

國立交通大學

資訊學院資訊科技（IT）產業研發碩士專班

碩 士 論 文

手機應用程式與 RESTful APIs 間的網路傳輸減量

Transmission Reduction between Mobile Phone
Applications and RESTful APIs

研 究 生：蔡金亮

指 導 教 授：黃俊龍 教授

中 華 民 國 九 十 九 年 八 月

手機應用程式與 RESTful APIs 間的網路傳輸減量

Transmission Reduction between Mobile Phone Applications and
RESTful APIs

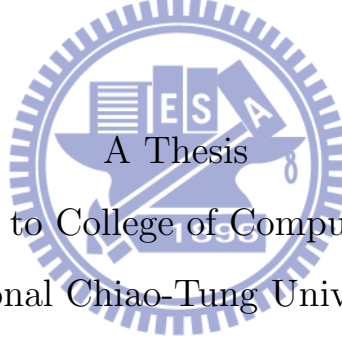
研 究 生：蔡金亮
指 導 教 授：黃俊龍

Student: Chin-Liang Tsai
Advisor: Jiun-Long Huang

國立交通大學

資訊學院資訊科技 (IT) 產業研發碩士專班

碩 士 論 文



Submitted to College of Computer Science
National Chiao-Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Industrial Technology R & D Master Program on
Computer Science and Engineering

August 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年八月

手機應用程式與 RESTful APIs 間的網路傳輸減量

學生：蔡金亮

指導教授：黃俊龍

國立交通大學資訊學院資訊科技 (IT) 產業研發碩士專班

摘 要

近幾年來，越來越多的使用者使用智慧型手機等手持設備上網，使用者可以透過網路獲取他們的資訊或者是在熱門的社群網站（例如Twitter、Facebook和MySpace）上更新他們的狀態。而這些熱門的社群網站通常會提供API（Application Programming Interface）。開發人員可以使用這些 API來創建一個新的網站或是開發手機應用程式。REST（Representational State Transfer）是最常見也最多網路服務提供者所使用的API架構。儘管這些熱門的社群網站或其他網站已經存在專門供手機瀏覽的行動版本網站，讓用戶可以使用手機瀏覽器來訪問。但是使用手機應用程式透過他們提供的RESTful APIs來存取這些網站會有很多優點，比方說，手機應用程式有很炫很好用的使用者介面，他們可以與手機作業系統結合，例如開放原始碼的Android平台。開發人員可以開發一個 RESTful客戶端應用程式來避免下載整個 HTML或JavaScript檔案所造成的網路傳輸量。但RESTful APIs在低頻寬的無線網路中也存在傳輸上的負擔與浪費。在本篇論文中，我們觀察到在低頻寬的無線網路中，手機應用和RESTful APIs間的傳輸負擔，然後我們提出一個系統架構，以減少這些傳輸負擔。進而加快回應時間與減少網路傳輸量。

Transmission Reduction between Mobile Phone Applications and RESTful APIs

Student: Chin-Liang Tsai

Advisor: Jiun-Long Huang

Submitted to College of Computer Science
Computer Science and Engineering
National Chiao-Tung University

ABSTRACT

In recent years, more and more users use the handheld devices such as smartphone to access the Internet. Users can get their data from the Internet or update their status to the hot social networking Web sites (e.g., Twitter, Facebook and MySpace). These popular social networking Web sites usually provide the API (Application Programming Interface). Developers can use these APIs to rebuild a new Web site or a mobile phone application. The REST (Representational State Transfer) scheme is most famous architecture style to call these APIs. Despite there are already exist friendly mobile version Web sites. Users can use mobile Web browser to access these hot social networking Web sites or others. But there are many advantages in using mobile phone applications to access these Web sites through the RESTful APIs they provide. For example, mobile phone applications have fantastic UI and they can integrate with the mobile phone operation system such as open-source Android platform. Developers can develop a RESTful client application to avoid to download entire HTML or Javascript files that will cause many network traffics. But RESTful APIs also have overhead in transmission. In this thesis, we observed the overhead between mobile phone applications and RESTful APIs in low-bandwidth wireless network. We proposed a system architecture to reduce these transmission overheads. And then, speed up the response time and decrease the total transmission bytes.

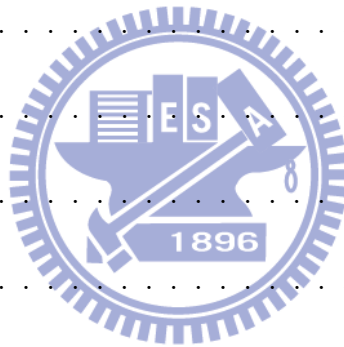
誌 謝

首先我要感謝我的指導教授黃俊龍老師，感謝他在碩士班期間內給予我研究上的指導與論文寫作的技巧，讓我遇到瓶頸時有個解決的方向。我也要感謝每位口試委員—交通大學的曾建超教授、中央大學的胡誌麟教授以及淡江大學的林順傑教授給我論文上的建議與指導。我更要感謝我的父母給我一個美好的家庭與經濟環境，讓我可以專心的做研究。也要感謝實驗室的振哲學長、學弟、建益同學、好友坤澤同學以及美拉小姐還有其他曾幫助過我的人的鼓勵與幫助，讓我有個對象可以討論，幫助思考，讓我解決論文中的盲點，使我的論文能順利完成。

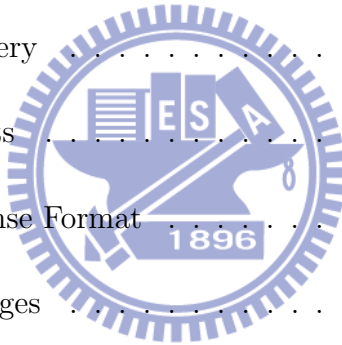


Contents

| | |
|--|------|
| 書名頁 | i |
| 中文摘要 | ii |
| 英文摘要 | iii |
| 誌謝 | iv |
| Contents | v |
| List of Tables | viii |
| List of Figures | ix |
| 1. Introduction | 1 |
| 2. Preliminaries | 5 |
| 2.1 Related Work | 5 |
| 2.1.1 Web Browsing | 5 |
| 2.1.2 Web Content Adaptation | 6 |
| 2.2 Open API | 7 |
| 2.2.1 REST Overview | 7 |
| 2.2.2 SQL-Style API | 8 |
| 2.2.3 OAuth | 8 |
| 2.3 Motivation | 9 |
| 2.3.1 Main Issue | 9 |
| 2.3.2 Request a RESTful API | 10 |
| 2.3.3 Observations | 11 |



| | |
|--|----|
| 3. System Architecture | 17 |
| 3.1 Proposed System Architecture | 17 |
| 3.2 HTTP Header Reduction | 18 |
| 3.3 Client-Side Library | 19 |
| 3.3.1 API Query Language | 19 |
| 3.3.2 Image Multi-Get Module | 22 |
| 3.3.3 Gzip Compression Request | 25 |
| 3.3.4 Spilt Combined Image | 26 |
| 3.4 Proxy-Side Library | 26 |
| 3.4.1 Parsing the Query | 26 |
| 3.4.2 Filtering Results | 26 |
| 3.4.3 Convert Response Format | 27 |
| 3.4.4 Combining Images | 27 |
| 3.4.5 OAuth Authentication | 28 |
| 4. Experimental Results | 29 |
| 4.1 Experimental Setup | 29 |
| 4.2 Common Plain Text | 30 |
| 4.3 Multiple Images | 35 |
| 4.3.1 Image Quality | 35 |
| 4.3.2 Image Resize | 35 |
| 4.3.3 Normal versus AQL Picasa Application | 36 |
| 5. Conclusion | 39 |

