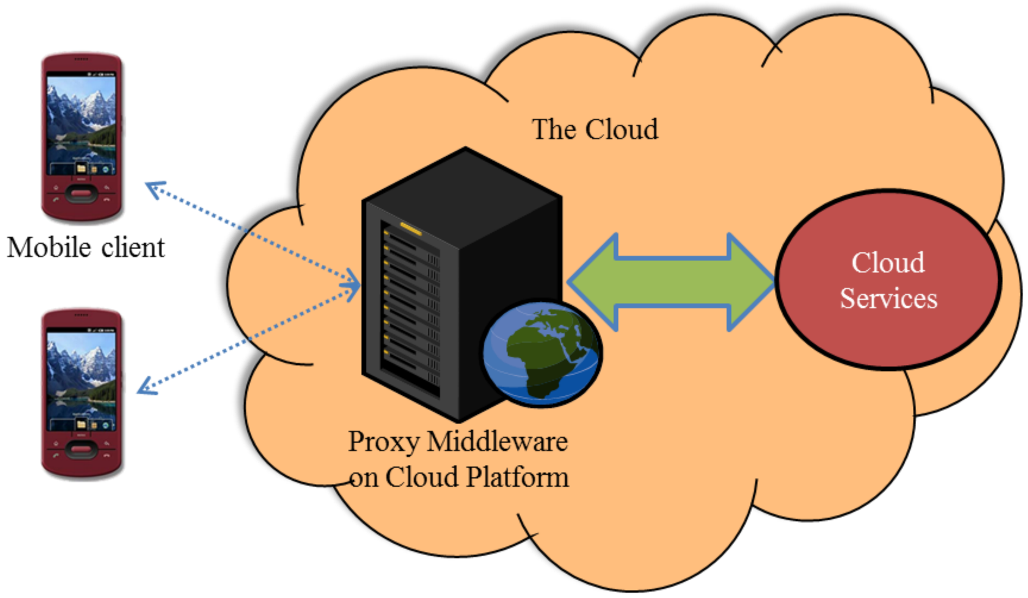


Figure 2.2 Consuming WS from Mobile Client through Proxy Middleware

The middleware [14] acts as a proxy that is hosted on the Cloud platforms which provide mobile clients access to Cloud services. The middleware improves interaction between mobile clients and Cloud Services, for example, adaptation, optimization and caching. The middleware also provides extended functions to mobile clients, such as service mashup. In general, the middleware enhances the functionality, reliability and compatibility of the interaction between mobile clients and Cloud Services. In order to overcome the challenges listed in the previous section, the Mobile Cloud Computing architecture provides the following features.

<u>(C1) Loss of connection</u>

- Client and middleware caching – Copies of service results are stored on both mobile clients and the middleware. When the mobile clients are not able to connect to the middleware, the client-side cache is used. When the middleware to WS connection is not available, the middleware returns its cached data to the mobile clients.
- Middleware push – When the middleware receives an update of service result, it immediately sends the update to mobile clients that are connected to the middleware.

When the mobile clients detect an available network connection, they automatically establish a connection to the middleware.

(C2) Bandwidth/Latency

- Protocol transformation – Protocol transformation reduces the latency as well as bandwidth of the client to service interaction. The middleware transforms SOAP WS to RESTful WS. SOAP is a verbose protocol which involves XML parsing, while RESTful WS can use light-weight format like JSON for the message. Transferring SOAP WS to light-weight protocols, like RESTful WS, reduces processing time as well as the size of the messages.
- Result optimization – Result optimization reduces the size of the service results, thus reduces the bandwidth used to interact with WS. The middleware converts the format of service results from XML to JSON and removes unnecessary data from the original service result. Less data transferring also reduces network latency.

(C3) Limited resources

- Cloud Computing – Connecting mobile clients to Cloud Computing extends the resources of mobile clients in a cost-efficient way. Cloud Services extends the functionalities of mobile clients, while Cloud Platforms provide computational power to mobile clients. The middleware is designed to be hosted on Cloud platforms, like GAE and Amazon EC2. Scalability is the top concern of the middleware. Cloud platforms provide automatic scaling for the middleware.
- Personal Mashup Platform – Service mashup allow mobile client to combine different services. However, service mashup requires interaction with WS and processing power. Because of the resources limitation (energy, processing power, software libraries) of

mobile clients, it is inefficient to do service mashup on the mobile clients. The middleware provides a Personal Mashup Platform which does service mashup for the mobile clients. The platform has generic interfaces for defining and consuming WS. The services are stored on the middleware and can be connected to form a work flow (a mashup service) which provides possibility to caching intermediate service results.

2.3 Research Goal and Hypothesis

The goal of this research is to find *an efficient and scalable architecture for connecting mobile devices to the WS*. The following lists three sub-goals and the features the architecture provides.

Goal 1. To enhance the interaction between mobile clients and Web Services

- Client and middleware caching
- Middleware push
- Protocol transformation
- Result optimization

Goal 2. To use the Cloud platform as a way to improve scalability and reliability of the middleware

- Cloud Computing

Goal 3. To provide a service mashup platform for mobile clients

- Personal Mashup platform

Chapter 3 LITERATURE REVIEW

This section reviews related research in the following fields: Mobile Web Services, interaction between mobile devices and WS, middleware for supporting mobile clients, and applications of Cloud Computing.

3.1 Web Services

According to W3C [7], the WS architecture includes a subset of four meta-models shown in *figure 3.1*. Although the concept is the same, a WS can have different levels of abstraction based on the four meta-models and each of them focuses on different perspectives.

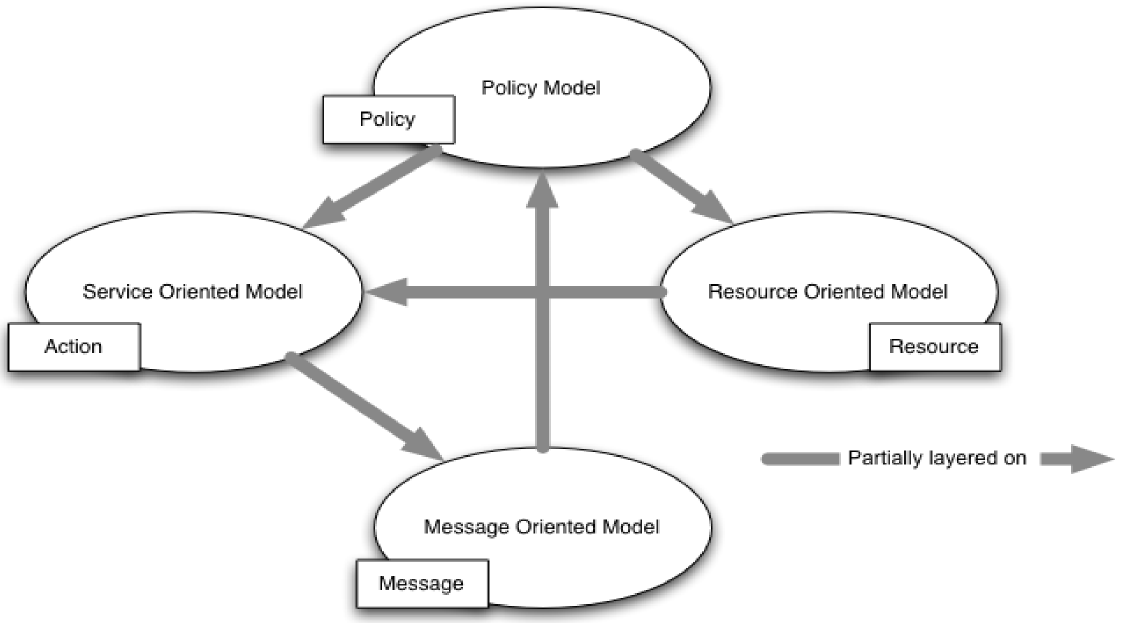


Figure 3.1 Four Meta Models of the WS Architecture [7]

- The Message Oriented Model (MOM) [7] focuses on messages, message structure, message transport without concerns of the reasons for the messages and their significance. A simple SOAP WS without discovery mainly follows the MOM.

- The Service Oriented Model (SOM) [7] focuses on aspects of service (the relationship to the real world) and action. Service discovery and composition are key concepts from SOM.
- The Resource Oriented Model (ROM) [7] views service as resource and focuses on the existents and owner relations of resources. RESTful WS is an example of ROM.
- The Policy Oriented Model (POM) [7] enforces policies (security and Quality of Service (QoS)) to both clients and services. POM can be applied to either SOM or ROM.

3.1.1 Mobile Web Services (MWS)

Mobile Web Service [15] refers to the mobile service client as well as to WS in the mobile environment. The release of new mobile platforms makes MWS easier to achieve. A survey done by Earl et al. [16] evaluated how well the current mobile platforms including Android, BlackBerry, iPhone, Symbian (S60), and Windows Mobile, support the concept of mobile network based research, for example, mobile service clients. According to the survey, all of these mobile platforms have certain limitations. For example, Android 1.0 lacks Bluetooth stacks and the ability to select network interfaces programmatically, which is fixed in Android 2.0. The iPhone framework lacks openness.

In my previous research [17], I proposed a novel architecture for consuming existing WS from a mobile client developed on an Android platform. Since RESTful WS rely purely on the HTTP protocol, the mobile client can consume RESTful WS through a built-in HTTP client. However, the mobile client does not support SOAP WS, because the Android platform lacks a library for parsing and creating SOAP messages. In addition, messages in XML format also takes more bandwidth and processing time compare to JSON format. The middleware thus provides style transformation (SOAP to RESTful), format conversion (XML to JSON) and other adaptations to the mobile client.

Resource utilization is another concern in a resource-constrained mobile environment. Previous research by Al-Turkistany [12] and Satyanarnynnan [13] indicate that the processing overhead of WS mainly comes from the usage of XML (about 400% compared to binary protocols). Tian [18] proposed an approach to improve performance with dynamic compressing of the WS response. In his approach, whether or not to apply compressing depends on the server load and the client network load. His experiments show that the performance of server and client improves only when the bandwidth of client network is scarce and the server is not under heavy load.

Caching is a common mechanism used to enhance user experience in server-client communication. For mobile service clients, caching is critical, due to their poor connectivity and limited bandwidth. Liu et al. [19] proposed a dual caching strategy to improve the performance and reliability of consuming WS from nomadic clients. In this model, caches are put on both nomadic clients and the server. The client cache is a proxy on the client devices and the server cache is on a remote computer which has reliable connection to the server. The overhead of Dual Caching grows linearly with request and response size, but the gain is a significant increase of performance for reading operations.

3.1.2 Summary

WS is currently the major technology for delivering services to end-user. In a mobile environment, most of the challenges are related to platform and resource constrains. Because RESTful WS only requires HTTP protocol, it suits the mobile environment better. Caching and optimizing/compressing are two approaches to deal with bandwidth constrain. In my approach, the middleware provides RESTful interfaces for mobile clients. It also caches and optimizes service results from Cloud Service.

3.2 Middleware for Mobile Device

Middleware is often used in a Distributed Computing (DC) system. DC systems [20] "consist of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network". Mobile clients are geographically distributed computers that connect to the middleware. In Emmerich's paper [21], he defined four requirements for general middleware.

- **Network communication:** In order to act as an integrated system, components residing on different hosts need to communicate with each other. It often involves some transport layer (TCP and UDP) and marshalling, a process of converting data structure to transferable format.
- **Coordination:** Since distributed systems have multiple points of control, different components need to coordinate and collaborate through synchronization.
- **Reliability:** Requests maybe lost during the network transmission. The middleware needs to deploy error detection and correction mechanisms to enhance reliability.
- **Scalability:** Distributed systems not only deal with client interactions, but also interactions between distributed components. Normally, distributed systems can scale horizontally (upgrade servers) and vertically (add more servers). Changes in the allocation of components could affect the system architecture, which refers as transparency in the reference model of open distributed processing [22].

- **Heterogeneity:** Components in a distributed system can be implemented with different languages and deployed on different platforms. Thus, the design needs to consider a heterogeneous environment.

Middleware is often used to extend functions for thin clients, like mobile devices. Uribarren et al. [23] proposed a middleware for adaptation in mobile environments. The proposed middleware hides the complexity of deploying ubiquitous applications. Applications are automatically moved between different platforms. For example, background music follows users when they change between devices. Devices are discovered transparently using the UPnP protocol [24]. Applications are packaged and stored in an App cache. Similar to the OSGi [25] bundle, the application package is an XML document that consists of configuration and a serialized executable. However, unlike OSGi, applications can launch independently, without pre-launching any coupled application.

When designing distributed systems, scalability should be the primary concern. Rajive et al. [26] did research on investigating scalable middleware to support mobile Internet applications. They designed a distributed middleware which resides between application servers and heterogeneous clients and provides presentation trans-coding, enforcement of quality-of-service, and security. The middleware performs session handoff when users want to move the current session state of an application running on the current client to another heterogeneous client. A distribution of middleware has a registration mechanism for load balance and communication protocol for exchange session data between middleware servers.

In Mobile Computing, middleware is commonly used for dealing with user context. Paolo et al. [27] proposed a context-aware middleware for Internet data services, called SCaLaDE (Services with Context awareness and Location awareness for Data Environments). Dey [28] defines context as “any information that can be used to characterize the situation of entities”. In SCaLaDE, the behaviors of services change based on three things: police, profile and context data which all saved on middleware. Polices include the capability and preferences of a particular end-node. The end-nodes are autonomous mobile agents which asynchronously collect context data and upload them to the middleware. The agents also describe patterns of system interconnection to the middleware. SCaLaDE consists of a series of upper level utility services such as query processing, caching, and transaction management, along with some lower level service such as naming and policy managing.

In conclusion, the proposed middleware solutions for mobile devices mostly focus on application and content adaptation. Coordination, scalability, reliability, and heterogeneity are four fundamental requirements for general middleware as well as middleware for mobile device [21]. Scalability can be achieved with distributed middleware [26]. Context can help middleware to adapt to the heterogeneous environment [27]. However, the goal of the research is to use middleware to improve the interaction between mobile clients and WS as well as use Cloud platforms to improve the scalability of the middleware.

3.3 Cloud Computing

The combination of virtualization, distributed computing and the service-oriented architecture creates a new computing paradigm, called Cloud Computing. According to Vouk [29], Cloud computing embraces cyberinfrastructure [30] which is one the key elements of successful information technology (IT). Based on the level of abstraction, Vaquero [9] defines three major scenarios in cloud computing.

- **Infrastructure as a Service (IaaS)** refers to service that exposes the hardware resources to users. Amazon EC2 [31] is a successful IaaS implementation in the market.

- **Platform as a Service (PaaS)** provides computational resources as high level application platforms. Google App Engine (GAE) [10] is an example of PaaS.
- **Software as a Service (SaaS)** focuses on exposing software functions as services (i.e. WS). Many service providers including Google, Yahoo, and Amazon offers their software functions as WS. Programmable Web [11] collected thousands of Web APIs from various categories.

Ostermann et al. [32] did an early performance evaluation of Cloud Computing by comparing Amazon EC2 to scientific computing infrastructure such as grids and PPIs. For a single job with a single EC2 instance, the CPU performance for floating point and double operation is 6-8 times lower than the claimed maximum of ECU (CPU unit defined by Amazon, one ECU equals 4.4 gigaflops per second) and the sequential IO operation has generally better performance compared to similar systems. For a single job with multiple EC2 instances (clusters), efficiency decreases with the increase of EC2 instances, due to the high network latency. However, for some jobs such as DGEMM [33], STREAM [34] and RandomAccess [35], EC2 clusters have similar or better performance than HPC clusters [36].

There are several open Cloud implementations. Vouk [29] presented an IaaS implementation based on Virtual Computing Laboratory (VCL) [37]. The end nodes include IBM BladeCenter blades [38] and computers in a university lab. The VCL implementation provides similar services like Amazon EC2, Map Reduce environment [39], and sub-cloud for Grid Computing [40]. Running since 2004, the VCL implementation reveals some open issues, like Cloud provenance data, utilization, optimization and portability of image.

Another open IaaS implementation similar to Amazon EC2 is Eucalyptus [41]. From the entry-point to end-node, there are four controllers: Cloud, Storage, Cluster, and Node controller.

They all communicate through WS interfaces. Instances are run as Virtual Machines (VM) on the end-nodes where node controllers are installed. At the cluster level, VM instances are interconnected via a Virtual Network (VN) which grantees connectivity of single access of a “set” of instances, isolation of separated Cloud allocation, and performance with option of choosing native network without VN. Eucalyptus also provides a storage service for VM images and user data similar to Amazon S3.

In summary, Cloud Computing is a new computing paradigm which aims to reduce the cost of both development and deployment. However, the real implications of using Cloud Computing vary in each case. Most Cloud systems are proprietary, and rely on infrastructure that invisible to researchers [41]. Hence, there are restrictions imposed by providers. Open Cloud implementations like Eucalyptus, provide an easy solution for IaaS and opens the possibility of creating private Cloud, but there are some issues that needs to be considered.

3.4 Service Composition and Mashup

Web Services are mainly derived from the service-oriented architecture that is based on Service-Oriented Computing (SOC). SOC [42] is a computing paradigm that utilizes services as fundamental elements for developing applications. In SOC, services are autonomous, platform-independent computational entities that can be used in a platform independent way [43], thus new services can be composited from existing services with low-cost. There are currently two styles of composing WS, the formal WS composition and light-weighted WS mashup.

There are several approaches to WS composition such as BPEL(J) [44], Semantic Web (OWL-S) [45] and Web Component [46]. According to the review of Liu et al. [47], all of them introduce strong overheads (developer’s skill and supporting infrastructure). However, this research focuses on a light weighted approach to service compositions, WS mashups [48] which "typically serve a specific situational need (short-live) and are composed of the latest, easy-to-use Web technology (RESTful WS, RSS and Atom)."

One subset of WS mashup is Enterprise Mashup (EM) [49] which is a paradigm that "end-users are empowered to adapt their individual business to their individual and heterogeneous needs". *Figure 3.2* shows the two styles of EM, wiring and piping. Hoyer [50] also categorized mashup tools in the market based on their functionality and target group.

- **Resource** is the actual content, data or application which expose interface as Web API, WS, and other. Piping integrates resources to processing chain/graph by directing output of one resource to input of next resource.
- **Widget** is a graphical interface which provides simple user interaction abstracting from the underlying resources. Wiring interconnects visually input and output parameters of widgets, which requires no programming skill.

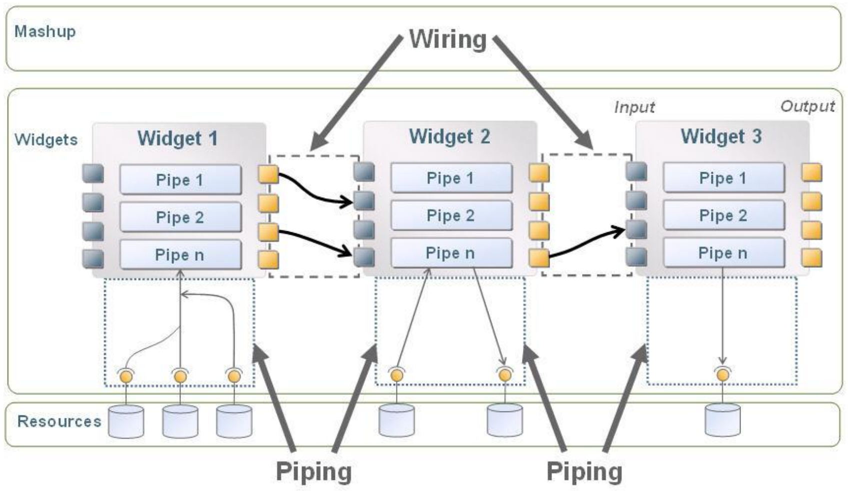


Figure 3.2 Enterprise Mashup (Wiring vs. Piping) [35]

Piping often involves Domain Specific Language (DSL) [51]. Maximilien [52] designed an online platform for service mashup based on DSL. Users can create and share mashup services using Web browsers interface. The core of the platform is a DSL Engine which can generate a Ruby on Rail application from the DSL code defined users. The DSL supports three essential functions in defining service mashup, data mediation, process/protocol mediation, and user interface customization.

The idea of mobile mashups has gained popularity recently. Xu [53] developed a mashup platform for mobile devices based on Aspect-oriented programming technology. The key feature of the platform is the mashup management framework which monitors and controls mashup execution. Both execution status and performance is monitored and compared to expecting QoS defined by Service Level Agreement (SLA). The adaptive engine then optimizes or resolves the QoS problems, for example, replacing composed services.

WS mashup shares certain advantages over the formal WS composition method, since a WS mashup requires less programming skills and overhead. Mashups can be achieved by either wiring at the interface/widget level or piping at the resource/service level. The proposed mashup platform supports “piping” mashups. The piping and QoS management is done on the middleware, but the interface for defining a mashup service is on mobile client.