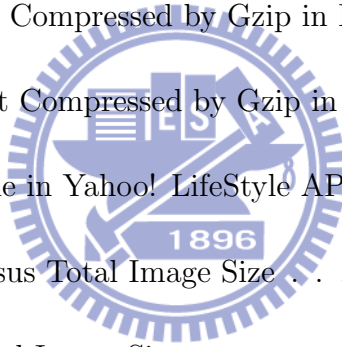


Bibliography 40



List of Tables

| | | |
|-----|---|----|
| 2.1 | Relationships Between SQL and HTTP Verbs | 8 |
| 2.2 | 3G Flat-Rate of Hinet emome | 10 |
| 2.3 | Yahoo! LifeStyle API Method List | 11 |
| 3.1 | Query Format of AQL and Relationships Between AQL and HTTP Verbs . | 19 |
| 4.1 | Specifications of Our Client Device (HTC Desire A8181) | 30 |
| 4.2 | Experimental Parameters in Yahoo! LifeStyle API | 31 |
| 4.3 | Ratio of XML Content Compressed by Gzip in LifeStyle API (Bytes) . . . | 32 |
| 4.4 | Ratio of JSON Content Compressed by Gzip in LifeStyle API (Bytes) . . . | 33 |
| 4.5 | Average Response Time in Yahoo! LifeStyle API (ms) | 34 |
| 4.6 | Quality Parameter versus Total Image Size | 35 |
| 4.7 | Resize Rate versus Total Image Size | 36 |
| 4.8 | Normal Picasa APP. versus PicasaAQL APP. | 38 |



List of Figures

| | | |
|------|--|----|
| 2.1 | Common REST Architecture | 7 |
| 2.2 | OAuth Authentication Flow | 9 |
| 2.3 | Common RESTful Call in Android Client | 10 |
| 2.4 | HTTP Request Header from Our Client | 12 |
| 2.5 | HTTP Response Header that Including Gzip Compression Information | 12 |
| 2.6 | HTTP Response Header from Yahoo! LifeStyle API Server | 13 |
| 2.7 | Result of Call “statuses/friends” from Twitter API Server | 14 |
| 2.8 | Normal flow to use API to get photos | 15 |
| 2.9 | Per Image Need Per Request and Response Pair | 15 |
| 2.10 | Sample Picasa Android Application | 16 |
| 3.1 | Proposed System Architecture | 17 |
| 3.2 | Simplified HTTP Request and Response Header | 18 |
| 3.3 | “Select *” and “Select id, name” From Twitter.statuses/friends | 20 |
| 3.4 | Flow of SubSelect Query | 21 |
| 3.5 | Two Implement of Picasa Android Application | 23 |
| 3.6 | Image Multi-Get Module | 24 |
| 3.7 | AQL HTTP Request and Response Header | 25 |
| 3.8 | Example of OAuth Authentication | 28 |
| 4.1 | HTC Desire A8181 | 29 |

| | | |
|-----|--|----|
| 4.2 | Gzip Compression in LifeStyle API (XML) | 32 |
| 4.3 | Gzip Compression in LifeStyle API (JSON) | 33 |
| 4.4 | Line Chart of Average Response Time in Yahoo! LifeStyle API (ms) | 34 |
| 4.5 | Line Chart of Quality Parameter and Resize Rate versus Total Image Size | 36 |
| 4.6 | Application Screen Capture | 37 |
| 4.7 | Line Chart of Response Time and Total Transmission Traffic | 38 |



Chapter 1

Introduction

In recent years, handheld devices (e.g., Android phone, iPhone and Windows phone) were very popular and the wireless networks (e.g., WiFi, 3.5G and 3G) were develop so quickly. Using mobile phone to access the Internet had became a trend. In addition, for the concept of Cloud Computing, more and more people put their data into the Internet. Moreover, there were many people using the social networking Web sites (e.g., Twitter, Facebook and MySpace) to contact with their friends. People can use the handheld devices such as the smartphone to access the Internet and get their data from the Cloud or update their status to the hot social networking Web sites. These popular social networking Web sites usually provide the API (Application Programming Interface). Developers can get data through these APIs to rebuild a new Web site or a mobile phone application. The REST (Representational State Transfer) [1] scheme is most famous architecture style to call these APIs. Conforming to the REST constraints is referred to as being “RESTful”¹. Developers can use RESTful APIs to develop the mobile phone applications such as Android² RESTful client applications.

Although it is very convenient to access the Internet by the mobile phone browser and the popular Web sites also provide the mobile version Webs, but there are some drawback by using the mobile phone browser to access the Internet in wireless environment. Comparing to the wired network, wireless network is high-cost, high-latency, low-bandwidth and low-reliability [2]. Current Web browsers greedy fetch the entire HTML from the

¹Representational State Transfer From Wikipedia. URL<http://en.wikipedia.org/wiki/Representational_State_Transfer>

²Google Projects for Android. URL<<http://code.google.com/intl/en/Android>>

server. This is inappropriate for use in low-bandwidth networks because mobile devices always design for easy-to-carry, so their screen size are small. Mobile Web browsers will fetch the off-screen objects to cause large response time for users. [3] indicated this problem and propose three mechanisms for Web browser to reduce Web response time. On the other hand, there are some schemes such as WAP [4] , WML³ , and BREW⁴ have also been developed to address the limitation of Web performance on mobile devices. However, using Web browser to surf the Web sites is a “what they provide, what you may see” scheme. There is another choice to access the Internet by the mobile devices. We suggest developing the RESTful mobile client application to get on-demand data. We choose the Android platform because it has free and familiar cross-platform development tool, moreover, it is an open source operating system platform. Developers who have object-oriented programming concepts will easy to get started. It will reduce cost of developing. In the Android platform, there were five reasons that mentioned by Google I/O 2010⁵ for why developing Android applications if mobile friendly Web sites already exist. We list as follow:

(1) Android applications will be able to integrate with the Android platform :

It will be able to use Intents⁶, Content Providers⁷ and be able to access all the private APIs available only to Android applications. (Can't do from browsers).

(2) Android applications can offer intents to other applications :

It will be able to enrich the behavior of the platform by offering new functionality to other applications.

(3) Android applications can run in the background :

If you would like to have your application refresh the data from the server and new

³Wireless Application Protocol Forum, WML 2.0 DTDs. URL<<http://www.wapforum.org/DTD/wml20.dtd>>

⁴Qualcomm Inc., Binary Runtime Environment for Wireless. URL<<http://brew.qualcomm.com>>

⁵Virgil Dobjanschi, Developing Android RESTful client applications. URL<<http://code.google.com/intl/zh-TW/events/io/2010/sessions/developing-RESTful-android-apps.html>>

⁶Android - Intents and Intent Filters. URL<<http://developer.android.com/guide/topics/intents/intents-filters.html>>

⁷Android - Content Providers. URL<<http://developer.android.com/guide/topics/providers/content-providers.html>>

data is actually retrieved from the server, your application has the option to present a notification to the user to let them know that particular data is available.

(4) Limited connectivity :

The network is comes and goes in some wireless environments. An Android application has the option to run in the background and retry operations in the background by using an alarm for the purpose of relieving the user for trying to hit that refresh button or post in the browser. An Android application can fast than browsers. It has the option of retrieving all the network content or the on-demand content in JSON (JavaScript Object Notation), XML (Extensible Markup Language) or some binary format. Android applications can parse it and store it in a database, and then, just retrieving content that's newer than the one that you already have or older than the one you already have, but not the same data. Browsers will retrieve entire HTML and JavaScript that cause more time to download.

(5) User interface :

Android applications can innovate in many ways to deliver a fantastic user experience to the user. It will be so much faster and easier in some cases to use than the browser.



Nowadays, developers can produce their own applications for some mobile devices, such as iPhone applications and Android applications. Developers can publish their innovative applications to on-line market. And then, other users can download them for free or buy them. If the users buy the application from the on-line market, the author of the application will gain a part of the money from it. Developers will happy to produce a powerful application to earn a huge amount of money by this way. They will consider to using the API that provided by popular Web sites, because there are very large numbers of users access it.

There were many advantages for developing Android applications with RESTful APIs, but the bandwidth was a bottleneck. We observed the bandwidth usage between Android RESTful client applications an API servers. We found that when we call the RESTful APIs, there exists some unnecessary bandwidth waste. In addition, we observed some common application that used the RESTful API such as social networking applica-

tions and photo album applications. There were many small images to download such as the cover image of album list. It will take a lot of bandwidth for all API request. For these issues, we proposed a system architecture to reduce the transmission overhead when we use the RESTful APIs.

We set up a proxy server between mobile phone applications and RESTful APIs. Mobile phone applications include our Client-Side Library that provide HTTP header reduction and use an API Query Language (AQL) query to communicate with our proxy. Our proxy interacts with the mobile client and the API server. It has Proxy-Side Library that can parse the query and get the result from the API server. Proxy-Side Library can reduce the results by filtering and compressing. Proxy-Side Library also provide a Image-Multiple-Get (IMG) module to process the images. It can parallel download the multiple images and then be able to optional compress, resize and combine the images in order to reduce the total image bytes.

The experimental result shows our system architecture can reduce the transmission traffic over 61% if the content is plain text data. For the images, according to image count and parameter setting, it could reduce the total transmission bytes about 80% and speed up the response time to about 50% when there were over 10 small images and the quality parameter and resize rate was set to 50.

In this thesis, we design a system architecture that can reduce the transmission between Android applications and RESTful API servers. We implement it on Android mobile device. The remainder of this thesis is organized as follows: In Chapter 2, we present the preliminaries including the related work and the motivation about why we design thus system architecture. In Chapter 3 , we show our system architecture. In Chapter 4, we show the experimental results and Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

2.1 Related Work

In this section, we review the related work. Previous works were focus on using Web browser to surf the Internet on handheld devices.

2.1.1 Web Browsing

Over the past decades, a large number of studies had addressed the Web browsing in a wireless environment because of the challenges of mobile computing such as device heterogeneity and constraints (screen size, battery lifespan, color depth, computation power, etc.), content heterogeneity (audio, video, image, etc.), the network (GPRS, wireless LAN, Bluetooth, etc.) and user preferences.

Housel et al. [5] proposed WebExpress that was a Client/Intercept based system for optimizing Web browsing in a wireless environment. It is transparent to the client and the server and facilitates highly effective data reduction and protocol optimization. The performance of WebExpress is 60% to 90% reductions in wireless network traffic and 36% to 97% improvements in application response time.

Chang et al. proposed WebAccel [3] that identify two major reasons, screen contention and bandwidth under-utilization, which result in large user-perceived response time. WebAccel uses an intelligent mix of prioritized fetching, object reordering, and

connection management to address these problems.

2.1.2 Web Content Adaptation

Content adaptation emerges to remedy the problem by offering the different mobile users suitable versions of the same object.

Han et al. [6] derive the theoretical conditions of transcoding and present adaptive transcoding policies for mobile Web browser. The advantages of [6] are without requiring modifications to Web servers and browsers, an HTTP transcoding proxy can dramatically reduce Web download times over low-bandwidth links via data compression, reduce per-byte costs over tariffed links via data compression, and tailor Web data to a variety of client devices via format conversion. However, in most cases, they need to perform lossy compressions that degrade the quality of images or sound significantly.

Hwang et al. [7] focuses on structure-aware transcoding heuristics. The goal is to develop a high-quality syntax-based Web transcoding system that allows universal access to Web pages without manual reauthoring.

Apart from transcoding scheme, some studies [8], [9], [10], [11], [12] focused on Web page layout modification techniques to solve the restrained capability and limited bandwidth on mobile devices.

Hua et al. [13] proposed an adaptive scheme called MobiDNA for serving dynamic Web content in a mobile computing environment. Utilizing the fragment information through a modified content adaptation algorithm to adapt the dynamic content at first, and then, saving the adapted content to the mobile client cache for reducing network transmission and Web content adaptation costs.

2.2 Open API

Open API¹ (often referred to as OpenAPI) is a word used to describe sets of technologies that enable websites to interact with each other by using SOAP, Javascript and other Web technologies. It is not limited to web-based applications. It is become an increasing trend in Web 2.0 applications. Open API is a common service-oriented Web site application. Web service providers can package their own Web services into a series of API and then open out for third-party developers to use. Third-party developers do not need a huge investment in hardware and technology.

2.2.1 REST Overview

REST is an architectural style that Roy T. Fielding firstly defined in his doctoral thesis [1]. REST treats everything as “resources” and use URI (Uniform Resource Identifier) indicate the location of resources. Resource users use the HTTP method to operate resources. HTTP also defines the four basic methods, namely GET, POST, PUT and DELETE that generally correspond to four types of data processing actions CRUD (Create, Read, Update and Delete). GET retrieves the current state of a resource in some representation, PUT updates a resource, POST transfers a new state onto a resource, and DELETE removes an existing resource.

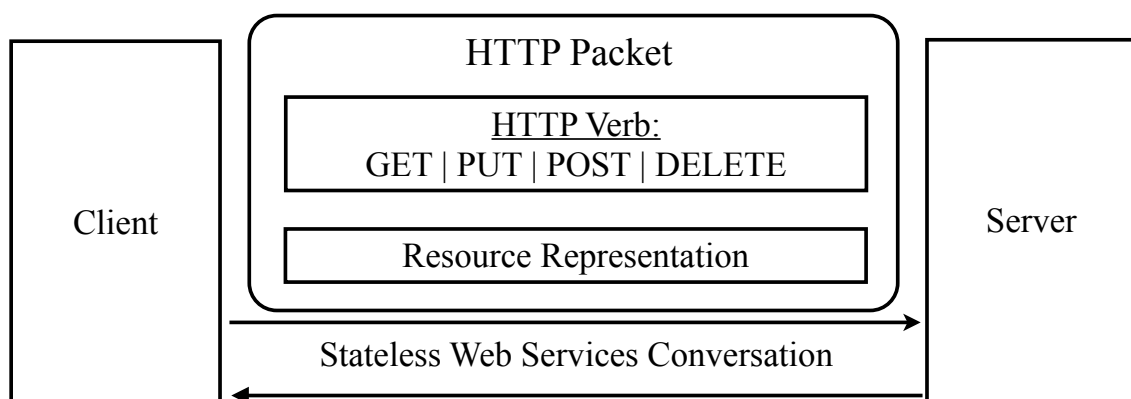


Figure 2.1: Common REST Architecture

¹Open API from Wikipedia. URL<http://en.wikipedia.org/wiki/Open_API>

2.2.2 SQL-Style API

Four basic HTTP methods just like SQL (Structured Query Language) four statements (Insert, Select, Update, Delete). Table 2.1 shows the relationship between SQL and HTTP verbs. SQL-Style API such as YQL [14] (Yahoo! Query Language) and FQL [15] (Facebook Query Language). The Yahoo! Query Language is an expressive SQL-like language that lets you query, filter, and join data across Web services. FQL is a way to query the same Facebook data you can access through the other API functions, but with a SQL-style interface.

Table 2.1: Relationships Between SQL and HTTP Verbs

| Action | SQL | HTTP | Description |
|--------|--------|--------|---|
| Create | Insert | POST | Create a resource without id |
| Read | Select | GET | Get a resource |
| Update | Update | PUT | Update a resource or create a resource with id if not exist |
| Delete | Delete | DELETE | Delete a resource |

2.2.3 OAuth

OAuth² provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end-user). It also provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (typically, a username and password pair), using user-agent redirections. Figure 2.2³ shows OAuth authentication flow. First, you need to sign up and submit some details about your application to the service provider. When your users get involved, your application uses your consumer key to obtain a request token, and then directs user to service provider. In this his time, user will direct to the url that consist of service provider's login url and request token. When users login and allow your application to access their private data, your application needs to exchange the approved request token for an access token, which tells Service Provider that your application has been given

²OAuth. URL<<http://oauth.net/>>

³OAuth Authentication Flow. URL<oauth.googlecode.com/svn/spec/core/1.0/diagram.pdf>

authorization to access user data. And then, your application will get the access token from service provider to access user's data until it expires.