3.5 Conclusion

Mobile technology continues to grow, which makes it easier to consume WS from mobile devices. Personalized service mashup is also required by mobile clients. However, mobile devices are still considered constrained devices compared to stationary computers. When developing a mobile WS client, developers and service providers need to consider the heterogeneity of mobile platforms.

WS is a widely adopted approach for providing service, but most existing WSs in the Cloud are not aware of mobile clients. RESTful WS is especially designed for lightweight and flexible interactions, for example mobile-service interaction. An available approach to add adaptation and service mashup to mobile clients is using middleware. Considering the four requirements for distributed systems, IaaS and PaaS can be ideal places to host middleware. *Table 3.1* categorizes and lists all the reviewed research based the area they can be applied on.

Mobile Client	• Earl et al. [16] reviewed different mobile platforms • Mobile clients consume RESTful WS through a proxy [17]
Client-server Interaction	• Meta-models of Web Service (MOM, SOM, ROM, POM) [7] • Improve response time with Dual Caching on PDA [19] • Reduce bandwidth consumption with compressing service results [18]
Middleware	• Definition of distributed computing [20] • General requirements of middleware [21] • Middleware support transparent deployment of ubiquitous applications [23] • Middleware for sharing application context across different mobile clients [26] • Middleware for managing user context collected from mobile agents [27]
Cloud Computing	• Different categories of Cloud Computing (SaaS, IaaS, PaaS) [9] • Early performance evaluations of EC2 [32] • Cloud implementation based on Virtual Computing Laboratory [30] • Personal Cloud implementation, Eucalyptus [29],[41]

Service Mashup	• Review of different approaches to WS composition [45] • Enterprise Mashup [48-50] • DSL based service mashup [52] • Mobile mashup based on Aspect-oriented programming [53]

Table 3.1 List of research solutions

In summary, the current research indicates:

- It is possible for mobile clients to consume WS [17].
- Adaptation is needed for mobile clients to interact with WS [18] [19].
- Middleware can extend the functionalities of mobile clients [23] [26] [27].
- Cloud Platforms are cost-efficient, scalable and reliable for hosting middleware [32].
- Service mashup is light-weighted WS composition and can be designed on server side [48-50] [52] [53].

However, there are still **open questions** related to connecting a mobile device to the Cloud that remain, namely:

- How to design a complete architecture for mobile devices to connect to Cloud Services?
- How to design mobile service clients compatible on different mobile platforms?
- How to improve interaction between mobile clients and Cloud Services?
- How to achieve personalized service mashup for mobile clients?
- How to implement the middleware on Cloud Platforms?

Chapter 4 DESIGN & ARCHITECTURE

4.1 Overview

The goal of the Mobile Cloud Computing (MCC) architecture is to provide a proxy for mobile clients connecting to Cloud services. *Figure 4.1* shows an overview of the MCC and its main features. The architecture consists of three parts, the mobile clients, the middleware and the Cloud services. Since Cloud services are usually controlled by service providers, the middleware performs all the necessary adaptation to the mobile clients.

Some services require real-time updates, for example, news, Blog, and Twitter service. The middleware also pushes updates of service results to mobile clients via HTTP or email immediately after it receives the updates.

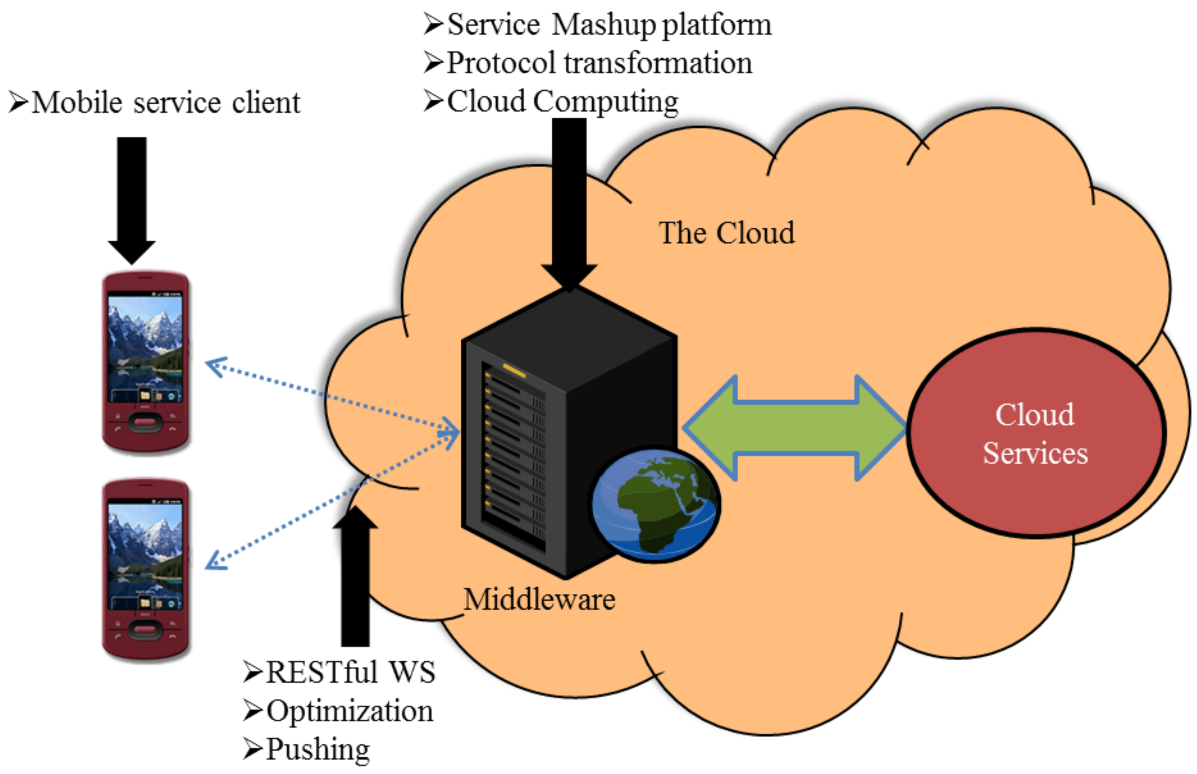


Figure 4.1Overview of MCC

The middleware is responsible for consuming the Cloud Services whether they are SOAP or RESTful WS and delivers the service result to the mobile client. On the mobile client, users can define WS or mashup services and later execute the pre-defined WS on the fly. The middleware provide RESTful WS interface for the mobile clients. *Figure 4.2* indicates how to consume/execute a pre-defined WS. Note that the execution starts with a HTTP GET request whose URL path contains the resource identifier to the WS. When WS are executed through the middleware, the follow steps are involved in the middleware.

1. The mobile client sends a HTTP GET request with an identifier of a WS to the middleware.
2. The middleware deals with interactions to the WS (and generates SOAP WS client if necessary).
3. The middleware extracts (JSON or XML parsing) the required service results from the original service result and form a new service results in JSON format.
4. The middleware stores a copy of result with the service ID in the database and returns the optimized result to the mobile client

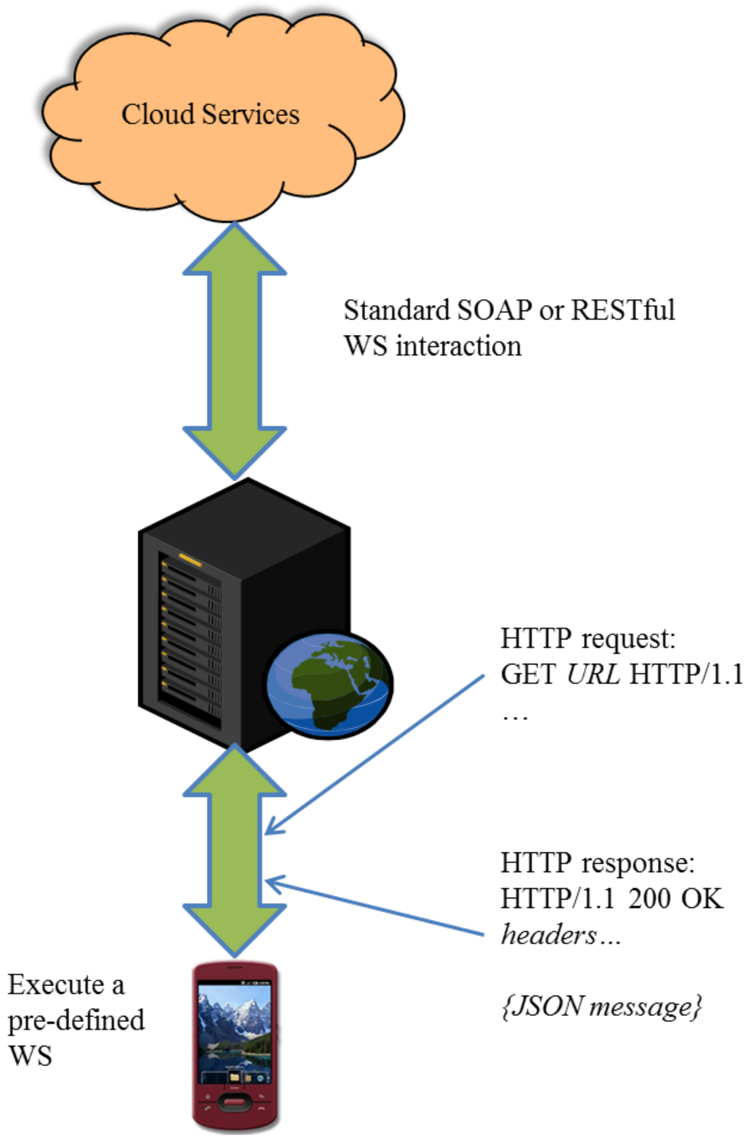


Figure 4.2 Consume/Execute a WS in MCC

The middleware is also a Personal Service Mashup Platform (PSMP) that is based on a novel data structure which represents WS as objects. The next section talks about the middleware design and how these functions are achieved. Section 3 describes the design of PSMP. The rest of the sections present implementations of mobile client and Cloud middleware.

4.2 Middleware Architecture

Figure 4.3 shows the components in the middleware architecture. The middleware has a RESTful service interface for mobile clients. Through the management interface, users can define and manage user profile, Mashup Services, Service Actions, and their parameters and results. All the requests through the management interface are passed to the service repository which reads and write data from and into the storage. The execution requests of Service Actions go through the service execution interface. These requests are primarily mapped to read operations in the service repository. The service executer composes service requests and passes them to the HTTP client which sends outgoing request to Cloud services. In general, the middleware provides the following features to improve the interaction between mobile clients and Cloud Services.

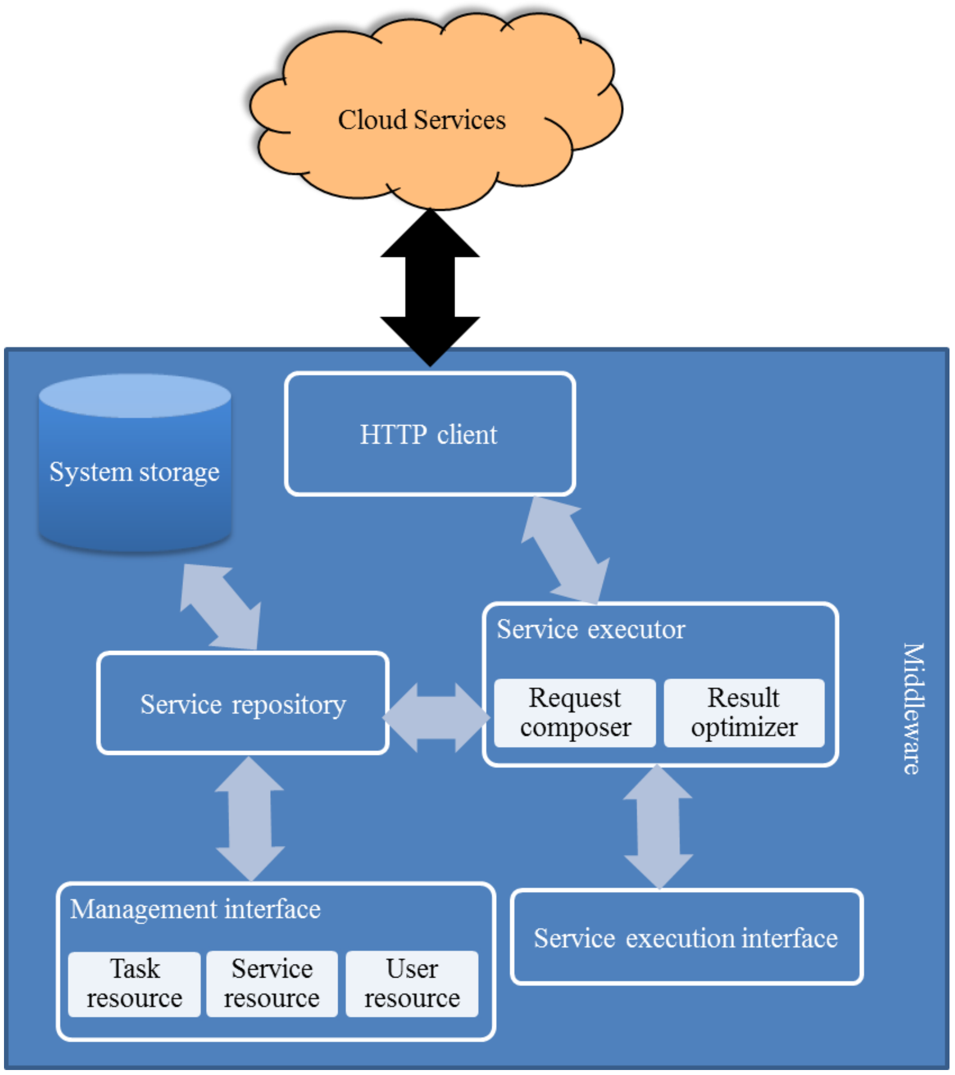


Figure 4.3 Middleware Architecture

- <u>Middleware pushing:</u> Mobile clients can subscribe to service resources and explicitly update service results cached on the middleware through the management interface. When the middleware receives an update of service results, it sends the update to all the mobile clients that subscribed to this service result. The update is pushed to the clients (e.g. via email).
- <u>Protocol transformation:</u> The middleware transforms the SOAP WS into RESTful WS. The service executor handles normal HTTP requests for RESTful WS as well as SOAP messages for SOAP WS. If the service is SOAP WS, the service executor generates a

specific SOAP client based on the provided WSDL, and then uses the generated client to interact with the Cloud Services. To the mobile clients, all the services executions are through a RESTful WS interface.

- Optimizing results: An unprocessed WS response contains data within a service specific format. However, there are two problems. First, the mobile clients do not need all the data. For example, the user may only need 5 instead of 10 news stories. Second, the original data format may also not be efficient for mobile clients. The result optimizer first extracts the required part of data from the raw response, and then makes a copy of the extracted result in various formats, for example, mobile HTML for mobile browsers and JSON for native mobile applications. The middleware also caches these copies of result in the service repository.
- Middleware Caching: Caching is based on the mashup services. The service repository saves the optimized service results into system storage for the latest execution of the mashup services. The service results update when the parameters of a mashup service change. Users can also clean the cache via the management interface.

Like most middleware, scalability is always a major concern. My approach is to take advantage of the Cloud platforms to host the middleware. Amazon EC2 and Google App Engine are the Cloud Platforms examined in this research. They both have very different service model and performance characteristics. Chapter 5 describes the middleware implementations based on EC2 and GAE in details.

4.3 Personal Service Mashup Platform

On the mobile client side, the middleware has a user interface which lets users define mashup services. The middleware has a service storage which stores user defined service data and an execution engine which executes WS and pipes input and output of WS. In order to support a service mashup, the middleware must first support consuming existing WSs. Specific WS calls are pre-defined by users using the mobile client and stored in the service storage for future execution. The following gives a user scenario of how to consume a WS from the mobile client through the middleware.

Kevin is a mashup service developer. He wants to know all the coming events in his city using his mobile phone. He knows that Yahoo Upcoming (RESTful WS) offers such service and reads its online API document which describes how the service is used (e.g. providing coordinates as parameters). Through the user interface on the mobile client, he then defines a mashup service (task) which contains a service action with all the required parameter and desired results. Finally, he executes the mashup service and gets the result displayed on the mobile client. Figure 4.4 shows the process with a sequence of screenshots on the mobile client.

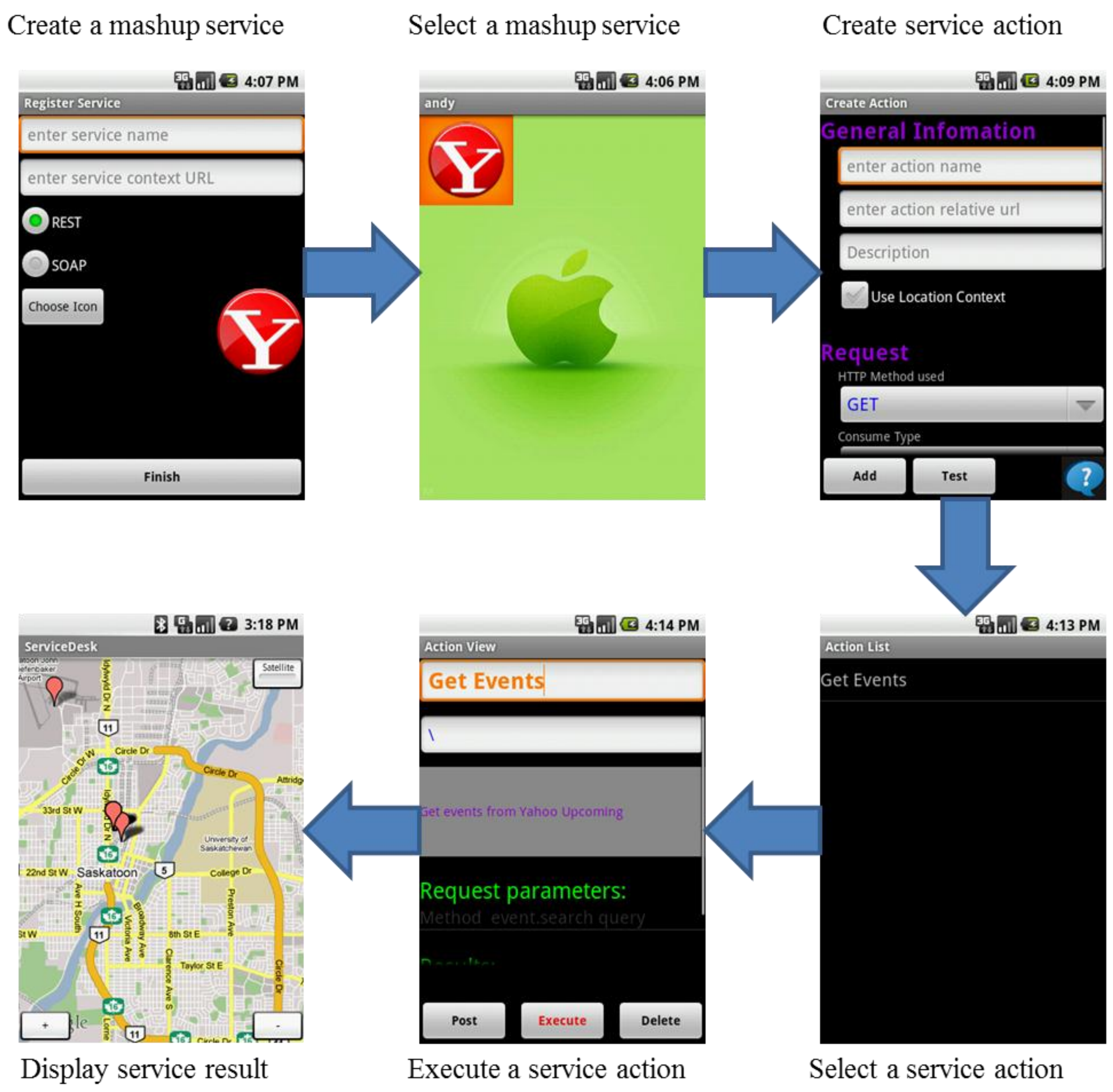


Figure 4.4 Screenshots of the mobile WS client

4.3.1 Service Entities

User defined WS calls are stored in the service storage as service entities. A WS essentially consists of two parts of information: service configuration describing the properties of the WS (meta-data) and how to consume the service (parameters), and the user-specific parameter values needed to be passed to the WS. There is a format describing a RESTful WS (WADL), but it is not widely adopted. In the middleware, service entities abstract the essential elements of RESTful WS. In the future, service entities will also be compatible for both SOAP and RESTful WS.

System storage is a database implementation. Each kind of entity is presented as a table. There are four kinds of entity: Mashup Service (MS), Service Action (SA), parameter, result and result value. *Table 4.1* lists and describes the key attributes of each kind of entity. *Figure 4.4* shows the hierarchical relations of each kind of entity.