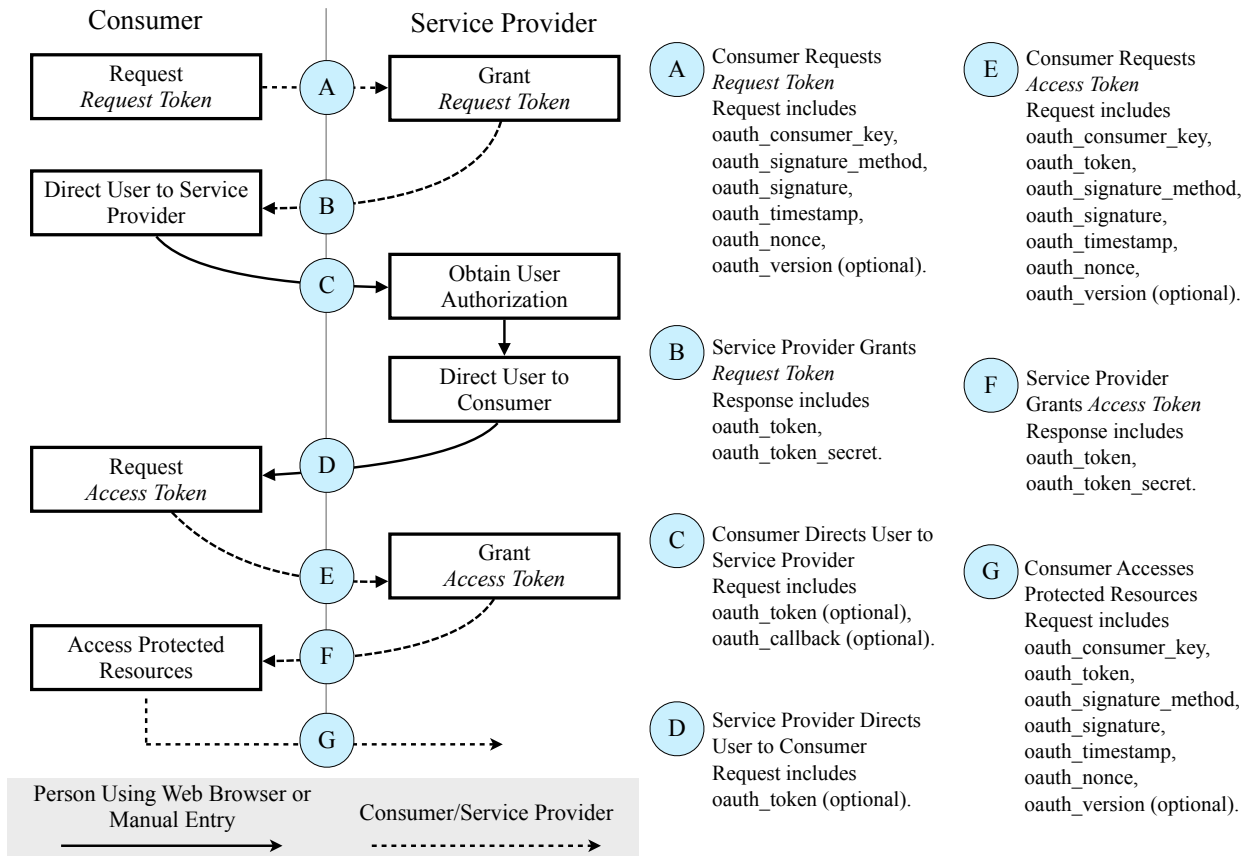


Figure 2.2: OAuth Authentication Flow

2.3 Motivation

In this section, we observed the behavior between Android application and API server. There are some overheads and inconvenient on it. According our observations and motivations, we proposed a better way to enhance them.

2.3.1 Main Issue

The Android application call the RESTful API in the wireless environment, the connection is more weaker than wireline network. Weak connection was a common issue in the wireless network. For the cost of wireless network, we survey the flat-rate of 3G

mobile network that formulated by Hinet⁴. Table 2.2 shows the 3G Flat-Rate of Hinet emome. The more packets you use, the more money you pay. Our system architecture was going to design for the purpose to alleviate the impact of the limited bandwidth between Android application and RESTful APIs in the wireless environment. It will reduce the total bytes of transmission, and then reduce the number of packets to save the transmission cost.

Table 2.2: 3G Flat-Rate of Hinet emome

| Flat-Rate Type | | 183 Style | 383 Style | 583 Style | 983 Style | 1683 Style |
|--|-------------------|-----------|-----------|-----------|-----------|------------|
| Monthly Fee (NT) | | 183 | 383 | 583 | 983 | 1,683 |
| Flat-Rate Data Packet (NT/Package) and Upper/Lower Bound | <= 500,000 | 0.005 | 0.0025 | 0.0013 | 0.0006 | 0.0003 |
| | (Upper Bound) | 2,500 | 1,250 | 650 | 300 | 150 |
| | 500,000~1,000,000 | 0.0025 | 0.0013 | 0.0006 | 0.0003 | 0.00016 |
| | (Upper Bound) | 2,500 | 1,300 | 600 | 300 | 160 |
| | >1,000,000 | 0.0013 | 0.0006 | 0.0003 | 0.00016 | 0.00008 |
| | (Lower Bound) | 1,300 | 600 | 300 | 160 | 80 |



2.3.2 Request a RESTful API

When developers using the RESTful API, their application need to send a HTTP request to API server, and then, API server response the result to the application. Figure 2.3 shows the common RESTful call in Android client. In Figure 2.3 , Android Client

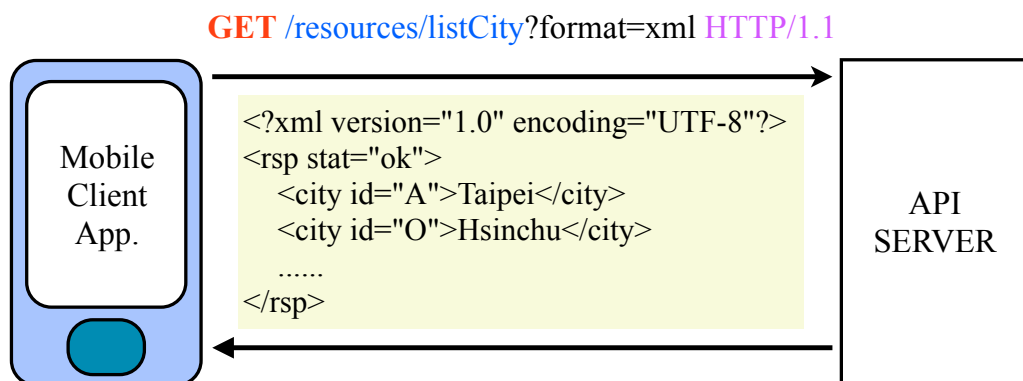


Figure 2.3: Common RESTful Call in Android Client

⁴3G flat-rate of Hinet emome. URL<<http://emome.asia/channel?chid=134>>

application assign a request URI in an HTTP GET request. This request URI include the parameters, one of the parameters is “format=xml” that “tell” the API server to return a result in XML format.

2.3.3 Observations

Observation Setup

We use the Google Android emulator with Android 2.2 Platform SDK⁵ to be our client platform and run an application on it. We use the Wireshark⁶ to capture and analysis the packets between the client application and API server.

Observation 1: HTTP Request in REST API

Table 2.3: Yahoo! LifeStyle API Method List

| API Method | HTTP Method | Description |
|----------------------|-------------|--|
| Auth.bootUp | GET | Boot Up the AppID, just need one times. |
| Addr.listCity | GET | List Cities |
| Addr.listDistrict | GET | List Districts |
| Addr.listArea | GET | List Area [Night Market / Shopping District] |
| Biz.search | GET | Search Businesses |
| Biz.getDetails | GET | Get Details |
| Biz.listReviews | GET | List Reviews |
| Biz.getPhotos | GET | Get Photos |
| Class.listClasses | GET | List all categories by specified category ID |
| Class.listBizInRange | GET | List the business in Range |
| Search.getTopQuery | GET | Get Top Query |
| Theme.getList | GET | List the carefully chosen topic |
| Theme.getDetail | GET | Get Theme Detail |

First, we observed the HTTP request when we called the REST API. There were

⁵Android SDK. URL<<http://developer.android.com/sdk/index.html>>

⁶Wireshark. URL<<http://www.Wireshark.org/about.html>>

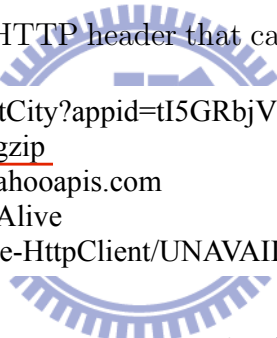
so many REST APIs in the internet. Their format were similar but not the same. In this observation, we choose the Yahoo! LifeStyle API [16] to our observed API server. Table 2.3 shows the method list and the request URI format as follow:

Request URI: `http://tw.lifestyle.yahooapis.com/v0.3/[API_Method]?[Parameters]`

In Yahoo! LifeStyle API, all the API methods use HTTP GET method to retrieve the resources. According to the method list, we would see one method need one URI, so if we want to get the different data from different methods, we need to send more than one HTTP request to call the API. Too many request will produce too many HTTP headers. We wonder to use fewer request to get more results.