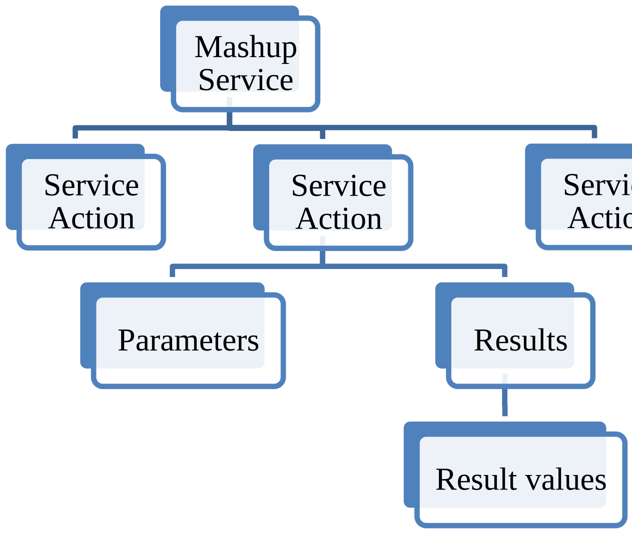


Figure 4.5 Hierarchical View of Service Entities

- Mashup Service (MS): MS is a container for service actions. The MS provides users with a conceptual grouping of similar service actions as well as a boundary for preventing outside access. Users can also share their MS with others.
- Service Action (SA): SA is the primary entity on which the service mashup is based. SA defines all the necessary attributes to consume a WS: the URL to find the WS, the interaction protocol, the parameters required by the WS, the desired results and so on. Piping is also applied on the SA level, which will be described later.

- Parameter: Parameter is not only a name-value pair, but also consists of meta-data, for example, the source of a value (user input or output of other service) and how they are passed to the WS (through URL query, HTTP header, content or else).
- Result: The result describes how the proxy processes the WS response and how the clients present the result. Only data interesting to the user will be extracted from the response. According to the result type, the clients display it in various forms (video, audio, image and so on).
- Result value: The reason for separating result and result value is that a result can have multiple copies of values depending on targeted clients. The proxy keeps copies of results in local database for different clients, for example HTML for browser, JSON for native application.

Entity kind	Attribute	Description	Possible Values
Mashup Service	owner	The creator of the MS is the owner	user id (i.e. e-mail)
	isPrivate	Whether or not let others access the MS	True/False
	style	WS style (SOAP or RESTful) of the containing SA	SOAP/RESTful
	base URL	The common base URL of the containing SA	URL (http://api.yahoo.com)
Service Action	name	Name of the SA	name identifier
	method	HTTP request method	GET/POST/PUT…
	consume format	The format that the WS accepts	Format standards (XML, JSON, Atom)
	produce format	The format of the WS response	Format standards (XML, JSON, Atom)
	parameters	Parameters passed to the WS	Reference to parameter entities
	results	Result definition	Reference to result entities
Parameter	name	Parameter name used in the WS	name identifier
	value	Parameter value	Depend on src
	src	Source of the parameter	User /Mashup
	Embedded type	How the parameter will be passed to the WS	Path/Query/Content…
Result	name	Result name	name identifier

	path	Path for extract result from WS response	XPath for XML
	type	Result content type	Text/Video/Audio…
	values	Result values	Reference to result value entities
Result value	received date	When the value is received (cached)	Date
	expire	How long the value expires	Time
	target	The targeted client	Mobile/browser/internal
	content	The actual content of the result value	Blob (Bytes)

Table 4.1 Entity Attributes table

4.3.2 Piping & Workflow

The Personal Service Mashup Platform (PSMP) provides mashup support based on piping. In Computer Science, piping [54] refers to chaining processes, so that the output of one process feeds to the input of next one. In PSMP, the result of a SA can be piped into the parameter of another SA using identifiers and dot notations. Note that for security reasons, piping can only apply to the SAs within the same MS. *Figure 4.6* shows an example of mashup with piping.

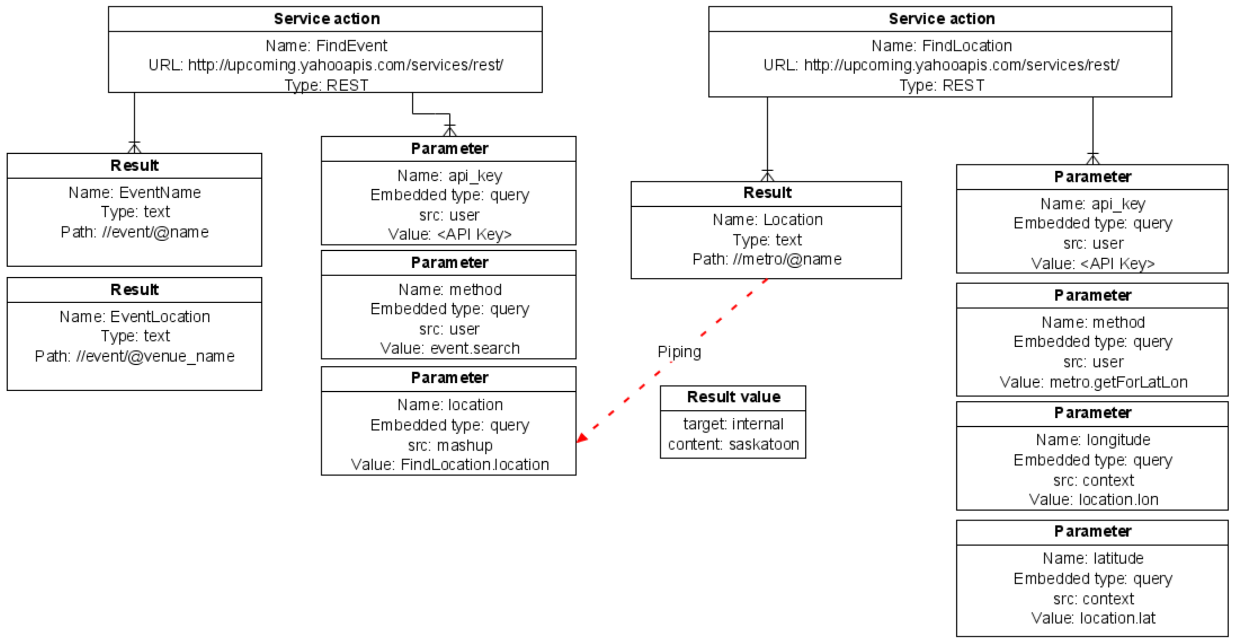


Figure 4.6 Mashup Example

The example involves two Service Actions (SAs) for finding events in the nearest city to the user's current location using Yahoo Upcoming service. The "location" result of the

"FindLocation" SA is piped into "location" parameter of the "FindEvent" SA. Because the source of the parameter is a "mashup", the value should be a reference to a result in the format as "<SA id>.<result id>". The result value used for piping is the one targeted for "internal".

The service mashup forms a workflow. In the Personal Service Mashup Platform (PSMP), a workflow connects several Service Actions (SAs) in a tree like structure. SAs in a lower level contribute (pipe results) to their parent SA. The root SA is the final goal of the mashup. *Figure 4.7* shows a small example of a mashup workflow. Workflow control involves three factors.

- **State control:** Each workflow has a state of execution. If the execution stops, it can be picked up later from its previous state.
- **Flow control:** SAs at the same level can be executed asynchronously. However, the parent SA must wait for its children to complete, since the parent depends on the outputs of its children.
- **Fault tolerant control:** If one of the child SAs fails, an alternative Service Action (SA) will replace it. If all of them fail, the workflow prevents further execution. It is the users' responsibility to replace broken SA.

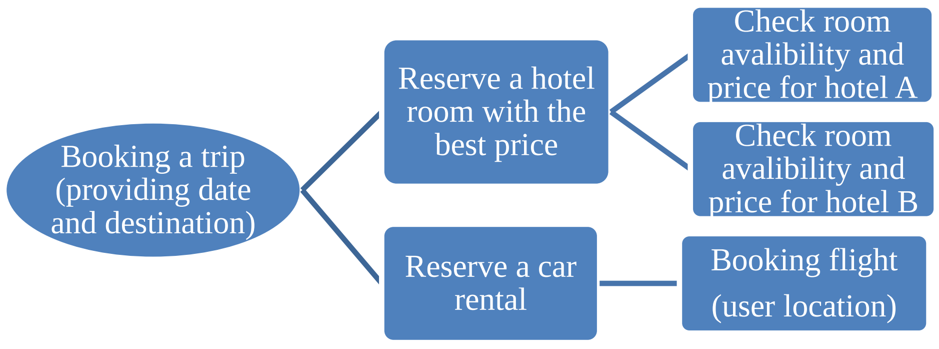

Booking a trip (providing date and destination)
Reserve a hotel room with the best price
Check room avalibility and price for hotel A
Check room avalibility and price for hotel B
Reserve a car rental
Booking flight (user location)

Figure 4.7 Example of Mashup Workflow

4.4 Mobile Client Design

On most of the new mobile platforms like Android and Blackberry OS 5.0, mobile WS clients can be run either as pure native applications or embedded browser applications. Native applications are platform dependent. They must be implemented using the programming languages that the platform supports and live in the application layer of the platform. *Figure 4.8* shows where native applications locate on the Android platform. Functionality and performance of native applications mainly depends on the core API libraries and the mobile platforms.

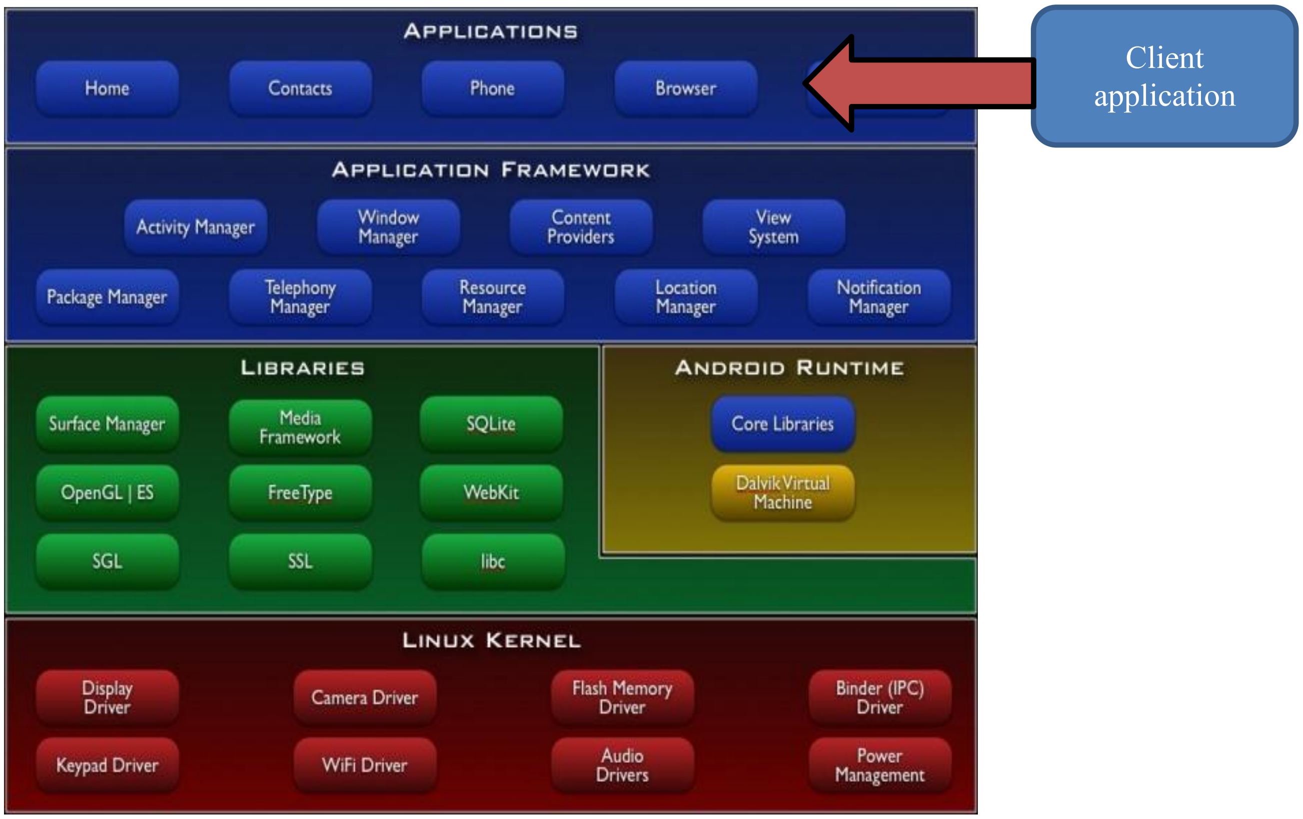


Figure 4.8 Native application on Android

Another way to implement the mobile WS client is using an embedded browser. The client application runs on a Web browser which is embedded inside of a native application. *Figure 4.9* shows how the embedded browser technology works. The client application can be completely implemented using a browser supported language like HTML, CSS, and JavaScript.

The embedded browser can load custom JavaScript libraries which can access the native codes inside the application.

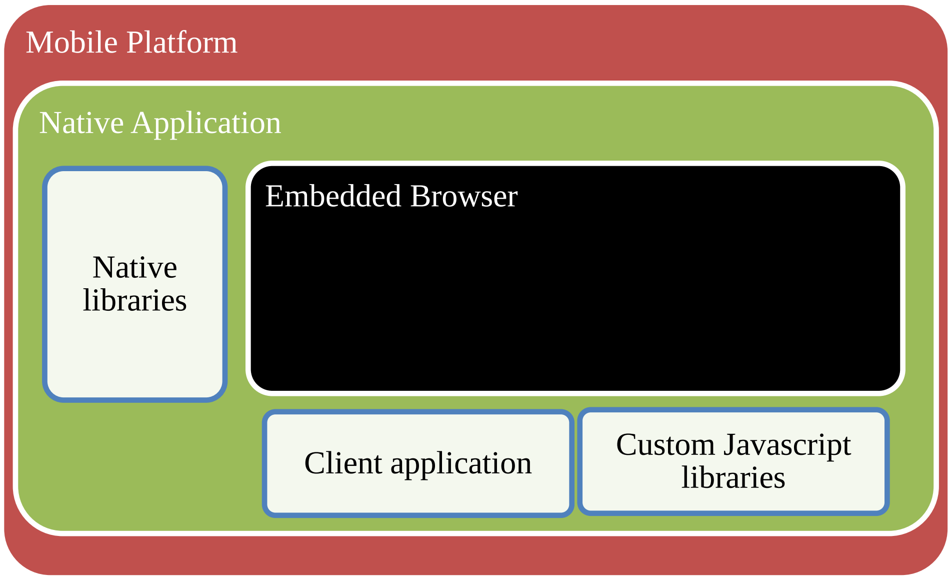


Figure 4.9 Client Application on Embedded Browser

Table 4.2 lists pros and cons of native and pure embedded browser applications (without custom libraries).

	Native application	Pure embedded browser application
Pro	• Performance (compiled code) • Full access to native API • Easy to test and debug • Rich GUI features	• Platform independent • Less specialty required • Easy to maintain and upgrade
Con	• Platform dependent • Maintenance and upgrade cost	• Browser compatibility • Performance (interpreter) • Browser limitations • No access to native API

Table 4.2 Pros and cons of native and pure embedded browser applications

The proposed mobile client architecture is a hybrid solution which combines both native and embedded browser application. *Figure 4.10* is the overview of the client architecture. It follows a basic Model View Controller (MVC) pattern. The User Interface (UI) is designed within the embedded browser using HTML, CSS and JavaScript. When the UI components need service data, they invoke the custom JavaScript libraries to pull the data from local cache. If the local cache does not contain a recent copy of inquired data, the RESTful client interacts with the middleware to get the data. The data are then passed to the data module and stored in the local file system. Note that the data passed to the embedded browser is in JSON format which can be easily parsed by JavaScript.

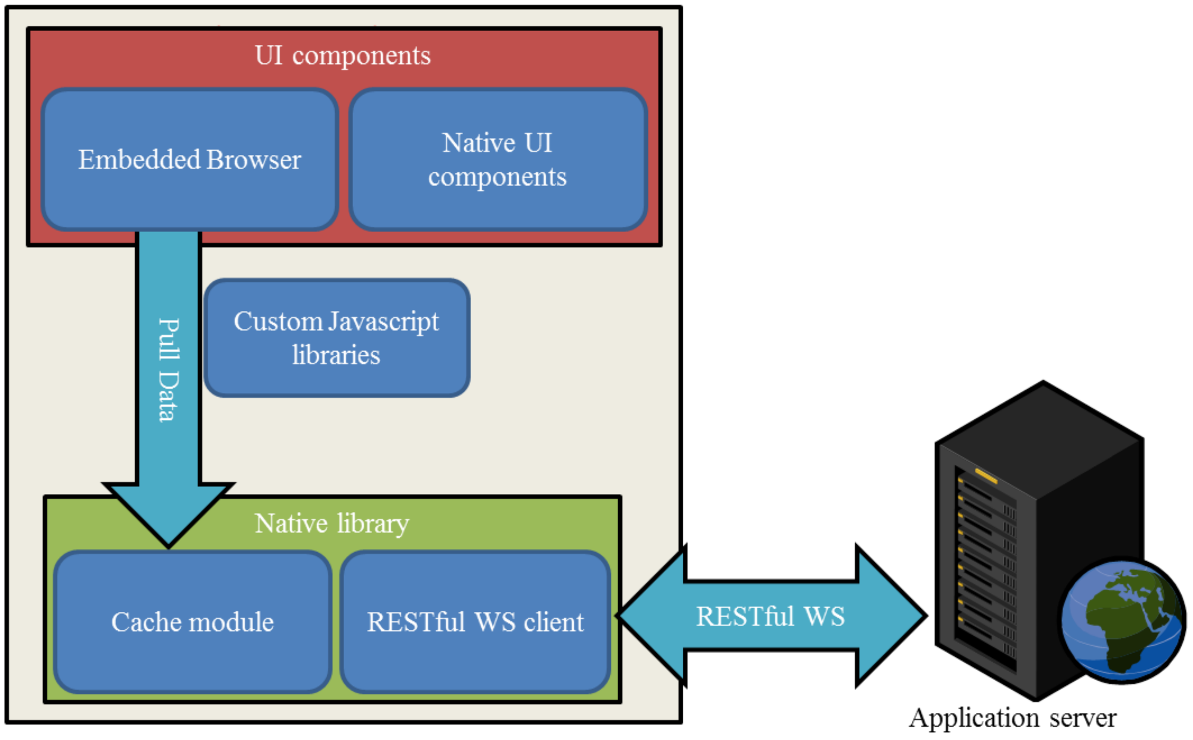


Figure 4.10 Mobile client architecture

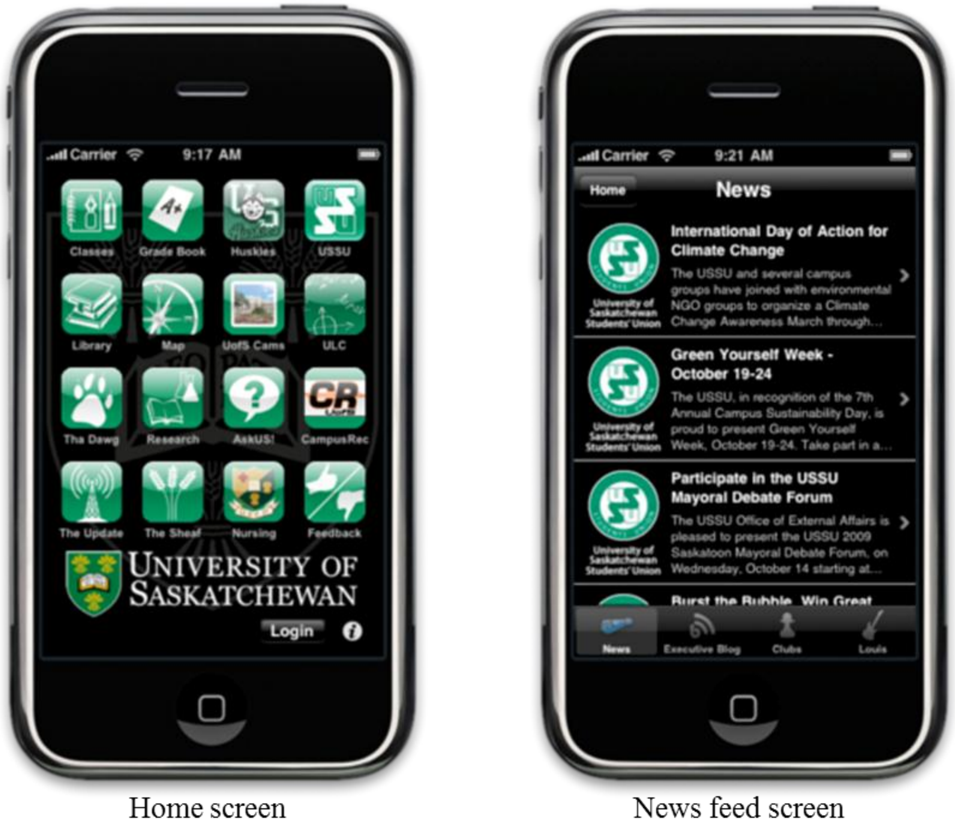
The separation of UI components and the client makes the architecture platform independent. To change the application to a pure native application, the embedded browser UI can be replaced by native UI and the client can be reused. The RESTful client can also implement push technology. Push technology enables a server to push content to the clients, in order to optimize the data traffic, energy and bandwidth used. The next chapter describes the Blackberry implementation of the mobile client.

Chapter 5 IMPLEMENTATION OF THE MOBILE CLIENT AND CLOUD MIDDLEWARE

5.1 Mobile Client implementation on Blackberry

I implemented the proposed mobile client architecture on Blackberry OS 5.0 as a BlackBerry WebWork application with native libraries. The BlackBerry WebWork is an embedded browser framework released in October of 2009. It includes Javascript libraries that implement several common functionalities of the Blackberry OS, for example, location service and file system access.

To verify the mobile client design, I integrated the design with an existing iPhone application for university students, called iUsask. The application is re-implemented with the mobile client design on Blackberry. Using the application, student can check their class and grade information as well as news from various departments. *Figure 5.1* is some screenshots of the iUsask on iPhone.



International Day of Action for Climate Change
The USSU and several campus groups have joined with environmental NGO groups to organize a Climate Change Awareness March through...

Green Yourself Week - October 19-24
The USSU, in recognition of the 7th Annual Campus Sustainability Day, is proud to present Green Yourself Week, October 19-24. Take part in a...

Participate in the USSU Mayoral Debate Forum
The USSU Office of External Affairs is pleased to present the USSU 2009 Saskatoon Mayoral Debate Forum, on Wednesday, October 14 starting at...

Burst the Bubble, Win Great

Figure 5.1 Screenshots of iUsask on iPhone

The client application can be divided into three layers, User Interface (UI), controller and cache manager. The UI layer has two implementations, native UI and embedded browser UI. *Figure 5.4 and 5.5* show how they look like on the device. *Figure 5.2 and 5.3* show the architecture of both implementations. The controller is the key coordinator among the UI, middleware, and cache manager. The controller creates the UI and gets data from the RESTful client or cache manager. If network connections are not available, the controller passes cached data to the UI components. Otherwise, it invokes the RESTful client to get data from the middleware. The cache manager then saves recent received data on a local file system (device memory or SD card).

With the native UI, the client interacts with the middleware asynchronously. When the native UI requires data, it passes a callback to the controller and continues to receive UI events.

The controller starts a new thread to interact with the middleware. When the data arrives, the UI gets updated through the callback. With this model, the native UI can be updated as soon as the data changes. The embedded browser needs to wait for the data to arrive, because the native library cannot receive a JavaScript callback. The embedded browser also cannot be updated automatically when the data changes.

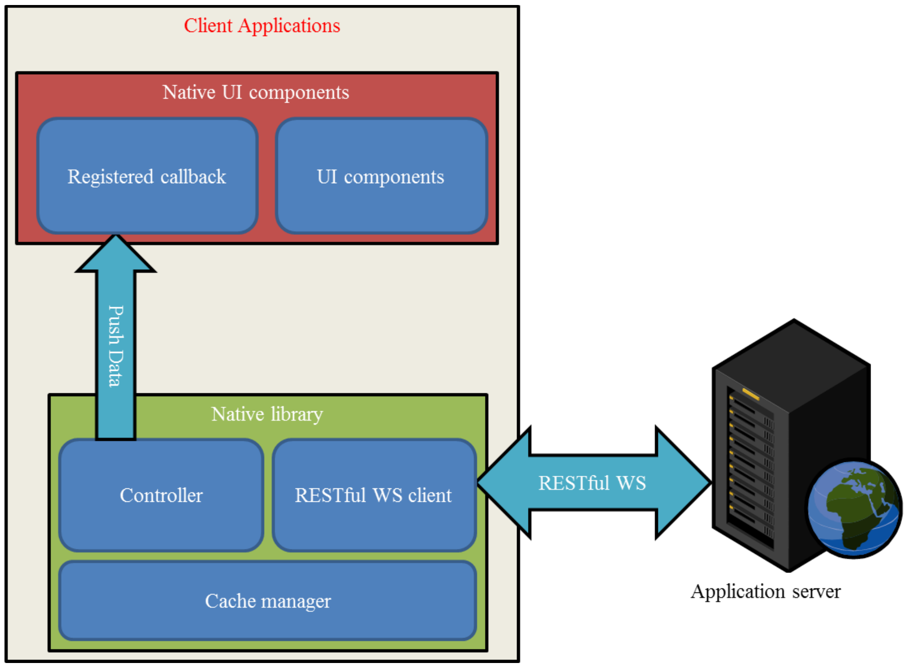


Figure 5.2 Native UI implementation

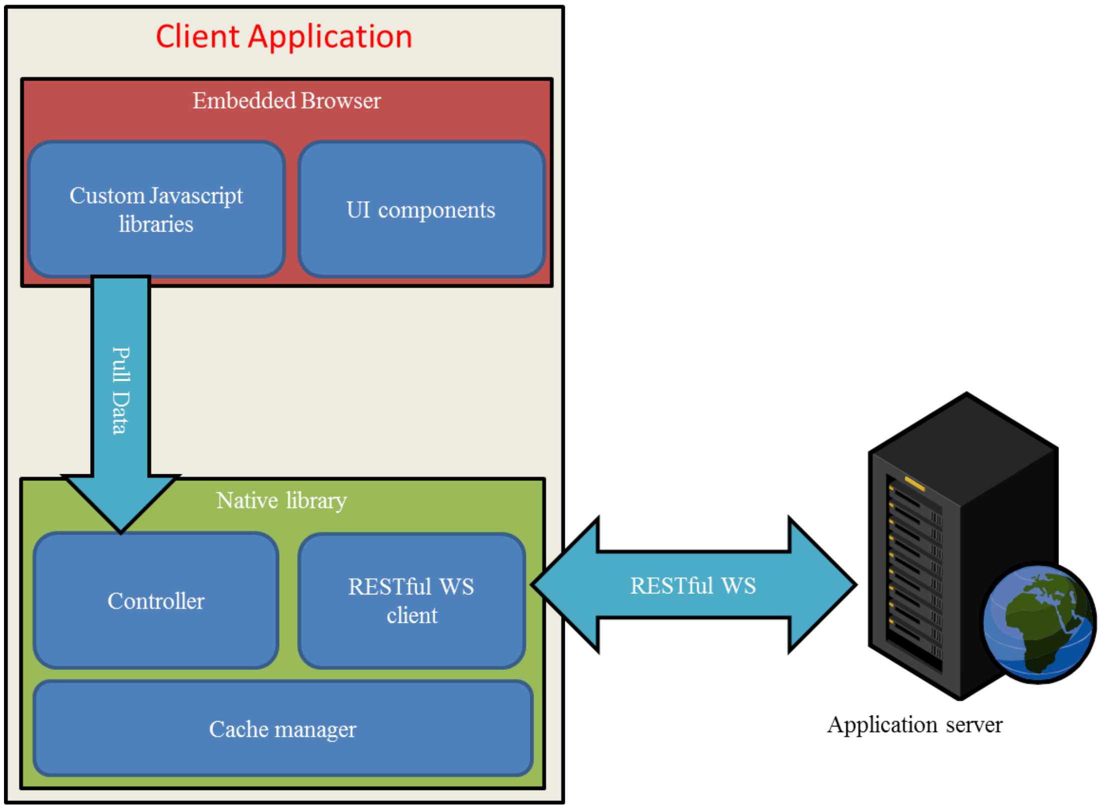
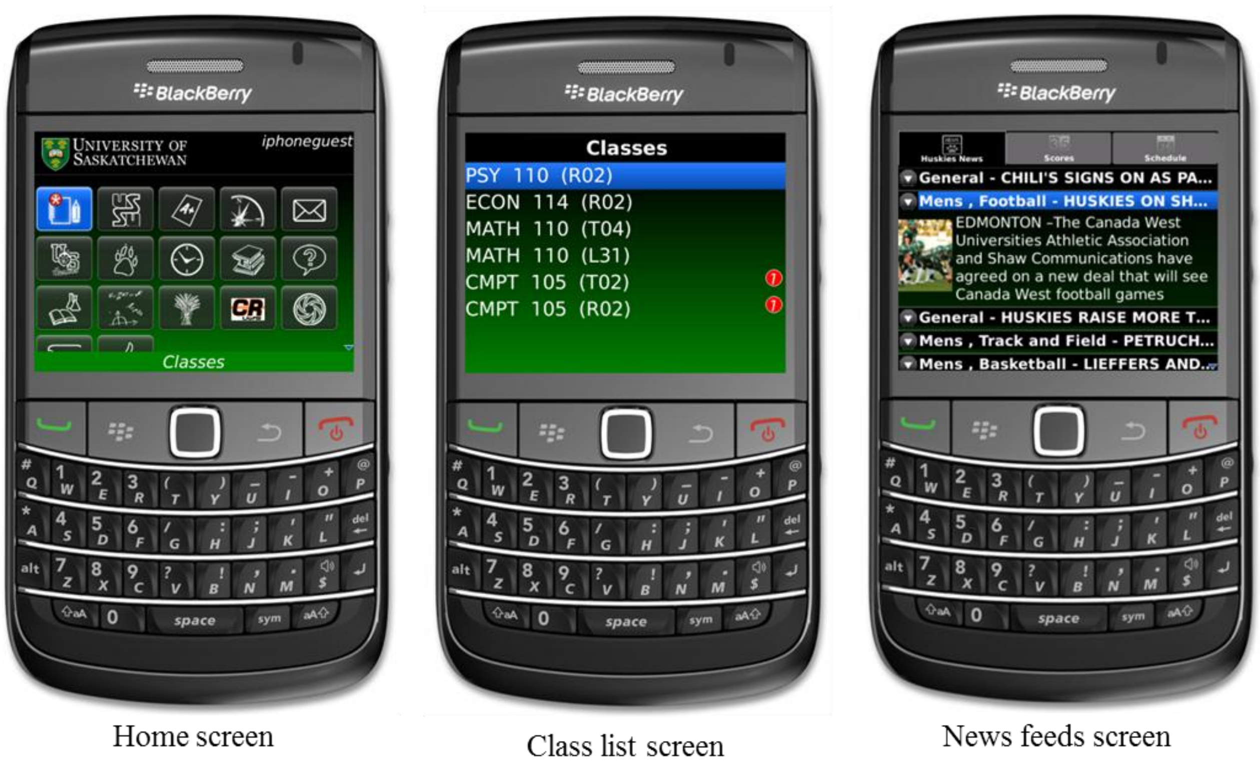


Figure 5.3 Embedded Browser implementation



Classes
PSY 110 (R02)
ECON 114 (R02)
MATH 110 (T04)
MATH 110 (L31)
CMPT 105 (T02)
CMPT 105 (R02)

General - CHILI'S SIGNS ON AS PA...
Mens , Football - HUSKIES ON SH...
EDMONTON -The Canada West Universities Athletic Association and Shaw Communications have agreed on a new deal that will see Canada West football games
General - HUSKIES RAISE MORE T...
Mens , Track and Field - PETRUCH...
Mens , Basketball - LIEFFERS AND...

Figure 5.4 Screenshots of Native UI

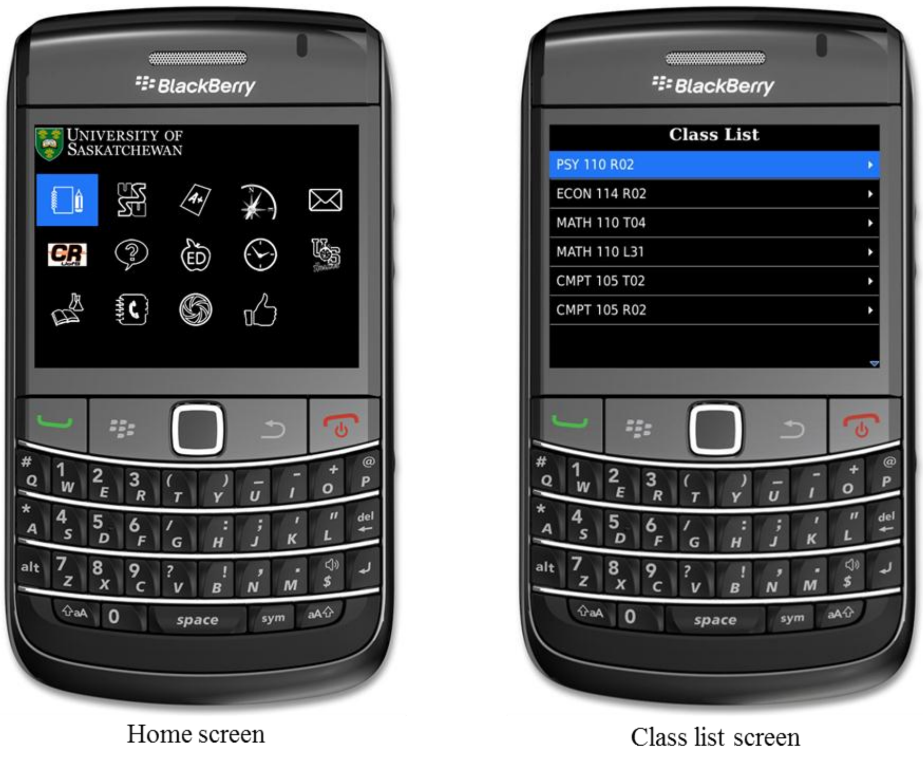


Figure 5.5 Screenshots of WebWork UI

5.2 *Cloud Middleware Implementations*

This section describes the two Cloud implementations of the middleware that can be deployed on GAE and EC2. Table 5.1 lists their key properties.

Properties	**EC2**	**GAE**
Focus	Infrastructure	Platform
Primary technology	Virtualization	Service-oriented Architecture
Service model	Virtual machine with OS image	Web application container
Service access interface	Command line Web Services	Command line
Auto-scale option	Elastic MapReducce	Billable option
Other bundled Services	Amazon S3 Amazon SimpleDB Amazon RDS Amazon SQS	Data store Memcache URL fetch Mail Task queue
Programming language	Any	Python Java
Charging model	Time & Resource	Resource

Table 5.1 EC2 and GAE Comparison

5.2.1 Amazon EC2 implementation

Following the standard pattern used by most Java developers, the middleware architecture is implemented as a Java Web application. The application exposes RESTful WS interfaces to mobile clients, since REST style WS are more suitable for mobile devices [17]. Because the middleware uses REST and Java servlet API, it has to be deployed on a Java HTTP server container, e.g. Glassfish (v3 Prelude). Glassfish uses Jersey library which is a Java RESTful API implementation (JSR-311). Since Glassfish has extensive IDE integration and uses non-blocking IO model a small footprint, it is used in many places, like GAE. The middleware also uses Apache HTTP client, a popular Java HTTP client library which provides functions of composing custom HTTP requests, sending and receiving HTTP requests and responses. The middleware expects the WS to return XML responses, so that results can be extracted using the Java build-in XPath library. The middleware uses a local MySQL database. User defined tasks, service actions, parameters and results are Java objects which map to database entities using the Java Persistent API (JPA). For evaluation purpose, the result values of each task are not cached. In this implementation, I use Oracle's JPA library, called TopLink.

Because EC2 is an IaaS, developers have full control of the system. They can choose deployment infrastructures (hardware, Operating System) and programming languages. Developers can also configure the system to increase the performance of certain applications, for example increasing application memory. The biggest advantage of EC2 is that developers are free to use any libraries. For example, many service providers offer a client library for their service, like Google, Yahoo, and Facebook. These libraries can be easily installed into the EC2 virtual machine.

One disadvantages of EC2 is maintenance and configuration. Although hardware is taken care of by Amazon, the VM still needs an IT administrator to monitor and backup. For example, failing to save the system image can cause serious data loss. Many enterprise level applications need be configured by experts in order to get the optimized performance. Another disadvantage is the resource utilization of the VM. The needs of resources vary from time to time and are hard to predict. There are also different needs for different resources. For example, an application may have good network access, but low computational power.

5.2.2 Google App Engine implementation

The middleware implementation on GAE is a small modification of the previous EC2 implementation. The middleware still has a RESTful interface to mobile clients, but the GAE platform itself is a Web application server which can only handle Java servlet requests. With the RESTlet 2.0 library, one of the first RESTful libraries supported by the GAE, the middleware can provide a RESTful interface through a servlet fa çade. E.g. all the requests go to fa çade servlet and then are mapped to different RESTful services. The Apache HTTP client library is not supported on the GAE, due to the restrictions from the provider. Instead the middleware constructs and sends HTTP requests through the URL fetch service which implements the Java.net interface. GAE provides reliable data store for storing predefined tasks, service actions and etc. It also supports Java Data Object (JDO) which is another API for Java object to database entity mapping.

Web applications on GAE follow the standard Java servlet API and most of Java Web developers are familiar with this environment. Hence, the GAE has very good community support. There is also a stable GAE development plug-in for Eclipse, a widely used open-source IDE, which accelerates the development. There is no cost for hardware maintenance and platform configuration. The Web application platform on GAE is pre-configured, and the GAE also takes care of system maintenance and updates. Compared to EC2, GAE requires cost to get the “free” quota. An application hosted on the GAE also guarantees high availability. Its quota counts the resource consumption, but does not limit how many resources an application can use, for example, CPU cycles consumed, number of requests processed and IO operations.

However, GAE is not very flexible since it uses a strict request-response model. E.g. any request that exceeds 30 seconds processing time will be dropped. The quota restricts its maximum scalability. For example the free quota states the maximum number of requests an application can receive is 7400 per minute. Besides the quota restrictions, GAE also has platform restrictions. The URL fetch service can only send requests to the standard port (80 or 443) at current time. For security reasons, many standard Java libraries are not available on GAE, for example, socket and threading. Because of that, many 3rd party libraries that based on these standard Java libraries cannot be run on GAE.

Chapter 6 EXPERIMENTS

6.1 Experiment Goals

The following is a list of experiments to evaluate the design of the middleware and the mobile client according to the research goals (section 2.3).

Goal 1. To enhance the interaction between mobile clients and Web Services

Experiment Goal 1.1. Evaluate the cross-platform capability of the mobile clients design.

Experiment Goal 1.2. Implement the mobile client in different models.

Experiment Goal 1.3. Consume RESTful WS through the middleware.

Experiment Goal 1.4. Transfers SOAP WS to RESTful WS to be consumed by mobile clients.

Experiment Goal 1.5. Reduce bandwidth consumption of mobile clients.

Experiment Goal 1.6. Push updates to mobile clients in real-time.

Goal 2. To use the Cloud platform as a way to improve scalability and reliability of the middleware

Experiment Goal 2.1. The middleware can be implemented on EC2 and GAE.

Experiment Goal 2.2. Cloud platform improves the scalability and reliability of the middleware.

Goal 3. To provide service mashup platform for mobile clients

Experiment Goal 3.1. Create and consume service mashup via the middleware on EC2.

The experiments evaluate the mobile client implementations, the middleware implementations on a laboratory server, and the Cloud implementations of the middleware. Table 6.1 lists the experiments name related to the above experiment goals.

Experiment Goals	**Experiments**
1.3 Mobile clients can consume RESTful WS through the middleware.	6.3 Consuming eBay WS through the Middleware
1.4 The middleware transfers SOAP WS to RESTful WS for mobile clients.	6.3 Consuming eBay WS through the Middleware
1.1 Evaluate the cross-platform capability of the mobile clients design.	6.4 Sending Service Request from the Mobile Client
1.1 Evaluate the cross-platform capability of the mobile clients design.	6.5 Native application vs. WebWork Application on Blackberry
1.5 The middleware reduces bandwidth consumption of mobile clients.	6.6 Bandwidth and Parsing Time Comparison of JSON and XML
1.6 The middleware push updates to mobile clients in real-time.	6.7 Receiving Updates with Push Technology
2.1 The middleware can be implemented on EC2 and GAE	6.8 Service Mashup through the Middleware Hosted on EC2
2.2 Cloud platform improves the scalability and reliability of the middleware.	6.9 Scalability of GAE and EC2
3.1 Create and consume service mashup via the middleware.	6.8 Service mashup through the middleware hosted on EC2

Table 6.1 List of Experiment Goals

6.2 Experiment Setup

The middleware is implemented as a standard Java Web Application. The middleware uses the Java EE5 standard, so it can be deployed in most Java server containers, for example, Glassfish, Jetty, and Jersey. The RESTful WS interface implements the Java EE6 standard (javax.ws.rs). The middleware also uses the Java Data Object (JDO) to interact with the MySQL Community Server 5.1 or Google's Big Table for GAE. In the following experiments, the middleware is deployed in three platforms, a laboratory server, an EC2 virtual machine and GAE. (See *Table 6.2* for hardware specifications) Because GAE uses Google's internal infrastructure, its hardware specification is not known. The laboratory server runs Windows 7 64-bit and the EC2 virtual machine runs Windows server 2003 Data center SP2 64-bit. The middleware is deployed in Glassfish V3 on both the laboratory server and the EC2 virtual machines. They share the same Glassfish configurations (see *table 6.3*). Because GAE uses their internal versions of Jetty, the middleware has to build the RESTful WS interface using RESTlet 2.0.