Observation 2: HTTP response in REST API

In this observation, we called the method “Addr.listCity” in Yahoo! LifeStyle API, Figure 2.4 was a part of the client HTTP header that captured by WireShark. We set the



```
GET /v0.3/Addr.listCity?appid=tI5GRbjV34Hn0.6F18.....  
Accept-Encoding: gzip  
Host: tw.lifestyle.yahooapis.com  
Connection: Keep-Alive  
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)
```

Figure 2.4: HTTP Request Header from Our Client

“Accept-Encoding” to “gzip” in request header because we wonder the data size is small in transmission. And then, we observed the HTTP response header. If Yahoo! LifeStyle API support Gzip compression, we will see the response header that include “Content-Encoding: gzip” like Figure 2.5. Figure 2.6 shows a part of the response header that cap-

```
HTTP/1.1 200 OK  
Date: Tue, 22 Jun 2010 19:06:53 GMT  
Vary: Accept-Encoding  
Content-Encoding: gzip  
Content-Length: 326  
Content-Type: text/html
```

Figure 2.5: HTTP Response Header that Including Gzip Compression Information

tured by Wireshark. We could observe the the header didn’t include “Content-Encoding: gzip”. The reason is Yahoo! LifeStyle API didn’t support the Gzip [17] compression. We

```
HTTP/1.1 200 OK
Date: Tue, 22 Jun 2010 19:09:28 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", ...
P3P: policyref="http://info.yahoo.com/w3c/p3p.xml", ...
Content-Type: text/xml; charset="utf-8"
X-Cache: MISS from tw.lifestyle.yahooapis.com
Connection: close
Transfer-Encoding: chunked
```

Figure 2.6: HTTP Response Header from Yahoo! LifeStyle API Server

also called the other API such as Twitter API [18], and Yahoo! Knowledge Plus API [19], they didn't support the Gzip compression, too. According to [20], gzipping generally reduces the response size by about 70%. The response data of REST API usually be XML or JSON format. If we compress it, the data size would reduce the response size by about 70%. The compression will reduce the transmission overhead. In addition, the request header "Connection" and "User-Agent" is unnecessary in this case. We would obtain the same result if we didn't set these headers in our request. Furthermore, the response header "P3P" and "X-Cache" is unnecessary for our application, too. We could remove them to save the network bandwidth.

Observation 3: Dependency of API Method

In this observation, we observed the parameters in API methods. We found some parameters for calling method A need method B's result. For example, if we want to list districts of Taipei when we use the Yahoo! LifeStyle API, we need to set the parameter "city=A" because "A" is stand for the id of Taipei in the Yahoo! LifeStyle API. But we need to call the API method "List Cities" to obtain the id of Taipei, so we need to call the API method "List Cities" first to get the city id, and then call the API method "List districts". For this case, we need to send two requests to the API server. More connections will take more network traffics. We wonder fewer connecting when the dependency exist.

Observation 4: Verbose response body

In this observation, we observed the response body from Twitter API server. We considered a scenario about social network application. The friend list is very important in social network application. If we want to list all the friends in our Android application, we always select the most important information about our friends (e.g., friend id, name, image link.) because the Android phone just has small screen to display the information of that. Figure 2.4 shows a part of the method “statuses/friends” response from Twitter API. The API server response the entire result and the content didn’t compress. We wonder to get on-demand results and compress them, but some API method didn’t provide such methods.

```
<?xml version="1.0" encoding="UTF-8"?>
<users type="array">
  <user>
    <id>159557145</id>
    <name>Andriod REST</name>
    <screen_name>AndroidREST</screen_name>
    <location></location>
    <description></description>
    <profile_image_url>http://s.twimg.com/a/1282351897/images/default_profile_6_normal.png</profile_image_url>
    <url></url>
    <protected>>false</protected>
    <followers_count>1</followers_count>
    <profile_background_color>9ae4e8</profile_background_color>
    <profile_text_color>000000</profile_text_color>
    <profile_link_color>0000ff</profile_link_color>
    <profile_sidebar_fill_color>e0ff92</profile_sidebar_fill_color>
    <profile_sidebar_border_color>87bc44</profile_sidebar_border_color>
    <friends_count>2</friends_count>
    <created_at>Fri Jun 25 18:23:06 +0000 2010</created_at>
    <favourites_count>0</favourites_count>
    <utc_offset></utc_offset>
    <time_zone></time_zone>
    <profile_background_image_url>http://s.twimg.com/a/1282351897/images/themes/theme1/bg.png</profile_backgr...
    <profile_background_tile>>false</profile_background_tile>
    :
  </status>
</user>
```

The Information we need.

The Information we don't care in our app.

Figure 2.7: Result of Call “statuses/friends” from Twitter API Server

Observation 5: Get Images From API

In this observation, we observed the API method about images. Social networking websites usually have photo album service. Using their APIs to get the photos just get the image links. If our Android application want to get the image files, we must using the image links to retrieve the image files. Figure 2.8 shows the normal flow when we use the API method to get the photos. Figure 2.9 shows the packets we captured by

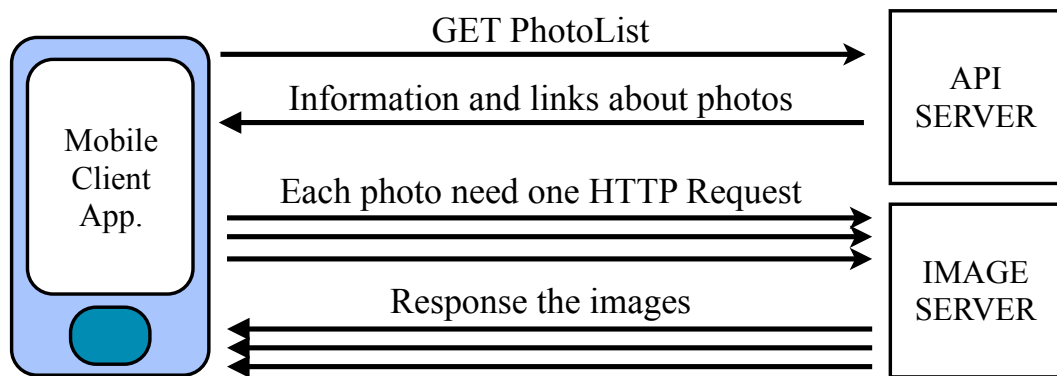


Figure 2.8: Normal flow to use API to get photos

WireShark. If we want to get many images, that will produce many HTTP connections (HTTP requests and responses). It will produce many redundant headers. If we want to

```
GET /_drMirRQ74-A/S_K_NEXfQfi/AAAAAAAABX4/-TBPLzf2Bho/s72/compass.jpg HTTP/1.1
HTTP/1.1 200OK (JPEG JFIF image)
GET /_drMirRQ74-A/S_K_NUsBcjl/AAAAAAAABX8/p49JRTUZIyE/s72/corbeillevidesz.jpg HTTP/1.1
HTTP/1.1 200OK (JPEG JFIF image)
GET /_drMirRQ74-A/S_K_NKRbixI/AAAAAAAABX0/HiQaetH-0Uc/s288/digg-1.jpg HTTP/1.1
HTTP/1.1 200OK (JPEG JFIF image)
```

Figure 2.9: Per Image Need Per Request and Response Pair

develop an Android application about album, we usually list a part of photos in small size, and then, select the photo we want to see the full size because the small screen size. So, if the image server return a big size image, that will waste a lot of bandwidth. Figure 2.10 shows a simple Picasa⁷ Android Application. The Picasa RSS (Really Simple Syndication) API just provides the thumbnail album cover images that size is 160 x 160 pixels. The Android HVGA⁸ (Half-size VGA) screen resolution is 480 x 320 pixels. It is hard to

⁷Picasa. URL<<http://picasaweb.google.com.tw>>

⁸HVGA from Wikipedia. URL<<http://en.wikipedia.org/wiki/HVGA>>

show all the albums without scrolling. This Simple Android Application downloads all the album cover images and show them in size 60 x 60 pixels. It would need 12 requests to download 12 images and cause 78 KB network traffic. We wonder to get the multiple photos by one request and the image size can be controlled.