Instance name	Specification
Standard server	4 GB memory, 64-bit platform Intel® Core™2 Quad CPU Q9400 @ 2.66GHz 2.67GHz 500 GB storage (RAID 0) Intel® 82567 Gigabit Ethernet
EC2 instance (c1.medium)	1.7 GB memory, 32-bit platform 5 EC2 Compute Units (2 virtual core with 2.5 EC2 Compute Unit each) 350 GB instance storage (340 GB plus 10 GB root partition) I/O Performance: Moderate
EC2 instance (c1.xlarge)	7 GB memory, 64-bit platform 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each) 1,690 GB instance storage (4 x 420 GB plus 10 GB root partition) I/O Performance: High

Table 6.2 Specification of EC2 Instance

HTTP version	HTTP 1.1
JVM Memory	1024 MB
Auto reload applications	Disable
Monitor	Disable

Access logging		Disable
TCP configuration	Byte buffer type	Heap
	Buffer size	8192
	Acceptor threads	1
	Max connection count	4096
	Read timeout	30000
HTTP thread pool	Max Queue size	4096 (max 4096 number of threads in the queue)
	Max thread pool size	200
	Min thread pool size	2
	Idle thread timeout	900s

Table 6.3 Glassfish configurations

Because some experiments require simulating a large number of mobile clients and calculating the response times, a real mobile device is not capable of doing such task. A performance testing tool called Tsung [55] is used as a load generator. It is responsible for generating and sending HTTP requests to the middleware in a specified rate. Tsung calculates the mean of response times every 10 seconds based on its log file. The load generator runs on the standard server for the eBay experiment and an EC2 c1.medium instance for the Cloud experiment. (See *table 6.2* for hardware specifications)

The mobile client is implemented on two platforms, Android and Blackberry. The Android device used is HTC Android Developer Phone which runs Android 1.5. According to the HTC product website, the processor used in HTC ADP is Qualcomm® MSM7201A 528 MHz and the device has 256 MB of ROM and 192 of RAM. The build-in Apache HTTP client is used to send HTTP request. The Blackberry device used is Blackberry Bold 9700 which runs Blackberry OS 5.0. Blackberry Bold 9700 has 624MHz processor and 256MB RAM. Both of them are connected to the Internet through wireless 802.11g. The client uses the IO libraries from RIM and Java ME.

6.3 *Consuming eBay WS through the Middleware*

This experiment compares the overhead associated with different WS interactions. eBay provides both SOAP and RESTful WS interfaces for their Marketplace service. Their RESTful WS return result in either XML or JSON format. The tested WS is “FindItemsByKeywords”, which returns a list of items match the keywords. The maximum list size is 100 and the keyword used is “Android”. The middleware is run on the standard server.

The following is a segment of JSON and XML result of the tested WS.

Test JSON Result

```
{"findItemsByKeywordsResponse":[{"ack":["Success"],"version":["1.8.0"],"times
tamp":["2010-10-
14T15:34:19.554Z"],"searchResult":[{"@count":"100","item":[{"itemId":["260674
835892"],"title":["BLACK HTC ANDROID G1 GOOGLE PHONE UNLOCKED GPS
"],"globalId":["EBAY-
ENCA"],"primaryCategory":[{"categoryId":["3312"],"categoryName":["Cell Phones
&
Smartphones"]}],"galleryURL":["http:\/\/thumbs1.ebaystatic.com\/pict\/2606748
358928080_1.jpg"],"viewItemURL":["http:\/\/cgi.ebay.ca\/BLACK-HTC-ANDROID-G1-
GOOGLE-PHONE-UNLOCKED-GPS-
\/260674835892?pt=Cell_Phones"],"productId":[{"@type":"ReferenceID","__value_
_":"82009038"}],"paymentMethod":["PayPal"],"autoPay":["false"],"postalCode":[
"L3T3H1"],"location":["Thornhill,Ontario,Canada"],"country":["CA"],"shippingI
nfo":[{"shippingServiceCost":[{"@currencyId":"CAD","__value__":"20.03"}],"shi
ppingType":["Flat"],"shipToLocations":["Worldwide"]}],"sellingStatus":[{"curr
entPrice":[{"@currencyId":"USD","__value__":"107.5"}],"convertedCurrentPrice"
:[{"@currencyId":"CAD","__value__":"107.74"}],"bidCount":["27"],"sellingState
":["Active"],"timeLeft":["P0DT4H40M25S"]}],"listingInfo":[{"bestOfferEnabled"
:["false"],"buyItNowAvailable":["false"],"startTime":["2010-10-
07T20:14:44.000Z"],"endTime":["2010-10-
14T20:14:44.000Z"],"listingType":["Auction"],"gift":["false"]}],"condition":[
{"conditionId":["3000"],"conditionDisplayName":["Used"]}]}
...
```

Test XML Result

```
<findItemsByKeywordsResponse
xmlns="http://www.ebay.com/marketplace/search/v1/services"><ack>Success</ack>
<version>1.8.0</version><timestamp>2010-10-
14T15:38:24.515Z</timestamp><searchResult
```

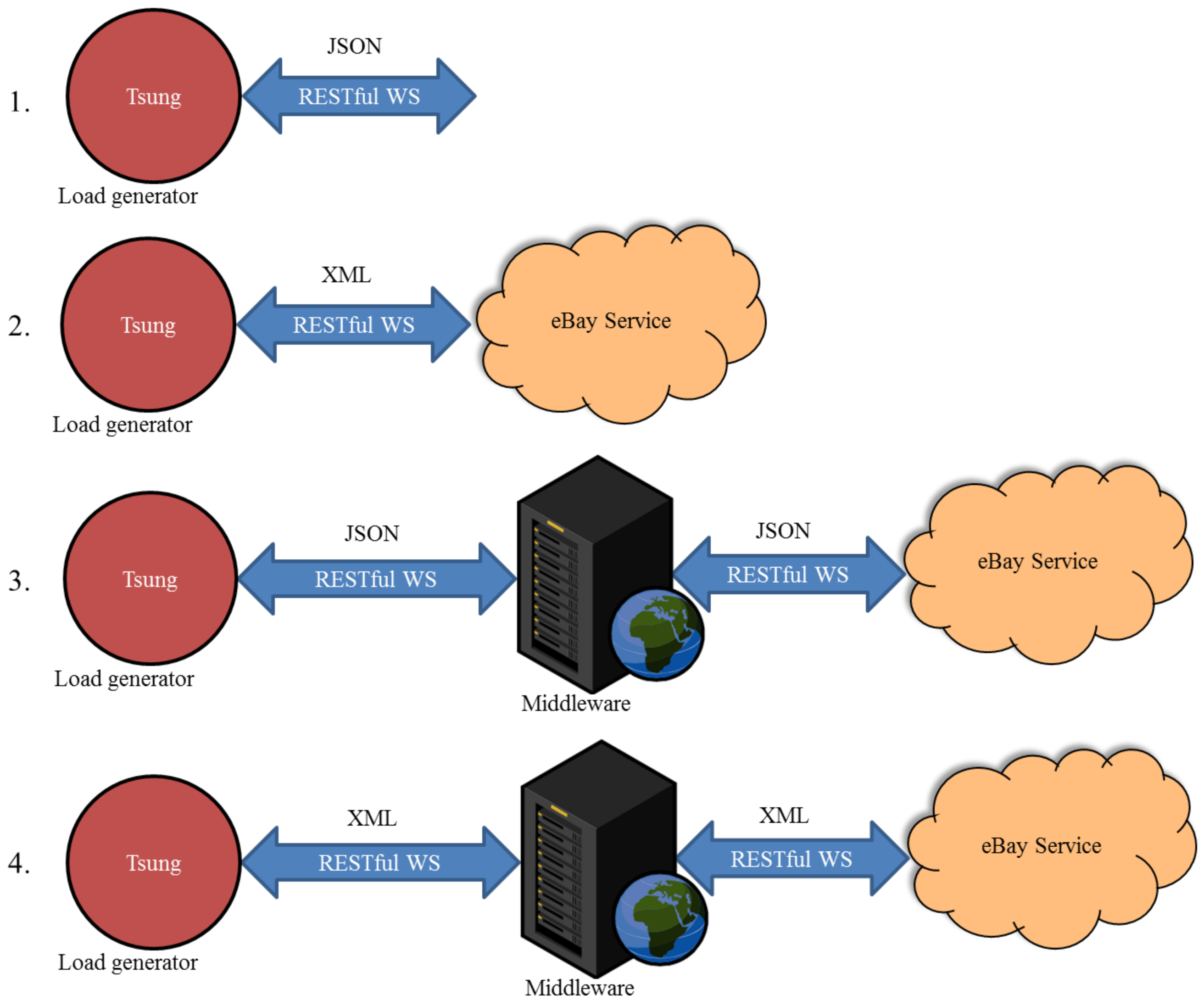
```
count="100"><item><itemId>260674835892</itemId><title>BLACK HTC ANDROID G1
GOOGLE PHONE UNLOCKED GPS </title><globalId>EBAY-
ENCA</globalId><primaryCategory><categoryId>3312</categoryId><categoryName>Ce
ll Phones &amp;
Smartphones</categoryName></primaryCategory><galleryURL>http://thumbs1.ebayst
atic.com/pict/2606748358928080_1.jpg</galleryURL><viewItemURL>http://cgi.ebay
.ca/BLACK-HTC-ANDROID-G1-GOOGLE-PHONE-UNLOCKED-GPS-
/260674835892?pt=Cell_Phones</viewItemURL><productId
type="ReferenceID">82009038</productId><paymentMethod>PayPal</paymentMethod><
autoPay>false</autoPay><postalCode>L3T3H1</postalCode><location>Thornhill,Ont
ario,Canada</location><country>CA</country><shippingInfo><shippingServiceCost
currencyId="CAD">20.03</shippingServiceCost><shippingType>Flat</shippingType>
<shipToLocations>Worldwide</shipToLocations></shippingInfo><sellingStatus><cu
rrentPrice currencyId="USD">107.5</currentPrice><convertedCurrentPrice
currencyId="CAD">107.74</convertedCurrentPrice><bidCount>27</bidCount><sellin
gState>Active</sellingState><timeLeft>P0DT4H36M20S</timeLeft></sellingStatus>
<listingInfo><bestOfferEnabled>false</bestOfferEnabled><buyItNowAvailable>fal
se</buyItNowAvailable><startTime>2010-10-
07T20:14:44.000Z</startTime><endTime>2010-10-
14T20:14:44.000Z</endTime><listingType>Auction</listingType><gift>false</gift
></listingInfo><condition><conditionId>3000</conditionId><conditionDisplayNam
e>Used</conditionDisplayName></condition></item>
…
```

The size of the JSON result is about 114 KB and the size of the XML result is about 140 KB.

The load generator sends HTTP request at the rate of 1 request per 10 second (exponential distribution, mean 0.1request/s), so the middleware does not overload. The duration is 10 minutes. The following experiments are conducted. (See *figure 6.1*)

1. Consume eBay RESTful WS directly with JSON result.
2. Consume eBay RESTful WS directly with XML result.
3. Consume eBay RESTful WS through the middleware with JSON result. The middleware forwards the complete result. (no parsing involved)
4. Consume eBay RESTful WS through the middleware with XML result. The middleware forwards the complete result. (no parsing involved)
5. Consume eBay RESTful WS through the middleware with JSON result. The middleware returns the optimized result in JSON format.
6. Consume eBay RESTful WS through the middleware with XML result. The middleware returns the optimized result in JSON format.

7. Consume eBay SOAP WS through the middleware.



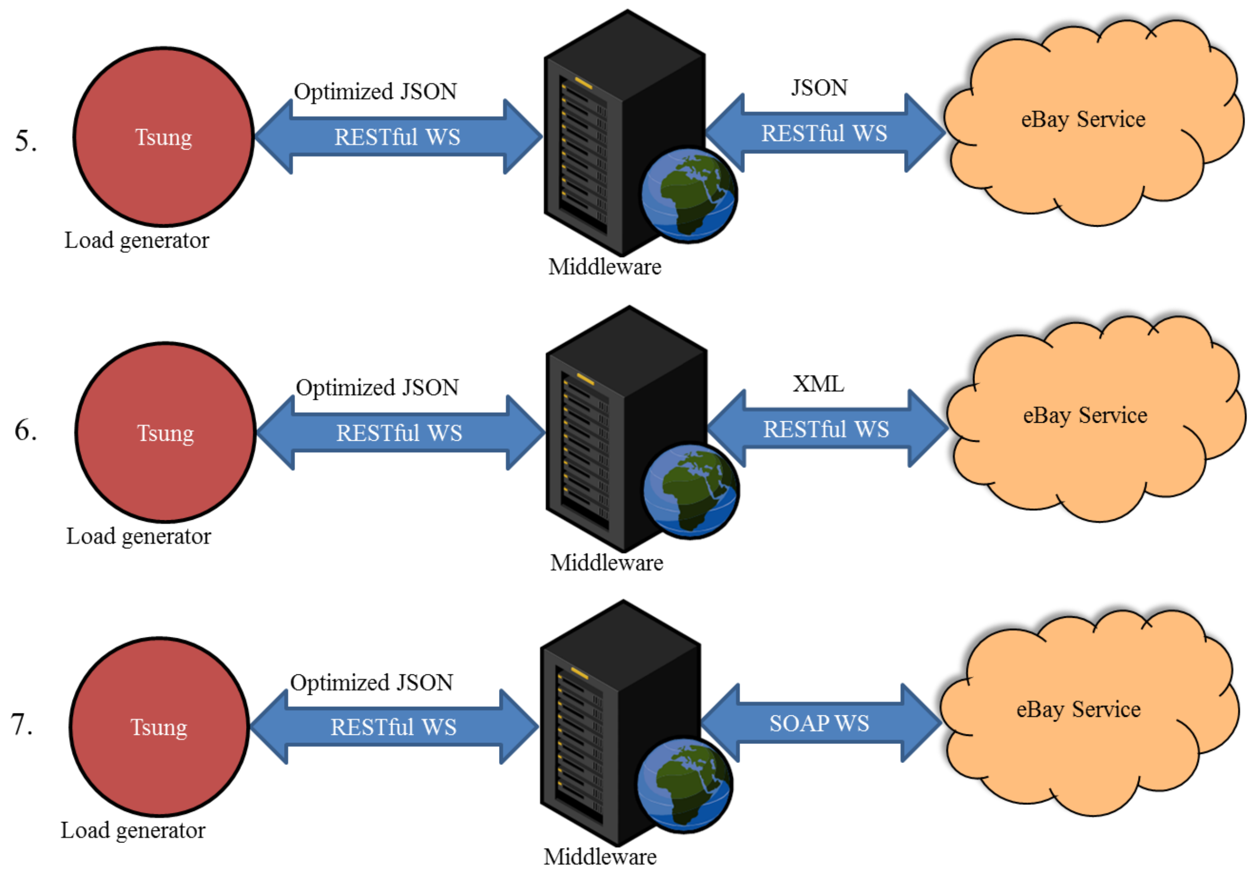


Figure 6.1Consume eBay WS Experiments

The middleware optimizes the result by extracting only the data required by the mobile client. Assume that the mobile client is only interested in the title of all list items, the size of the optimized result is 5.44 KB. The following is a segment of optimized the result in JSON.

```
["BLACK HTC ANDROID G1 GOOGLE PHONE UNLOCKED GPS ","HTC A3333 Wildfire Red
Android Quadband Unlocked Phone","HTC HD2 16gb  ( SIM Unlocked ), Android
Capable", "NEW Battery For GOOGLE G1 Android HTC TMOBILE PHONE E13",
…]
```

Table 6.3 shows the highest, low, and overall mean of the response time. *Figure 6.2* shows a bar graph comparing the response times of different interactions. There is overhead associated with the middleware. However, result optimization significantly reduces the bandwidth.

Experiment name	Description	Lowest (s)	Average (s)	Highest (s)
1. JSON direct	Consume eBay RESTful WS directly with JSON result.	0.50	0.95	1.75
2. JSON	Consume eBay RESTful WS through the	0.47	1.14	2.42

middleware	middleware with JSON result. The middleware forwards the complete result. (no parsing involved)			
3. JSON middleware optimized	Consume eBay RESTful WS through the middleware with JSON result. The middleware returns the optimized result in JSON format.	0.55	1.13	2.83
4. XML direct	Consume eBay RESTful WS directly with XML result.	0.55	1.09	3.25
5. XML middleware	Consume eBay RESTful WS through the middleware with XML result. The middleware forwards the complete result. (no parsing involved)	0.60	1.21	5.08
6. XML middleware optimized	Consume eBay RESTful WS through the middleware with XML result. The middleware returns the optimized result in JSON format.	0.55	1.39	5.07
7. SOAP middleware	Consume eBay SOAP WS through the middleware.	1.11	2.94	5.24

Table 6.3 Response Time of Consuming eBay WS

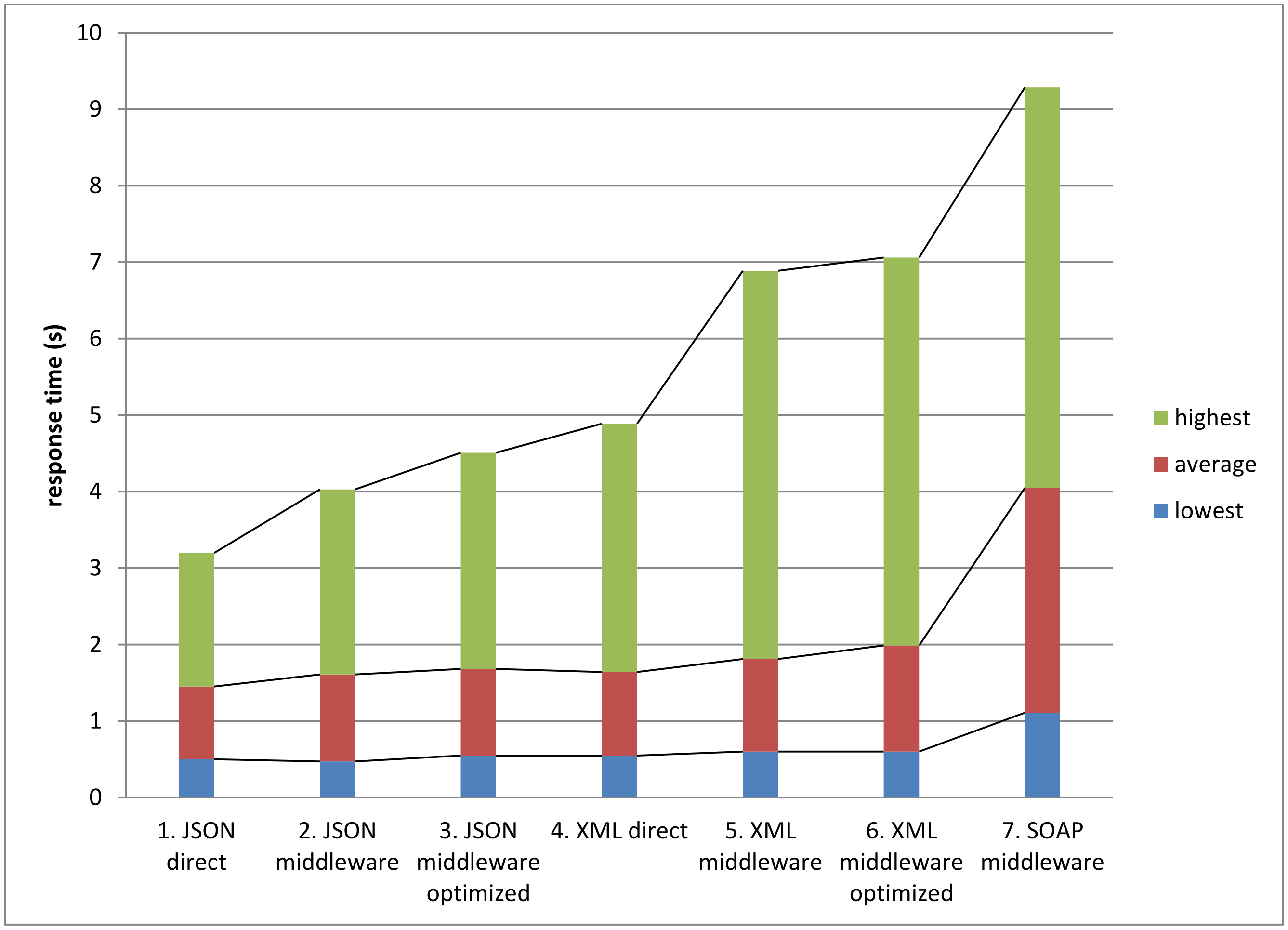


Figure 6.2 Bar Graph of Response Time

- Direct vs. middleware: Compare the experiment 1 and 2, 4 and 5, whether the eBay services return JSON or XML, the middleware adds a little overhead (on average 0.19s with JSON and 0.12s with XML) on the response time. Because the middleware does not do any processing of the service results, the overhead is mainly caused by network latency between the client and middleware.
- JSON vs. XML: Compare the JSON experiment (1, 2, 3) and XML experiment (4, 5, 6), interactions utilized by JSON have less response time than XML. It is because the verbose XML messages are large which causes network transmission delay.

- Optimized vs. non-optimized: Compare the results of experiment 2 and 3, result optimization with JSON reduces the response time a little (0.1s on average). Compare the results of experiment 4 and 5, result optimization with XML adds a little overhead (0.18s on average). The middleware adds overhead with parsing and extracting data from the original result. However, the result optimization reduces the amount of data transferred (reduced from 114 JSON, 140 XML to 5.44KB), which reduces response time. This also implies that parsing XML is slower than parsing JSON.
- RESTful vs. SOAP: As the experiment 7 indicated, SOPA WS has higher response times than the rest of the experiment with RESTful WS. SOAP is verbose protocol which means more data needs to be transferred. In addition, processing time required for the middleware creating Java object from the SOAP message. The advantage of SOAP WS is easy access of the results, because the results are represented as a Java object.

6.4 Sending Service Request from the Mobile Client

To prove the mobile client design is valid and platform independent, this experiment implemented the mobile client on both Android and Blackberry platform which can send RESTful WS requests to the middleware. To understand the platform limitations, this experiment measures the maximum request rate which is defined as the fastest speed the mobile clients can send RESTful WS requests at. In addition, knowing the maximum request rate of the mobile clients, one can estimate how much load will be on the middleware.

On both Android and Blackberry, the testing applications are native applications which have a HTTP client to send a GET request. The GET request retrieves Google search page. The tested mobile client generates and sends 20 GET requests in a closed loop (sequentially) and records the time for the whole process. (See *figure 6.3*) The sending of HTTP requests is synchronized. E.g. Request is sent after the pervious response arrives. If the device can send N requests in T time, then the

Maximum request sending rate on ADP = N / T

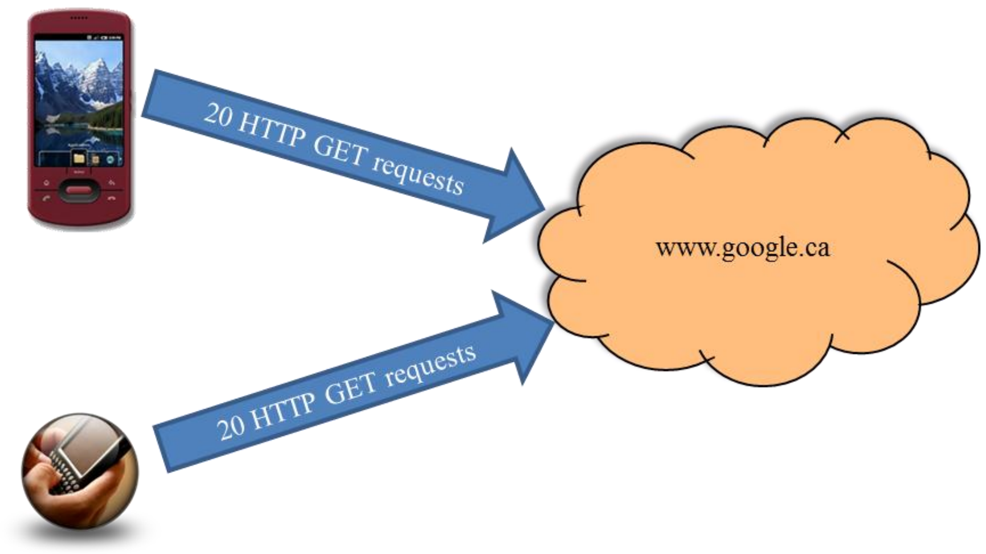


Figure 6.3 Max request rate experiment

I ran the experiment 20 times. *Table 6.4* lists the statistical results.

Device	Average time for sending 20 requests	Standard deviation	Maximum request rate
HTC ADP	28174.6ms	4875.8	0.71request/s
Blackberry Bold 9700	210643.1ms	238.55	0.095request/s

Table 6.4 Result of Max Request Rate

The standard deviation (σ) shows the variance within samples and is calculated using the following formula where *x* is each simple value, N is the sample size, μ is mean value.

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

The result shows that the HTC ADP is able to send HTTP requests almost 10 times faster than Blackberry Bold 9700. Blackberry OS 5.0 has 6 network transport types: Blackberry Internet Service, Mobile Data Service, Direct TCP, WIFI TCP, WAP 1.0, WAP 2.0. When send a HTTP request, the OS attempts obtain the first available connection from them. If the attempt fails, the OS wait for a certain time and gives another try. This of course slows the rate of sending HTTP requests on the Blackberry. Android makes HTTP request faster by using the Apache HTTP client which uses directly HTTP connection via TCP. However, the client is not aware of what kind of transport type the connection uses.

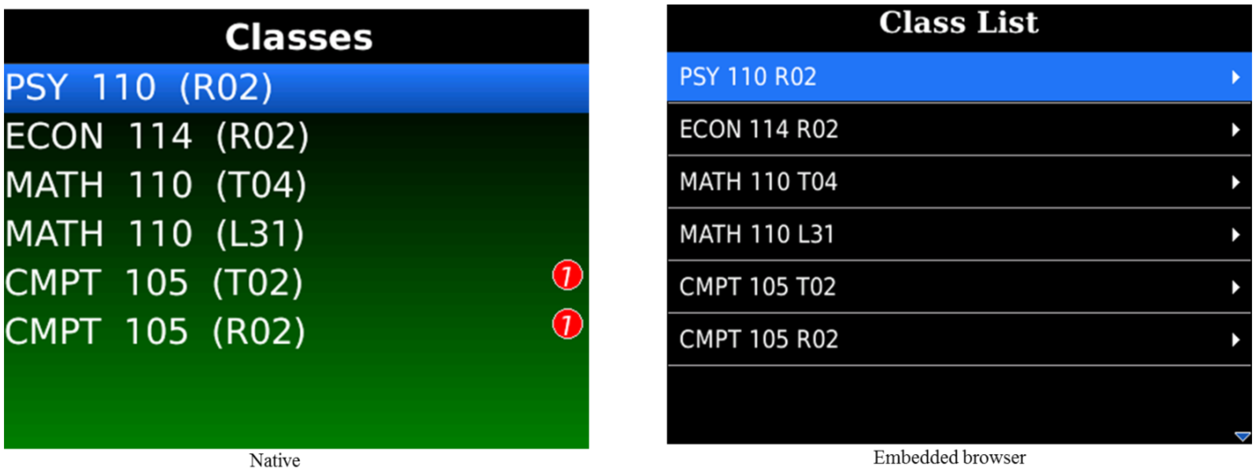
6.5 Native vs. WebWork Application

The experiment examines the performance difference of native and WebWork (an embedded browser framework on Blackberry OS 5.0) application to understand the advantages and disadvantages of both. The process includes interacting with middleware to get data and rendering the data to the screen.

Both native and WebWork application have client to interact with the middleware. The native application uses the native HTTP connection API from Java library, while the WebWork application uses the XMLHttpRequest from the WebWork JavaScript framework. The client sends a HTTP GET request and the middleware returns the following data describing a list of classes. Below is the transferred data is in JSON format.

```
{"taking":[
{"subject":"PSY","coursenum":"110","section":"R02","term":"201001","crn":"25882"},
{"subject":"ECON","coursenum":"114","section":"R02","term":"201001","crn":"25889"},
{"subject":"MATH","coursenum":"110","section":"T04","term":"201001","crn":"21477"},
{"subject":"MATH","coursenum":"110","section":"L31","term":"201001","crn":"26665"}],
"instructing":[{"subject":"CMPT","coursenum":"105","section":"T02","term":"201001","crn":"2
7795"}],
"assisting":[{"subject":"CMPT","coursenum":"105","section":"R02","term":"201001","crn":"277
94"}] }
```

The native application renders the class list on the screen with native UI components and the WebWork application renders it with HTML and CSS. *Figure 6.4* shows the rendered class list screen.


Classes

PSY 110 (R02)
ECON 114 (R02)
MATH 110 (T04)
MATH 110 (L31)
CMPT 105 (T02)
CMPT 105 (R02)

Class List

PSY 110 R02
ECON 114 R02
MATH 110 T04
MATH 110 L31
CMPT 105 T02
CMPT 105 R02

Native

Embedded browser

Figure 6.4 Class list screen

I did 50 sample runs of both native and WebWork application and recorded the total time (HTTP interaction + JSON parsing and UI rendering) it takes to reach the class list screen. The network used is wireless 802.11g through university routers. The total time taken is 0.855s on average (SED = 0.366) for the native application and 0.476s (SED = 0.213) for the WebWork application. The results shows that the WebWork application is about two times faster compared to the native application. The major time consuming task for native application is HTTP interaction (about 0.566s in average), because of the Blackberry HTTP connection API. In the WebWork application, HTTP connections are obtained from the browser, which are initialized at application start. However, the WebWork application can only obtain 2 HTTP connections concurrently and they cannot send cross-domain HTTP requests. Native application does not have the above restrictions and offer rich UI components.

6.6 Bandwidth and Parsing Time Comparison of JSON and XML

JSON and XML are two widely used formats for transferring WS message. Many Cloud Services offer a selection to choose one of them. Since mobile clients have limited processing power and bandwidth, this experiment evaluates the use of JSON and XML.

The experiment 6.3 indicated that for the same content, the XML message is larger and thus slower in terms of response time compare to JSON message. To further prove that, I use a Twitter WS which returns the 20 most recent tweets (updates) in both XML and JSON format. The mobile client parses the XML result with standard Java DOM parser and JSON result with parser from www.json.org.

Mobile Platform	Format	Message size (KB)	Average parsing time (ms)	Standard deviation
Android	XML	46.8	321.7	25.33
	JSON	29.1	40.0	6.33
Blackberry	XML	46.8	587.8	9.11
	JSON	29.1	248.6	8.67

Table 6.5 Size and parsing time of JSON and XML message

Table 6.5 shows the average parsing times and their standard deviations for the XML and JSON message over 20 independent samples on Android and Blackberry. First, comparing the size, the size of XML result is 46.8KB and 29.1KB for JSON. To represent the same information, the XML format requires more bandwidth. Second, considering the parsing time, parsing XML message is more resource consuming than parsing JSON message on both Android and Blackberry. The slowness is not only due to the size, but also the complexity of parsing. Finally, JSON format also has very stable parsing time. However, it is very difficult to represent complex data structure in JSON format.

6.7 Receiving Updates with Push Technology

There are two approaches to request data from a server, pulling and pushing. (See *figure 6.5*) Pull means that clients initiate a request to obtain resources on the server. The server then returns the requested data. Push means the server initiates the interactions. The server knows each client and what resources it needs. The server sends updates to clients whenever the resources are changed.

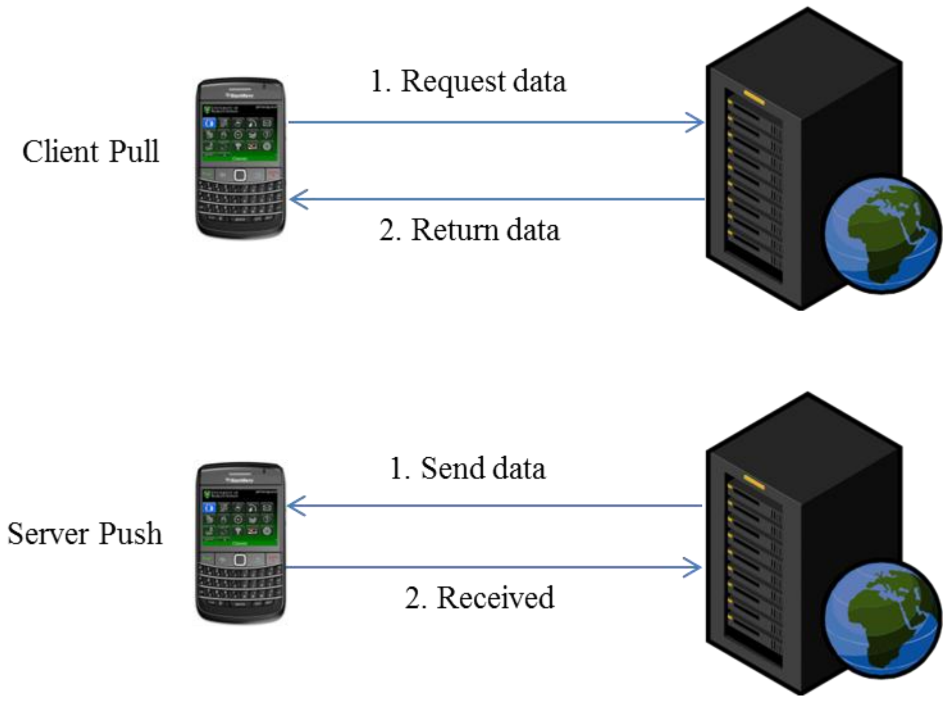


Figure 6.5 Pull and Push

Pull is a commonly used pattern for client-server interaction, for example browsers and Web pages. When clients need real-time updates, the clients initiate constant pulling (e.g. pull every 2 seconds). A constant pull wastes a lot of energy and bandwidth with sending requests and receiving duplicated data. Push is more efficient in terms of bandwidth and energy, since only the updates are sent to the clients and only when the resources are changed on the server.

In the experiment, I use e-mail as the push method and compare it to the standard HTTP pull method on Blackberry. The server keeps a list of news which updates every 30 seconds for 30 minutes. Updates have a constant size of 530 bytes. For the standard HTTP pull, the mobile client pulls from the middleware every 10 seconds to keep the news list updated. If no change on the list, the server returns no content. For e-mail push, the server pushes the change via the email account setup on the mobile client when the news list changes. (See *figure 6.6*)

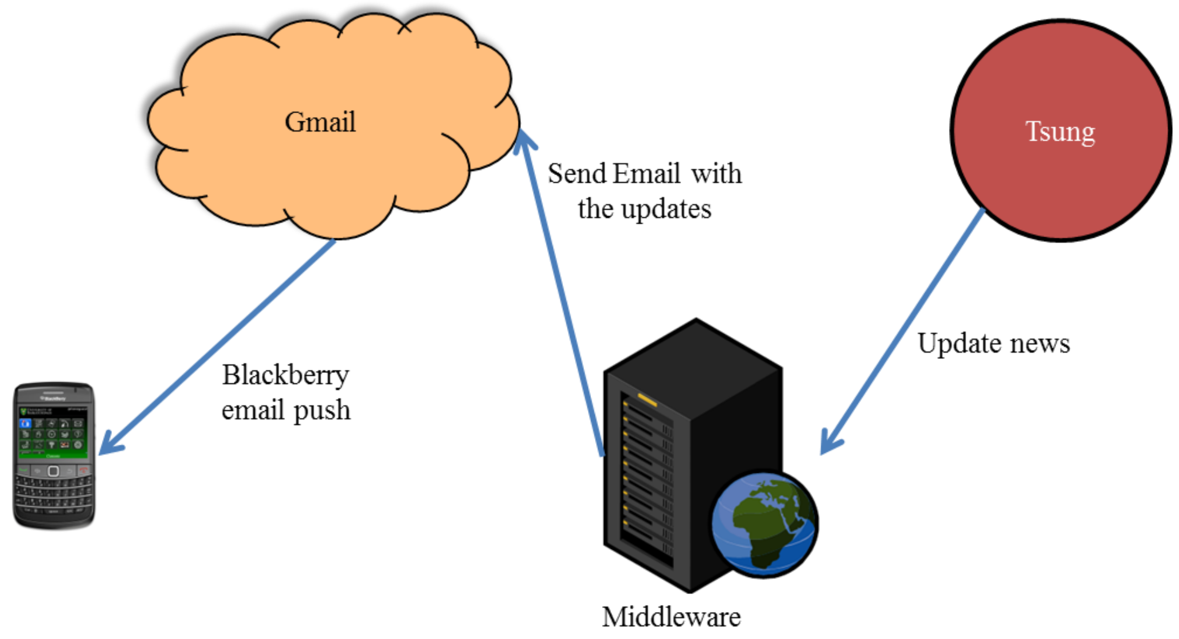


Figure 6.6 Blackberry Email Push

To evaluate push and pull, I measure the following values in my experiment.

1. Bandwidth used: The total data transferred for the mobile client during the experiment includes upload and download.
2. Energy consumption: Network interaction like HTTP and SMTP consumes energy. The more network interaction is involved, the more energy is consumed.
3. Update elapse time: Time when the update received on the middleware – time when the update reaches the mobile client.

Interaction	Bandwidth used (KB)		Energy consumption (number of request sent)
	Upload	Download	
Pulling	29.88	48.18	180 HTTP GET requests
Pushing	unknown	31.98	60 email (SMTP)

Table 6.6 Pull vs. Push

Table 6.6 shows the bandwidth used and Energy consumption of the mobile client during the 30 minutes. *Figure 6.7* indicates the update elapse times of total 60 update.

1. Bandwidth used: For the pulling experiment, the client sends 29.88KB and receives 48.18KB data in total. For the pushing experiment, each email message is 533byte, thus 31.98KB in total of 60 emails is downloaded. However, the upload amount is unknown, because how the Blackberry email push interacts with Gmail server is not revealed. The bandwidth difference is caused by the message headers of different protocol (HTTP and SMTP).
2. Energy consumption: For the pulling experiment, the client sends 180 HTTP GET request and receives 180 responses in total. While only 60 email message are received via SMTP for the push experiment. The pulling experiment consumes more energy. Noted that the energy consumption can be reduced by increasing the pulling interval. However, less frequent pulling increases the update elapse time.

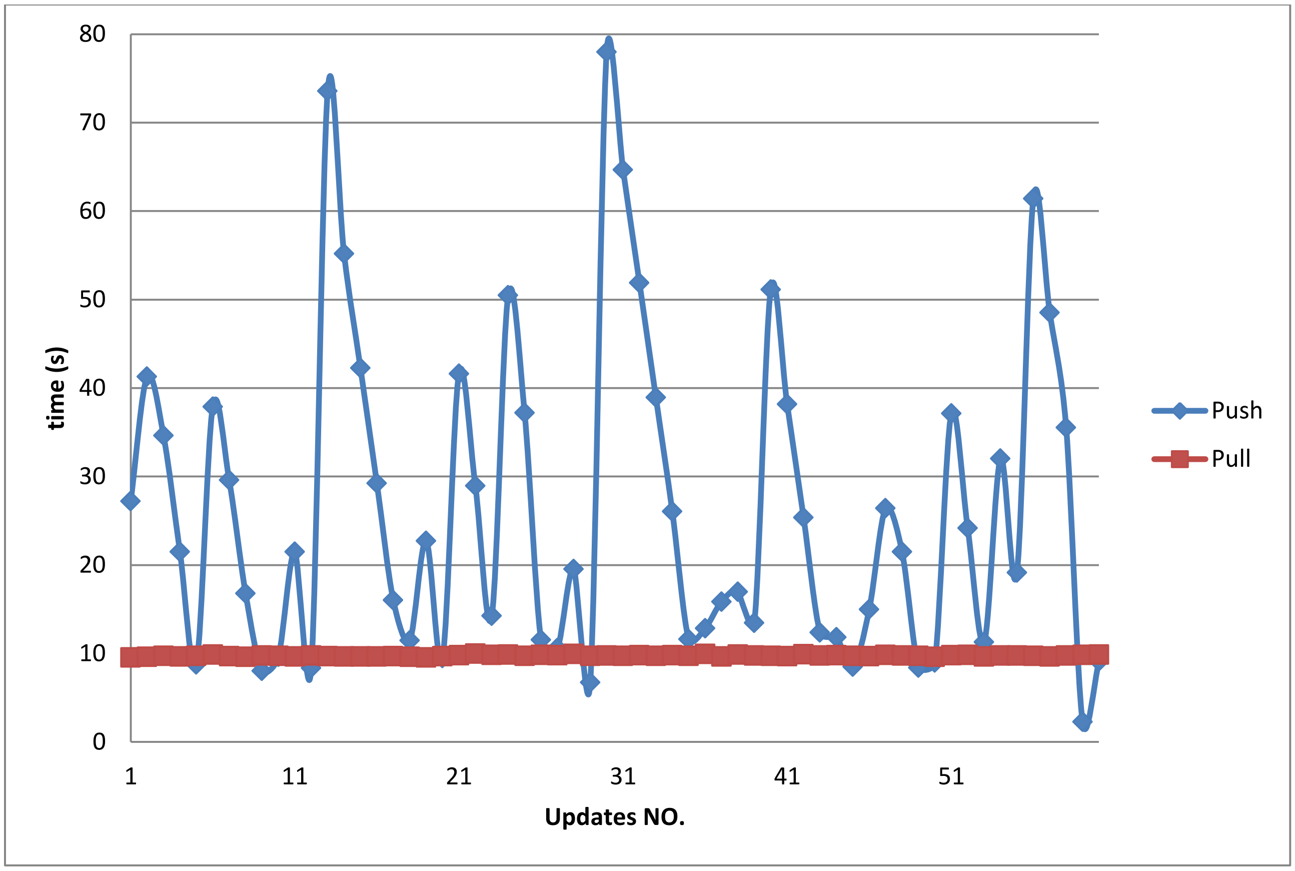


Figure 6.7 Update time of pulling and pushing

3. Update elapse time (see *figure 6.7*): For the pulling experiment, the update elapse time is almost constant (mean 9747, SED 91), because the middleware receives update at a constant rate, the time difference between each client pulls and new update in the middleware is also constant. However, it is very unlikely the update happens at a constant rate in real case. The rate of pulling needs be adjusted according to the time distribution of update. For the pushing experiment, the update elapse time fluctuates a lot (mean 26404, SED 17867), because several messages are batched into one push message. Each peak is when the push message arrived. It is a more energy efficient way of transfer data, since it requires less network interactions.