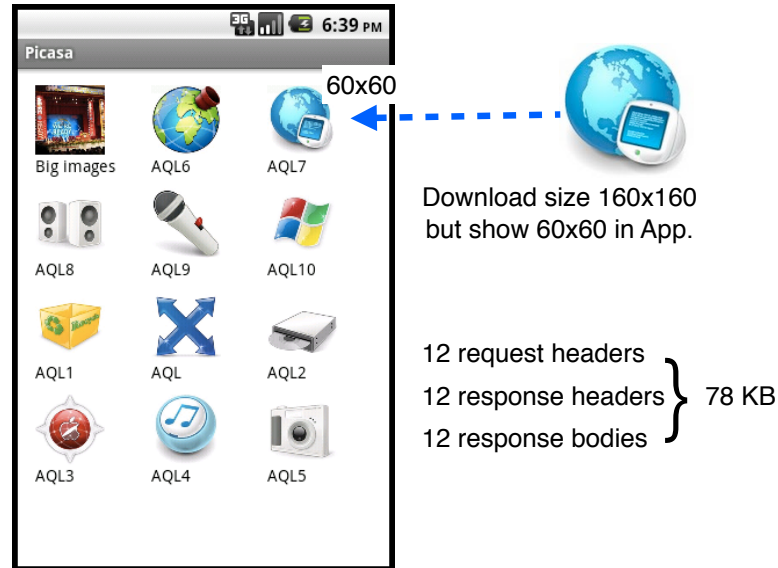


Figure 2.10: Sample Picasa Android Application

Summary of Observations

According to our observations, we could list the requirements of our system as follow:

- (1) We wonder to reduce the request and response header size.
- (2) We wonder to use fewer request to get more results.
- (3) We wonder to get more smaller response body by compression.
- (4) We wonder fewer connecting when the dependency exist.
- (5) We wonder to get on-demand results.
- (6) We wonder to get multiple photos by one request and the image size can be controlled.

The above-mentioned six system requirements with the purpose of reducing the transmission overhead between Android application and RESTful APIs.

Chapter 3

System Architecture

3.1 Proposed System Architecture

According to our motivation and system requirements that mentioned in Chapter 2, we proposed a system architecture that satisfied them. Figure 3.1 shows our system architecture. We setup a proxy between the Android application and the API server in place of logic flow (direct connect with API server). The Android application include

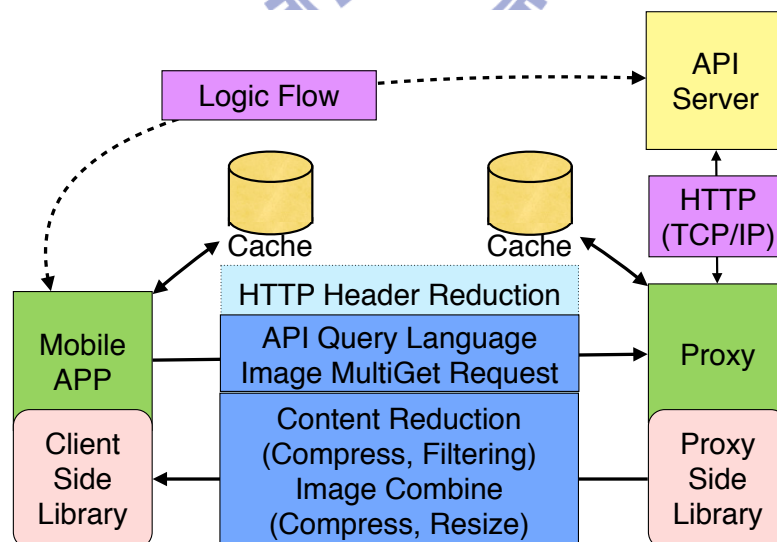


Figure 3.1: Proposed System Architecture

the Client-Side Library that interconnect with Proxy-Side Library in the proxy. Android application call the REST API through the proxy. The proxy interconnect with API Server through wireline network (strong connection). Client and Proxy-Side can option

enable cache service if your Application need better response time. We do the HTTP Header reduction between the Client and Proxy-Side connection. The details and other functions we will explain in follow sections.

3.2 HTTP Header Reduction

We set the HTTP request and response header as simple as possible. For example, Figure 2.4 in Observation 2 shows the HTTP request header that we call the REST API. The header “Host” and “Accept-Encoding: gzip” is necessary for our request, but the “User-Agent” is unnecessary because we will get the same result if we don’t set it. We can add the header “Accept-Encoding: gzip” and the parameter “appid” in the Proxy-Side to reduce the traffic from client to server. And then, we simplify the HTTP response header by our Proxy-Side Library. Figure 3.2 shows our headers. We removed the “P3P” information and added the compression information.

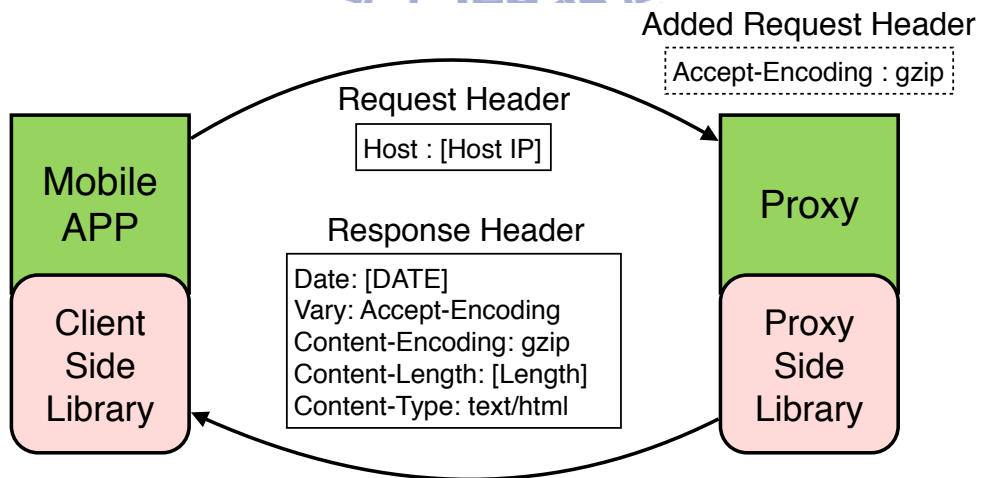


Figure 3.2: Simplified HTTP Request and Response Header

3.3 Client-Side Library

In this section, we will describe the details about our Client-Side Library.

3.3.1 API Query Language

The API Query Language(AQL) is a SQL-Style Language for our Request. It transform one or multiple REST call to the one SQL-Style query. Table 3.1 shows the query format of AQL and relationships between AQL and HTTP verbs.

Table 3.1: Query Format of AQL and Relationships Between AQL and HTTP Verbs

| Action | AQL | HTTP | Query Format |
|--------|--------|--------|---|
| Create | Insert | POST | INSERT INTO <i>[API].[Method]</i> (<i>[Parameter Key(s)]</i>) VALUES (<i>[Parameter Value(s)]</i>) |
| Read | Select | GET | SELECT <i>[Field]//[Attribute]</i> FROM <i>[API].[Method]</i> WHERE A or B A: <i>[Key-Value-Parameters]</i> B: <i>[Key] IN ([SubSelect])</i> |
| Update | Update | PUT | UPDATE <i>[API].[Method]</i> SET <i>[New-Value]</i> WHERE <i>[Key-Value-Parameters]</i> |
| Delete | Delete | DELETE | DELETE FROM <i>[API].[Method]</i> WHERE <i>[Key-Value-Parameters]</i> |

Insert

The “Insert” query is to create a resource to the API provider. Developer assign a specific API name, method and required parameters to the query. For example, developer can use query “INSERT INTO Twitter.statuses/update (status) VALUES (Tweet)” to send a status update to Twitter through our proxy. When our Proxy-Side Library

receive the query, the query will be parsed into real REST URI for Twitter API such as “http://api.twitter.com/version/statuses/update.xml?status=Tweet”¹. Proxy use the HTTP POST to access the resource and then return the result to the Client-Side Library. If the API method need any authentication such as OAuth. Developer must direct user to Service Provider to obtain user authorization first. User need to allow Proxy-Side Library to get the Access Token. The Authentication Flow is show in Figure 2.2. And then, Server will have permission to access the resource of service provider.

Select

The “Select” query is use to get resource from API server. It is always the most query that an application call. To “Select” the specific field is useful to get on-demand resource. Figure 3.3 shows the result of query “SELECT id, name FROM Twitter.statuses/friends”². Method “statuses/friends” will response the information about friends of Twitter user.