6.8 Mashup Service through the Middleware Hosted on EC2

To evaluate the middleware implementation on EC2, this experiment creates and consumes a mashup services through the middleware hosted on EC2 comparing with direct consuming the mashup service on the client side. (See *figure 6.8*) The mashup service combines two Yahoo Upcoming services. (See *table 6.7*)

API name	Input	Output
Find city	Geo-coordinates	Nearest city information
Find events	City name	List of events in the city

Table 6.7 Yahoo Upcoming Services

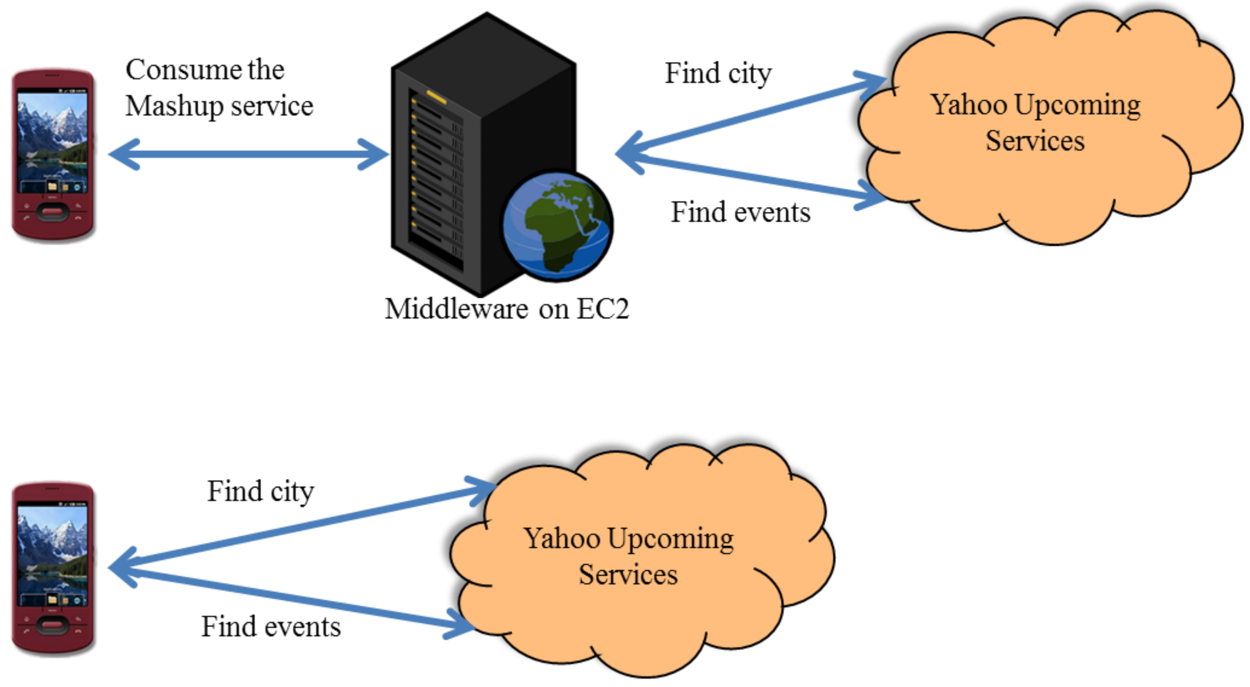


Figure 6.8 Consume mashup service

The mashup service finds all events in the nearest city according to the user location. The middleware is deployed on an EC2 m1.small instance (see *table 6.2* for hardware specification). The client is on Android Developer Device. First, the client sends HTTP POST requests and predefines the mashup service. Note that the IDs are randomly generated UUID. The following is the HTTP POST request to define the “find city” and the “find events” Service Action. Note that the location input of the “find events” Service Action is piped from the result of “find city” Service Action.

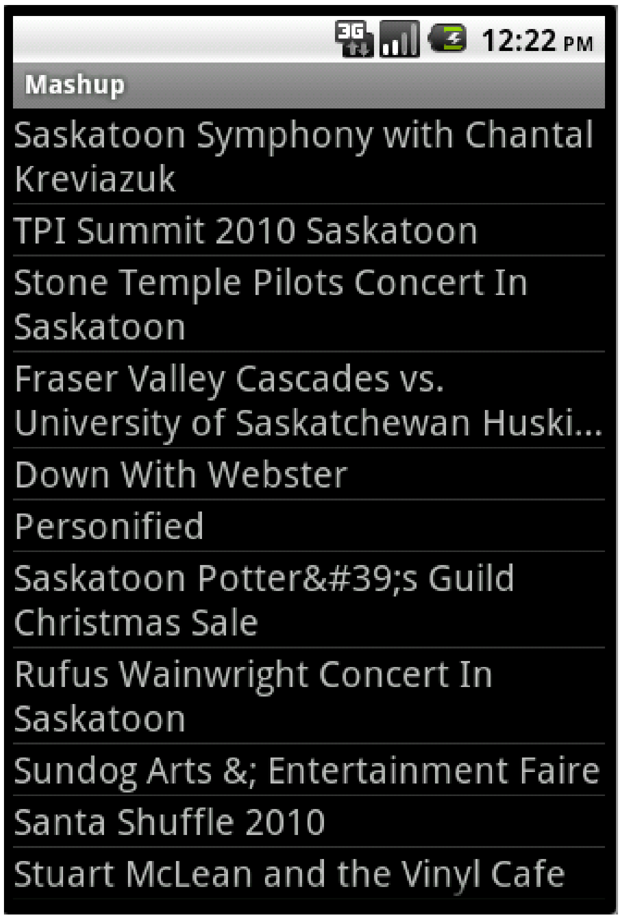
Define the “find city” Service Action (HTTP POST):
```
POST         /madmuc_servicedesktop/resources/services/actions?serviceID=…
HTTP/1.1
Host: ec2-174-129-160-220.compute-1.amazonaws.com
Content-Type: application/json

{"conType":"application\/xml","actionDis":"get                          city
info","httpMethod":"GET","outParam":[{"resultName":"rsp.metro.name","forma
t":"text"],"proType":"application\/xml","actionURL":"","context":false,"ac
tionName":"GetForCoord","inParam":[{"value":"********","embedMethod":"quer
y","paramName":"api_key","isPipped":false},{"value":"52.1333","embedMethod
":"query","paramName":"latitude","isPipped":false},{"value":"-
106.666","embedMethod":"query","paramName":"longitude","isPipped":false}]}
```

Define the “find events” Service Action (HTTP POST):
```
POST         /madmuc_servicedesktop/resources/services/actions?serviceID=…
HTTP/1.1
Host: ec2-174-129-160-220.compute-1.amazonaws.com
Content-Type: application/json

{"conType":"application\/xml","actionDis":"get   events   in   the   nearest
city","httpMethod":"GET","outParam":[{"resultName":"rsp.event.name","forma
t":"text"],"proType":"application\/xml","actionURL":"","context":false,"ac
tionName":"GetEvents","inParam":[{"value":"********","embedMethod":"query"
,"paramName":"api_key","isPipped":false},{"value":"GetForCoord/rsp.metro.n
ame","embedMethod":"query","paramName":"location","isPipped":true}]}
```

Then, the client sends a HTTP GET request to execute the mashup service. The result is represented as a list shown in *figure 6.9*. The following is the HTTP request and response.

Consume the Mashup Service (HTTP GET):
```
GET /madmuc_servicedesktop/resources/mashup?actionID=… HTTP/1.1
Host: ec2-174-129-160-220.compute-1.amazonaws.com
Accept: application/json
```

Mashup Result (HTTP Response):
```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Lenght: application/json


{"rsp.event.name":[{"value":[  Saskatoon  Symphony  with  Chantal  Kreviazuk,
TPI  Summit  2010  Saskatoon,  Stone  Temple  Pilots  Concert  In  Saskatoon,
Fraser  Valley  Cascades  vs.  University  of  Saskatchewan  Huskies,  Down  With
Webster, Personified" … ],"format":"text"}]}
```



Saskatoon Symphony with Chantal Kreviazuk
TPI Summit 2010 Saskatoon
Stone Temple Pilots Concert In Saskatoon
Fraser Valley Cascades vs. University of Saskatchewan Huski...
Down With Webster
Personified
Saskatoon Potter&#39;s Guild Christmas Sale
Rufus Wainwright Concert In Saskatoon
Sundog Arts &; Entertainment Faire
Santa Shuffle 2010
Stuart McLean and the Vinyl Cafe

Figure 6.9 Events list

Figure 6.10 shows the overhead of consuming the mashup service on the middleware (middleware series) versus directly consuming and combining the two services on the mobile client (client series). The x-axis is the number of executions of the mashup service (50 samples in total). The time interval between each sample request is 1 minutes, so they do not cause a heavy load on the middleware. The y-axis is the total processing time including network latency and parsing time. The average response time of the middleware mashup is 753.48ms with a standard deviation of 99.5. The average response time of client side mashup is 942.22ms with standard deviation of 97.7. Both of the two series have a lot of fluctuations which is mainly caused by network latency. The result shows executing the mashup on the middleware is faster than executing it on the client, because the middleware has access to more bandwidth, network connections and processing power.

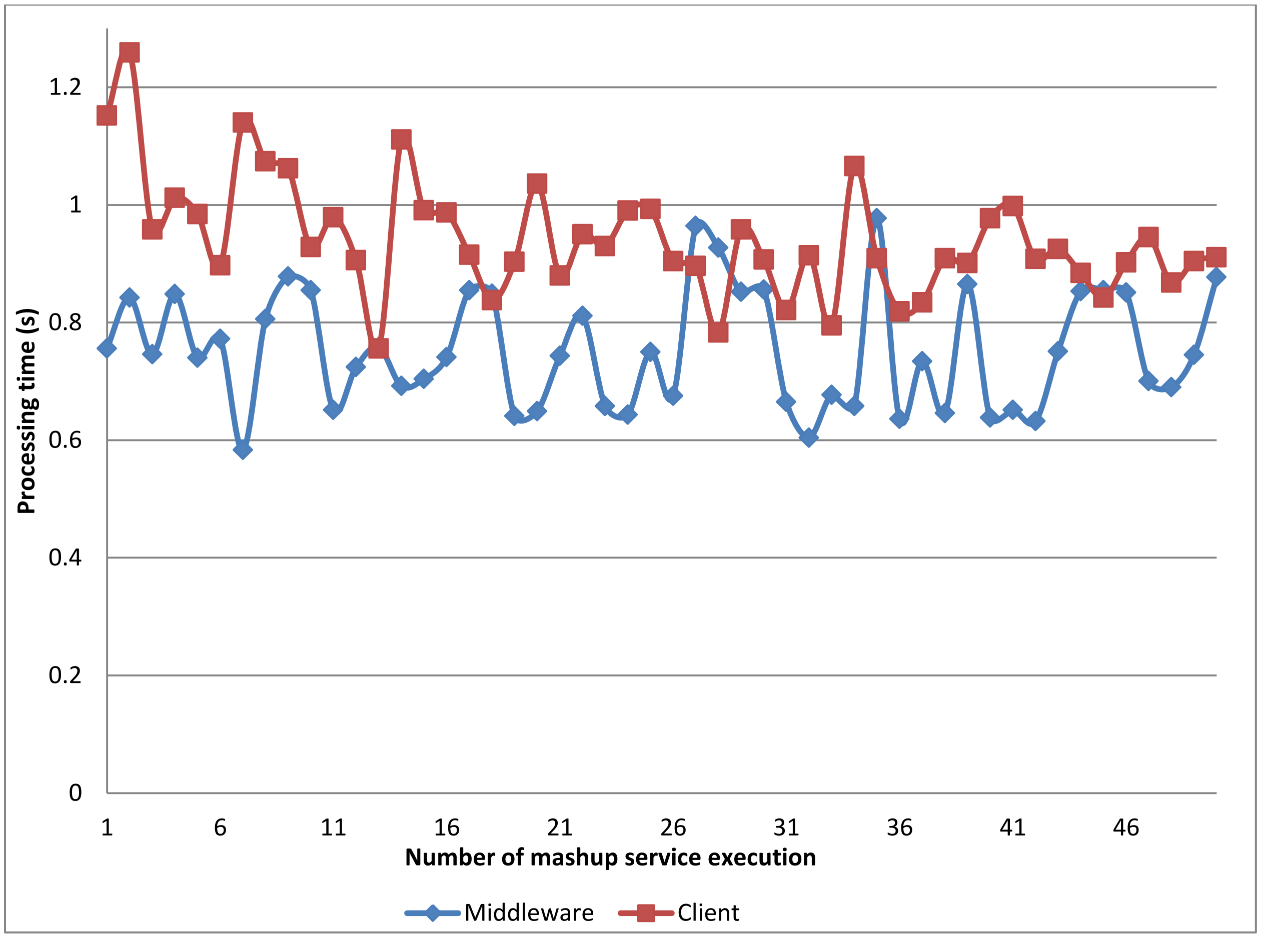


Figure 6.10 Processing time of mashup services

6.9 *Scalability of Amazon EC2 and GAE*

The main function of our middleware is adaptation and service mashup which involves mainly CPU, and network IO operations. When the middleware obtains service results from different Cloud Services, it establishes outbound network connections. When the middleware receives the results, it analyses and combines them. This experiment examines the scalability and robustness of our middleware design in the two different Cloud environments (Amazon EC2 and GAE). In particular, it shows the response time of the middleware for processing a service mashup request, how the response time of a mashup request changes when the load of the middleware increases and at which request rate the middleware fail to response. The test server is EC2 c1.xlarge instance and load generator is on EC2 m1.medium instance. (See *table 6.4* for hardware specification).

Figure 6.10 shows how the middleware processes a mashup request. When the middleware receives a GET request, it first obtains service results from the Cloud Services. Because most of the Cloud Services have limited numbers of request call, I use a web page (http://google.ca) instead of Cloud Services. For simulating CPU computation, the middleware calculates a Fibonacci number from 1 to 35 using normal recursion (without accumulator). Final, it returns a response with the calculated result to the client.

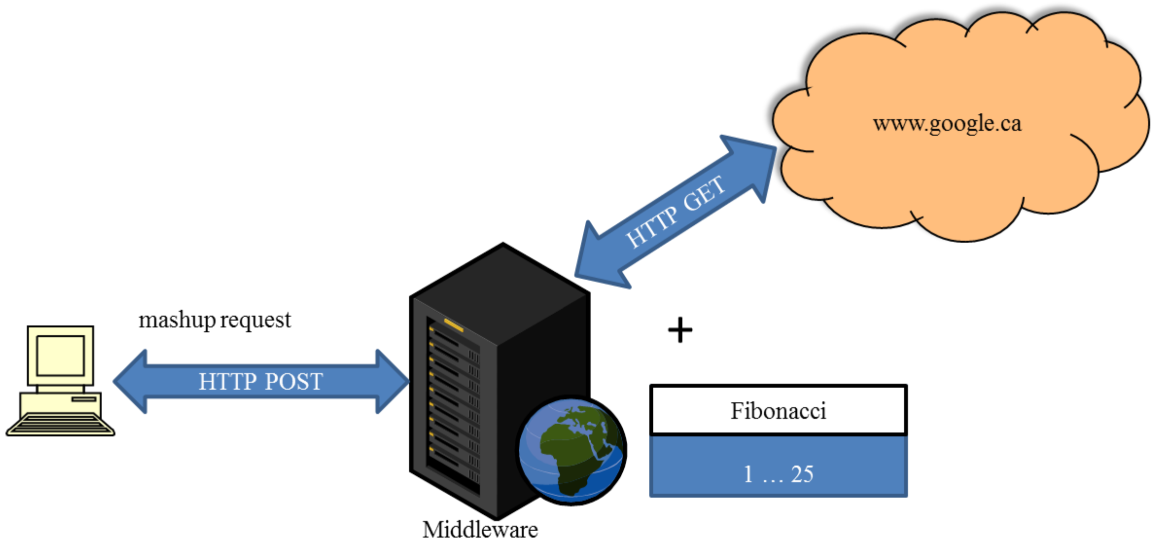


Figure 6.11 Process of a mashup request

The duration of each load is 30 minutes. Response time is calculated every 10 second. The full result is shown in *figure 6.11* for GAE and *figure 6.12* for EC2. The x-axis is the rate of sending HTTP request. At each request rate, there are three values on the y-axis, the average of response time, the highest and lowest response time.