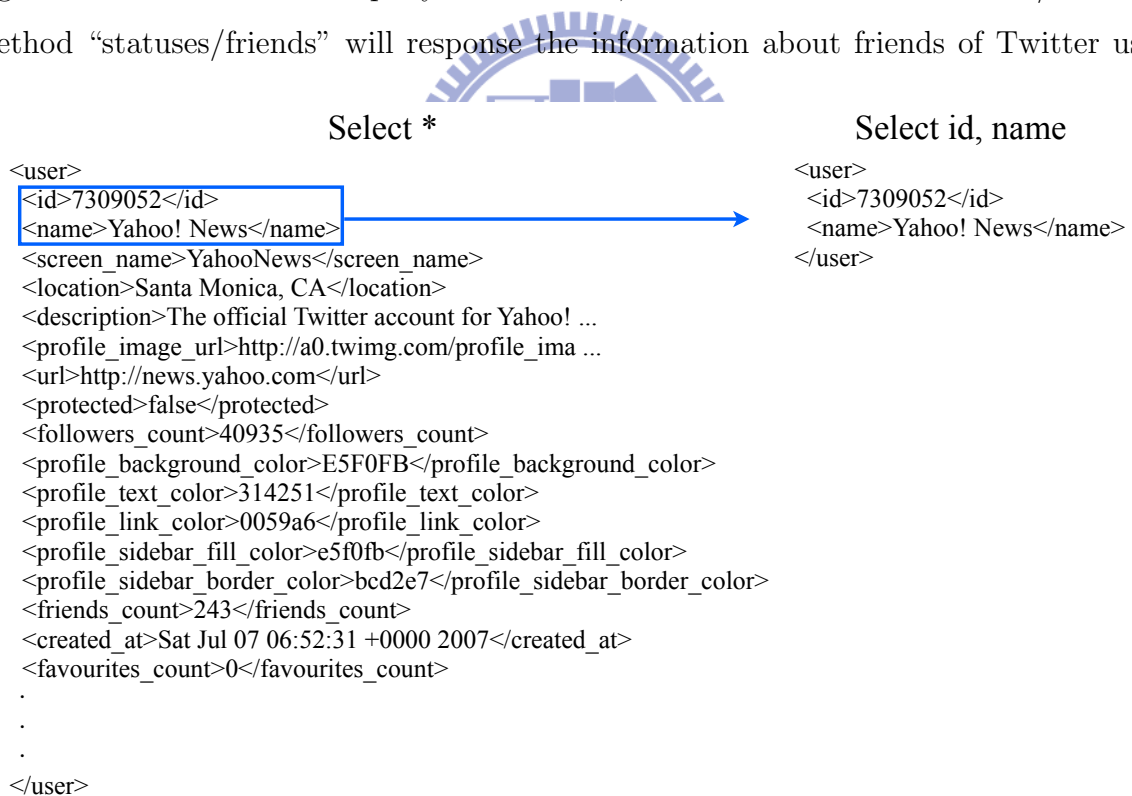


Figure 3.3: “Select *” and “Select id, name” From Twitter.statuses/friends

The information include the background color of friends’ Twitter page. When we develop

¹POST statuses/update. URL<<http://dev.twitter.com/doc/post/statuses/update>>

²GET statuses/friends. URL<<http://dev.twitter.com/doc/get/statuses/friends>>

an Android application, we usually have our user interface. So we don't care about the color or image of their background image information. In this case, we can know all of our Twitter friends' id, name and latest status id by AQL. It can save a lot of bandwidth. We also can use "SubSelect" to solve the problem of "Dependency of API Method" because we can send one query to the Proxy-Side Library, and then, Proxy-Side Library parse the query and call the specific API respectively to complete the result of query. Figure 3.4 shows the flow of SubSelect query. Just like YQL, we can use AQL to call the method of API and set the parameter to result from another API. The "Select" query can retrieve the XML of RSS feed by query "SELECT * FROM xml where url=[RSS_URI]". The "Select" query also can retrieve the value from XML attributes. For example, "<media:thumbnail url='http://example.jpg' />" can use "SELECT media:thumbnail//url" to get the value "http://example.jpg".

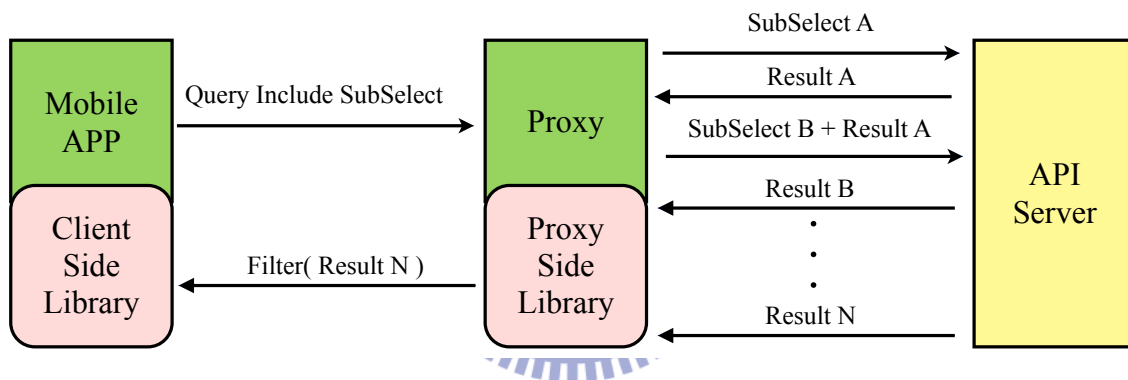


Figure 3.4: Flow of SubSelect Query

Update

The "Update" query is used to update the resource by REST API call. Android application send the "Update" query through Client-Side Library. When the Proxy-Side Library received the query, the query would be parsed and using HTTP "PUT" or "POST" method to send a update request to the API server.

Delete

The "Delete" query is used to delete the resource by REST API call. Android application sends the "Delete" query through Client-Side Library. When Proxy-Side

Library received the query, the query would be parsed and using HTTP “POST” or “DELETE” method to send a delete request to the API server.

Summary of AQL

Developers can use the AQL or not by their preferences because in four AQL query types, the “Select” query is more effectivity than others. AQL provides a friendly way to call the REST API. Facebook provides their own SQL-Style language called FQL but it just providing “Select” Query. We also can implement multi-query in our Client-Side Library if we want to send more than one query by one HTTP request. For example, if users want to send their message to Twitter, Facebook or other Web sites, or upload photos to different album Web sites (e.g., Flickr, Picasa), they can bundle multiple “Insert” query to one HTTP request. Our Proxy-Side Library will do all the queries for users.

3.3.2 Image Multi-Get Module

For the purpose of minimizing HTTP requests [20], CSS Sprites³ are the preferred method for reducing the number of image requests. Combine your background images into a single image and use the CSS background-image and background-position properties to display the desired image segment. We adopted this idea for our system function. Figure 3.5 shows two implement of Picasa Android Application. For this case, App. A is normal design that download all the album cover images and show them in size 60 x 60 pixels. It would need 12 requests to download 12 images and cause 78 KB network traffic. If we implement it like App. B, we resized all images and combined them to one images first, it just need one request to get all the album cover images and just cause 28 kb network traffic. It is 64% reduction of this case. For the purpose of reducing the network traffic of getting images, the “Image Multi-Get” (IMG) module is provides a multiple image getting method to reduce the request header, compress the images and resize the images. Figure 3.6 is shows the work flow of Image Multi-Get Module. Just like the idea of CSS-Sprite, Client-Side send one request to get multiple images to reduce the HTTP

³CSS Sprites: Image Slicing’s Kiss of Death. URL<<http://www.alistapart.com/articles/sprites>>