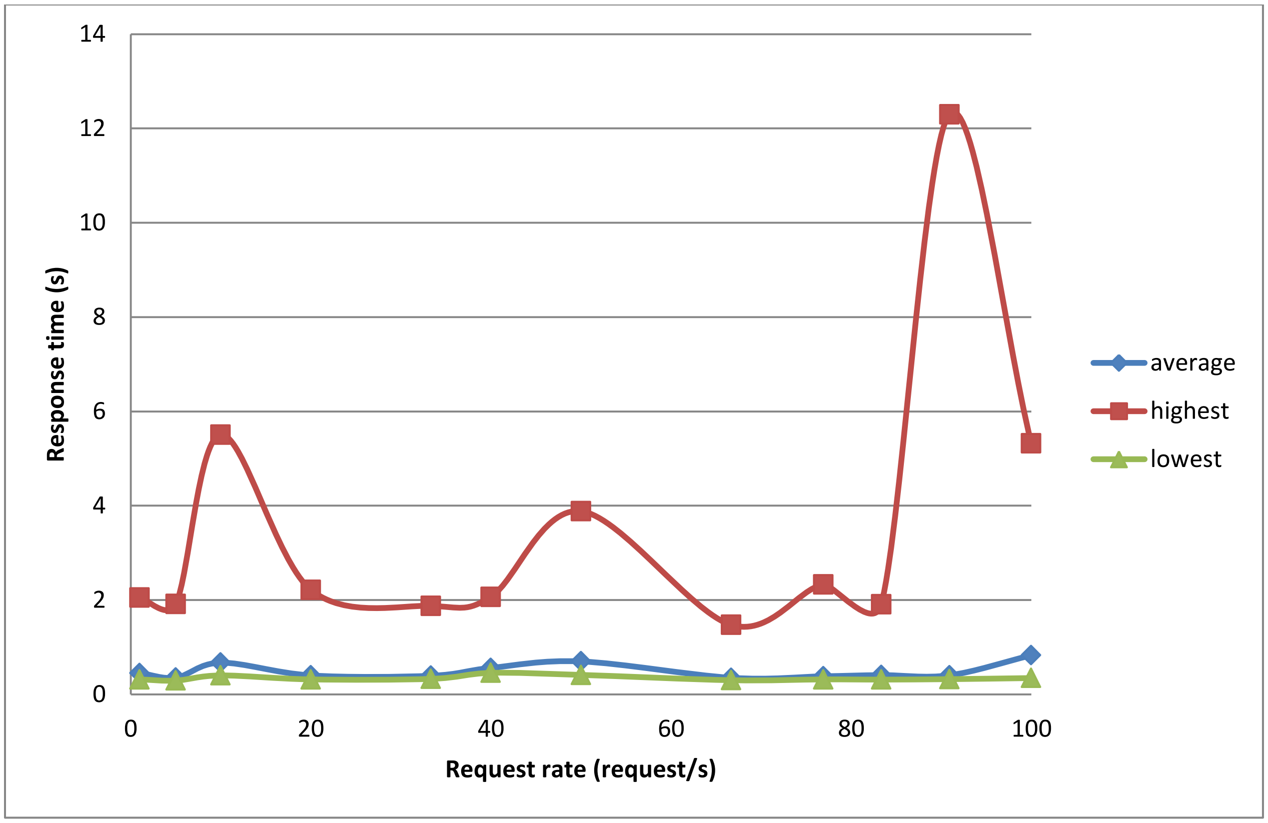


Figure 6.12 Response Time of GAE

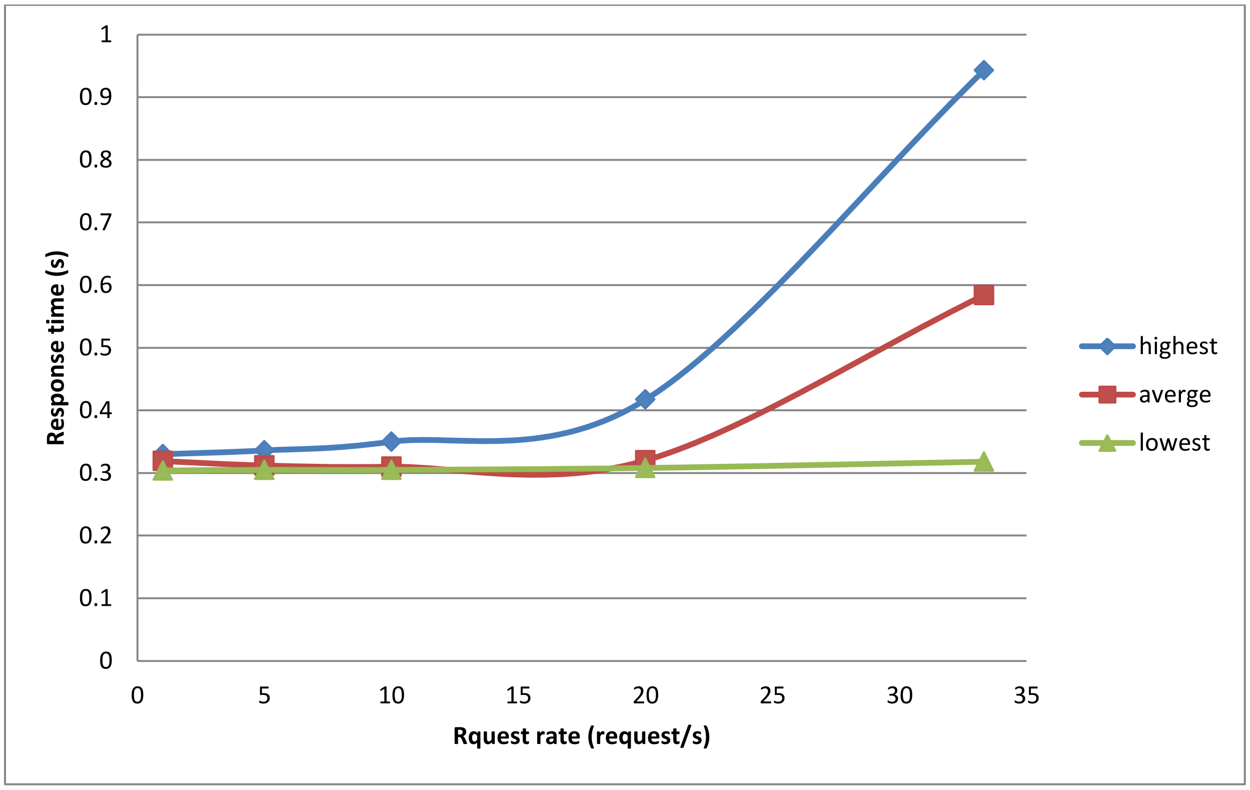


Figure 6.13 Response Time of EC2

The result shows that GAE scales. The average and lowest response time is constant for GAE. The error rate is below 0.1% for all request rates shows high availability. There are several points where the highest response is exceptional. Because GAE shares the resources of Google's infrastructure, the amount of resources for an application is not constant all the time. For example, when Google experiences a high volume of load, a GAE application may get fewer resources, thus response slower.

EC2 with Glassfish can only reach 33 requests per second without significant errors due to the limitations of the Java language. With 40request/s, the average response time is 145.9s and half of the requests failed or dropped. The highest and average response time increases when the request rate increases. This can be explained by the cloud mode of EC2. An EC2 instance reserves a constant amount of resources when it starts. EC2 instance does not share resource with each other. With the same amount of resource, it is expected that the response time increases as the load increases.

6.10 Summary

The experiments evaluated the Cloud Mobile Computing according to the three main research goals. This section summarizes the results of the experiments for each of the research goals.

Goal 1. **To enhance the interaction between mobile clients and Web Services**

Experiment Goal 1.1. Evaluated the cross-platform capability of the mobile clients design. Experiment 6.4 proved that the mobile client can be implemented on Android as well as Blackberry. Both of them can send HTTP requests, thus consume RESTful WS easily.

Experiment Goal 1.2. Evaluated the mobile client with different implementation models. Experiment 6.5 showed that the mobile client can implemented as embedded browser application and native application on Blackberry. The embedded browser application has better performance than native application. The native application has better access to platform features.

Experiment Goal 1.3. Consume RESTful WS through the middleware. Experiment 6.3 showed the overhead of consuming RESTful WS through the middleware. Without result optimizations, the middleware adds a little overhead (about 10% on average for both JSON and XML) to the response time.

Experiment Goal 1.4. Transfers SOAP WS to RESTful WS to be consumed by mobile clients. Experiment 6.3 showed SOAP WS have larger response time than RESTful WS in general and the client can consume a SOAP WS through the RESTful WS interface provided by the middleware.

Experiment Goal 1.5. Reduces bandwidth consumption of mobile clients. Experiment 6.3 showed that result optimizations reduce response size from 114KB to 5.44KB, but adds overhead to the response time. With JSON messages, the overhead is 0.1s on average, while with XML message, the overhead is 0.8s on average. Experiment 6.6 shows the parsing time of JSON message is less than the parsing time of XML message.

Experiment Goal 1.6. Push updates to mobile clients in real-time. Experiment 6.7 showed the updates takes more time to reach the mobile clients, when using push via email, because the underlying implementation of email client uses pulling. However, pushing saves bandwidth (46.08KB less) and energy (120 request less) compare to pulling.

Goal 2. To use the Cloud platform as a way to improve scalability and reliability of the middleware

Experiment Goal 2.1. The middleware can be implemented on EC2 and GAE. Experiment 6.8 showed the mobile client can consume service mashup through the middleware hosted on Amazon EC2.

Experiment Goal 2.2. Cloud platform improves the scalability and reliability of the middleware. Experiment 6.9 showed the GAE is scalable and reliable. The GAE can handle up to 100 request/s with less than 0.1% errors. The response time is also constant. However, Amazon EC2 with Glassfish cannot scale well (33request/s), because the limitations of Java language.

Goal 3. To provide service mashup platform for mobile clients

Experiment Goal 3.1. Create and consume service mashup via the middleware on EC2. Experiment 6.8 showed that despite the network transmission overhead, it is still efficient to execute service mashups on the middleware. The response time of a service mashup request through the middleware is about 200ms less on average, comparing to execute the service mashup directly on the mobile client.

Chapter 7 SUMMARY AND CONTRIBUTION

As service consumers, mobile devices have unique properties. They are small and portable. They are personal devices with various sensors. However, mobile devices have limitations, for example, small bandwidth, loss connectivity and less process power. On the another hand, the existing services are normally designed for stationary clients. For example, SOAP is a verbose protocol which involves a lot of XML parsing. To overcome the limitations, this research presents the Mobile Cloud Computing architecture for connecting mobile device to the existing Cloud Services.

The proposed mobile client design is mobile platform independent. The mobile client provides an interface for users to define mashup services and consume them through the middleware. It interacts with the middleware through RESTful WS interface. The mobile client has been implemented on two major mobile platforms, Android and Blackberry. The mobile client design involves both native application and embedded browser. For better compatibility, the interface can be implemented on embedded browser with HTML, CSS and JavaScript, while the actual client component is implemented in platform dependent language.

The middleware provides adaptation for mobile clients to Cloud Services. To support existing SOAP WS, the middleware transforms the SOAP WS to RESTful WS and XML message to JSON format. The middleware also provides result optimizations which extract the required data from the original service results. Finally, the middleware uses email push to efficiently deliver content to the mobile client.

The middleware is a Personal Service Mashup Platform which provides personal service mashup for mobile clients. Users can define and save mashup services on the middleware. The middleware does the mashup for the mobile clients including interacting with Cloud Services and combining the service results. In addition, the middleware caches the service result.

The middleware has been implemented and hosted on both EC2 and GAE. Cloud Computing enables a scalable and cost efficient way to deploy the middleware. GAE is highly scalable, because the applications share Google's infrastructure. However, because resources are shared, the Quality of Service (QoS) is hard to control. EC2 is very stable but hard to scale, since users have the complete control of the Virtual Machines.

The experiments proved the following design of the mobile client and middleware.

- The mobile client is able to consume both SOAP and RESTful WS through the middleware.
- The mobile client can be implemented on different mobile platforms.
- The mobile client can be implemented as a native as well as embedded browser application.
- JSON format works more efficiently than XML format in mobile environment.
- Middleware push saves energy and bandwidth.
- The mobile client can consume mashup service from the middleware.
- It is more efficient to do mashup on the middleware than the client-side.
- The middleware can be hosted on GAE and EC2.
- GAE is scalable, while users have full control of the virtual machine on EC2.

Chapter 8 FUTURE WORKS

8.1 SOAP WS Support

Currently, the mashup platform allows users to create mashup services only from RESTful WS. To fully support SOAP WS, there are two features I need to add to the middleware.

- Data structures that represent SOAP WS: When users define a SOAP WS, the SOAP WS needs to be stored in a database in a certain structure. Therefore, we need to add all the SOAP WS properties, such as endpoint, method, parameters into the current definition of SA.
- Constructing and parsing SOAP message: SOAP WS requires HTTP requests and responses to follow the SOAP standard, which is a special XML format. The middleware needs to construct a SOAP message from a pre-defined SOAP WS (SA), as well as parsing the SOAP message in the response to extract the desired results.

8.2 Caching on Mobile Client

Caching is a common strategy to cope with limited bandwidth and lost connectivity. There are different approaches for mobile client-side caching. Three types of catching strategy (see *figure 7.2*) are experimented: basic caching, live connection, and piggy-back fetching. In the prepared experiment, I will examine validity and efficiency of the three catching strategies.

- Basic caching: The cached data is loaded as the application start. The cached data is destroyed on application exit.
- Live connection: This is an implementation of publish-subscribe model. For each subscribed resource, the proxy keeps a HTTP connection with the server. Whenever the

subscribed resource has changed, the server sends the changed data to the proxy through that HTTP connection.

- Piggy-back fetching: The server has a list of resources a client subscribing. Each time the client send a HTTP request to obtain a resource, the server return a HTTP response with changes on other resources.

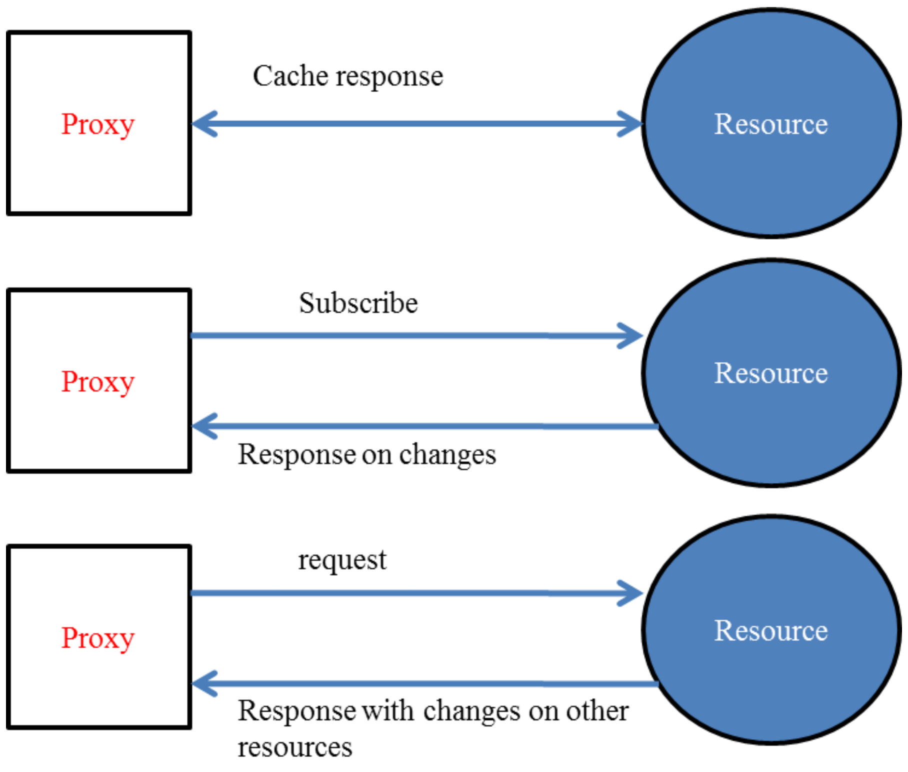


Figure 8.1 Three types of caching strategies

REFERENCES

[1] *Portio Research Mobile Factbook,* Portio Research, 2009.

[2] S. Yates, *It's Time To Focus On Emerging Markets For Future Growth*, Forester, 2007.

[3] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D.F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More,* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[4] "Web Services Glossary," 2004. Last retrieved from http://www.w3.org/TR/ws-gloss/ on December 6, 2010

[5] "Web Services Description Language 1.1," *Web Services Description Language (WSDL) 1.1,* 2001. Last retrieved from http://www.w3.org/TR/wsdl on December 6, 2010

[6] "UDDI version 3.02 Spec Technical Committee Draft," 2004. Last retrieved from http://uddi.org/pubs/uddi-v3.0.2-20041019.htm on December 6, 2010

[7] "Web Services Architecture," 2004. Last retrieved from http://www.w3.org/TR/ws-arch/ December 6, 2010

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

[9] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.,* vol. 39, 2009, pp. 50–55.

[10] "Google App Engine," *Google Code*, Mar. 2010. Last retrieved from http://code.google.com/appengine/ on December 6, 2010

[11] "API Dashboard," *Programmable Web,* Mar. 2010. Last retrieved from http://www.programmableweb.com/apis on December 6, 2010

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

[13] M. Satyanarnynnan, "Mobile computing," *Computer,* vol. 26, 1993, pp. 81-82.

[14] D.E. Bakken and M. Api, *Middleware*, 2001.

[15] P. Farley and M. Capp, "Mobile Web Services," *BT Technology Journal,* vol. 23, 2005, pp. 202-213.

[16] E. Oliver, "A survey of platforms for mobile networks research," *SIGMOBILE Mob. Comput. Commun. Rev.,* vol. 12, 2008, pp. 56–63.

[17] Q. Wang and R. Deters, "SOA's Last Mile-Connecting Smartphones to the Service Cloud," *Cloud Computing, IEEE International Conference on*, 2009, pp. 80-87.

[18] M. Tian, T. Voigt, T. Naumowicz, H. Ritter, and J. Schiller, "Performance considerations for mobile web services," *Computer Communications,* vol. 27, 2004, pp. 1097 - 1105.

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

[20] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.,* vol. 21, 1989, pp. 261–322.

[21] W. Emmerich, "Software engineering and middleware: a roadmap," *ICSE '00: Proceedings of the Conference on The Future of Software Engineering,* New York, NY, USA: ACM, 2000, pp. 117–129.

[22] K. Farooqui, L. Logrippo, and J.D. Meer, "The ISO Reference Model for Open Distributed Processing: an introduction," *Computer Networks and ISDN Systems,* vol. 27, 1995, pp. 1215 - 1229.

[23] A. Uribarren, J. Parra, J.P. Uribe, M. Zamalloa, and K. Makibar, “Middleware for Distributed Services and Mobile Applications,” *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, New York, NY, USA: ACM, 2006, Article 10.
[24] “UPnP Standard.” Last retrieved from http://www.upnp.org/standardizeddcps/default.asp on December 6, 2010
[25] “OSGi - The Dynamic Module System for Java,” Mar. 2010. Last retrieved from http://www.osgi.org/Main/HomePage on December 6, 2010
[26] T. Phan, R. Guy, and R. Bagrodia, “A Scalable, Distributed Middleware Service Architecture to Support Mobile Internet Applications,” *WMI '01: Proceedings of the first workshop on Wireless mobile internet*, New York, NY, USA: ACM, 2001, pp. 27–33.
[27] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, “A mobile computing middleware for location- and context-aware internet data services,” *ACM Trans. Internet Technol.*, vol. 6, 2006, pp. 356–380.
[28] A.K. Dey, “Providing Architectural Support for Building Context-Aware Applications,” Georgia Institute of Technology, 2000.
[29] M.A. Vouk, “Cloud computing: Issues, research and implementations,” *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, 2008, pp. 31–40.
[30] D.E. Atkins, K.K. Droegemeier, S.I. Feldman, H. Garcia-molina, M.L. Klein, D.G. Messerschmitt, P. Messina, J.P. Ostriker, and M.H. Wright, “Revolutionizing Science and Engineering Through Cyberinfrastructure,” 2003, pp. 50–56.
[31] “Amazon Elastic Compute Cloud,” *Amazon Elastic Compute Cloud (Amazon EC2)*, Mar. 2010. Last retrieved from http://aws.amazon.com/ec2/ on December 6, 2010
[32] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, “An Early Performance Analysis of Cloud Computing Services for Scientific Computing,” Dec. 2008.
[33] “Compute/Calcul Canada.” Last retrieved from https://computecanada.org/?pageId=138 on November 25, 2010
[34] “Memory Bandwidth: Stream Benchmark Performance Results.” Last retrieved from http://www.cs.virginia.edu/~mccalpin/papers/balance/ on December 6, 2010
[35] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, and D. Takahashi, “The HPC Challenge (HPCC) benchmark suite,” *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA: ACM, 2006, p. 213.
[36] “HPC Challenge results,” *HPC Changllenge*, 2008. Last retrieved from http://icl.cs.utk.edu/hpcc/hpcc_results.cgi on December 6, 2010
[37] “Apache VCL Project,” *Apache VCL*, Mar. 2010. Last retrieved from http://cwiki.apache.org/VCL/ on December 6, 2010
[38] “IBM BladeCenter,” Mar. 2010. Last retrieved from http://www-03.ibm.com/systems/bladecenter/ on December 6, 2010
[39] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, 2008, pp. 107–113.
[40] G.F. F. Berman and T. Hey, *Grid Computing: Making the Global Infrastructure a Reality*, New York: Wiley, 2003.
[41] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The Eucalyptus Open-Source Cloud-Computing System,” *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*,

Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[42] M. Papazoglou and D. Georgakopoulos, “Service-oriented Computing,” *COMMUNICATIONS OF THE ACM*, vol. 46(10), 2003, pp. 25-65.

[43] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, “Service-oriented Computing,” *Communications of the ACM*, vol. 46, 2003, pp. 25–28.

[44] F. Curbera, *Business Process Execution Language for Web Services, Version 1.0*, 2002.

[45] “OWL-S: Semantic Markup for Web Services,” 2004. Last retrieved from http://www.w3.org/Submission/OWL-S/ on December 6, 2010

[46] J. Yang and M.P. Papazoglou, “Web Component: A Substrate for Web Service Reuse and Composition,” *Advanced Information Systems Engineering*, 2002, pp. 21-36.

[47] X. Liu, Y. Hui, W. Sun, and H. Liang, “Towards Service Composition Based on Mashup,” *Services, 2007 IEEE Congress on*, 2007, pp. 332-339.

[48] J. Yu, B. Benatallah, F. Casati, and F. Daniel, “Understanding Mashup Development,” *Internet Computing, IEEE*, vol. 12, Oct. 2008, pp. 44-52.

[49] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner, and C. Schroth, “Enterprise Mashups: Design Principles towards the Long Tail of User Needs,” *Services Computing, 2008. SCC '08. IEEE International Conference on*, 2008, pp. 601-602.

[50] V. Hoyer and M. Fischer, “Market Overview of Enterprise Mashup Tools,” *Service-Oriented Computing ICSOC 2008*, 2008, pp. 708-721.

[51] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: an annotated bibliography,” *SIGPLAN Not.*, vol. 35, 2000, pp. 26–36.

[52] E.M. Maximilien, A. Ranabahu, and K. Gomadam, “An Online Platform for Web APIs and Service Mashups,” *Internet Computing, IEEE*, vol. 12, Oct. 2008, pp. 32-43.

[53] H. Xu, M. Song, H. Chen, And J. Song, “Research on SOA Based Mobile Mashup Platform for Telecom Networks,” *The Journal of China Universities of Posts and Telecommunications*, vol. 15, 2008, pp. 31 - 36.

[54] C. Isaacson, *Software Pipelines and SOA: Releasing the Power of Multi-Core Processing*, Addison-Wesley Professional, 2009.

[55] “Tsung.” Last retrieved from http://tsung.erlang-projects.org/ on December 6, 2010.