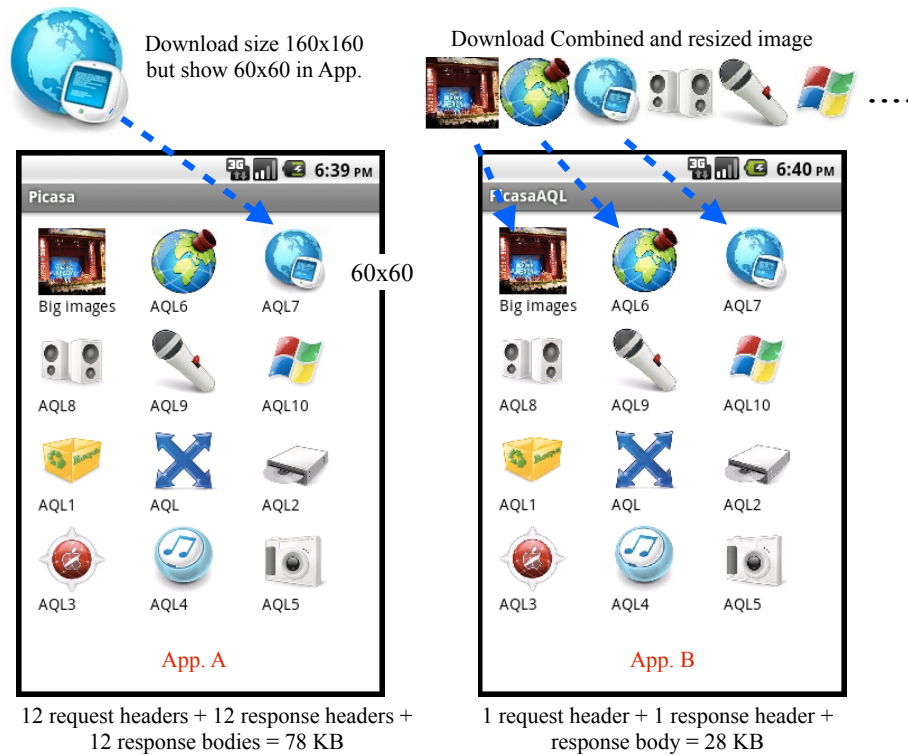


Figure 3.5: Two Implement of Picasa Android Application

request headers. We can send a set of image urls or get multiple urls from API Server by AQL to the IMG module as Step 1 in the Figure 3.6. If we just send a set of image urls, the Proxy-Side will download all the images according to the urls. And then, the Proxy-Side Library combines all the images to one image. The image size for each image will add to response header. For example, response header “Size: 24562_45231_64523” is stand for three images size. First image size is 24562 bytes, the second image is 45231 bytes and the third image is 64523 bytes. When the Client-Side Library received the combined image, it will spilt the image according to the response header. If we want to get the urls from the result of AQL, we can use the term “[IMG]” to indicate the Proxy-Side Library to get the urls from API Server, and then send the urls to the IMG module to combine the images and return to Client-Side Library. For example, we use query “SELECT [IMG].Url FROM LifeAPI.Biz.getPhotos WHERE ID=LB4VVSXM38511” to get the image urls of result of query “SELECT Url FROM LifeAPI.Biz.getPhotos WHERE ID=LB4VVSXM38511” and then send the urls to IMG module to get the combined image. For the combined image, we have three optional parameters to set:

- (1) Optional Quality Parameter:

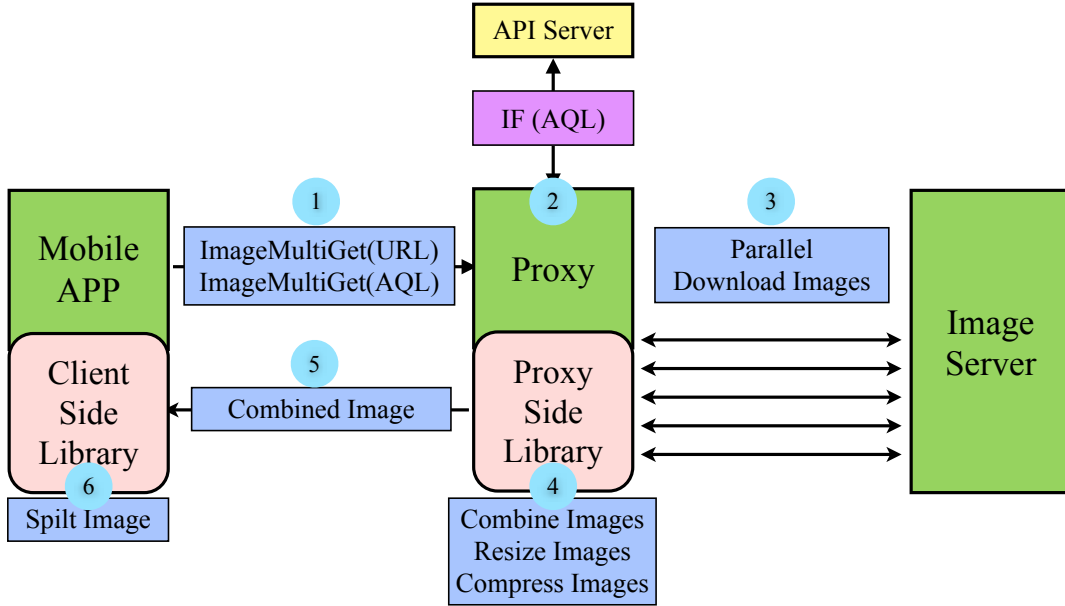


Figure 3.6: Image Multi-Get Module

If the quality parameter is set, the combined image will be compressed by JPEG. The quality parameter range is 0 to 100 percentage. For example, if our Android Application want to get the 80% quality images, we can use AQL to set the “SELECT” statement to “SELECT [IMG].[80].TagName_of_Url” to get the 80% quality images from IMG module. When we compress the image, the total bytes of image will be reduced, too. There is a trade-off between the image quality and total bytes. Developers can according to their application requirement to set the quality parameter.

(2) Optional Resize Rate:

If the resize rate is set, the combined image will be resize by the given rate. The resize rate range is 1 to 100 percentage. For example, if our Android Application want to get a size is 50% of original image size and quality is 90%, we can use AQL to set “SELECT” statement to “SELECT [IMG].[90].[50].TagName_of_Url” to get the 90% quality and 50% size images from IMG module. When the image is resized to smaller size, the total bytes of image will become smaller.

(3) Optional Resize To Fixed Size:

If our application want to get a set of fixed size images, we can set the AQL “SELECT” statement to “SELECT [IMG].[100].[100*100].TagName_of_Url” to resize an

original image to 100 x 100 pixels. It is useful for a photo album application to set a small size of thumb cover like Figure 3.5.

The “IMG” module is based on PHP GD library ⁴. Before we combine the images, we use PHP GD function “ImageCreateTrueColor” to create a new true color image. And then, use PHP GD function “ImageCopyResized” to resize the original image by given resize rate. Moreover, we use PHP GD function “ImageJpeg” to assign a quality parameter and then save a result image to the buffer and record the total bytes of this image. Finally, when all the images had been buffered, the images in buffer will be merged and returned to client. The response header will add the size information about all the images.

3.3.3 Gzip Compression Request

Our Client-Side Library extends `DefaultHttpClient`⁵ and add a response interceptor to support the Gzip decompression. The compression header “Accept-Encoding: gzip” is going to be added in the proxy, so we do not need to add it in our Client-Side Library. Figure 3.7 is request and response header of the query “Select * from LifeAPI.Addr.listCity”.

AQL HTTP Request Header

```
GET /aql/doQuery.php?q=select+*+from+LifeAPI.Addr.listCity HTTP/1.1
Host: 140.113.240.106
```

AQL HTTP Response Header

```
HTTP/1.1 200 OK
Date: Sun, 08 Aug 2010 18:52:53 GMT
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length 326
Content-Type: text/html
```

Figure 3.7: AQL HTTP Request and Response Header

⁴PHP GD URL<<http://www.php.net/manual/en/book.image.php>>

⁵DefaultHttpClient. URL<<http://developer.android.com/reference/org/apache/http/impl/client/DefaultHttpClient.html>>

3.3.4 Spilt Combined Image

When the Client-Side Library get the combined image, the combined image will be spilt to multiple images according to the response header “Size” that added by Proxy-Side Library. For example, if the response header is “Size: 4310_2394_2914”, we will know there are three images, first image is 4310 bytes, second image is 2394 bytes and third image is 2914 bytes. Client-Side Library can cut the input stream to images according to byte length information in response header. The multiple images will option be a Bitmap⁶ array or another style that Android developer like to use.

3.4 Proxy-Side Library

In this section, we will describe the details about our Proxy-Side Library. Our Proxy-Side Library was written by PHP and based on Apache HTTP Server. The Client-Side Library use the HTTP “GET” or “POST” to send the query to our Proxy-Side Library.

3.4.1 Parsing the Query

The query will be parse to url of resource that use to access the API server. For example, the query “INSERT INTO API.Method ($p_1, p_2, p_3, \dots, p_n$) VALUES ($v_1, v_2, v_3, \dots, v_n$)” will parse to url “http://Uri.Of.API/Method? $p_1 = v_1 \& p_2 = v_2 \& p_3 = v_3 \& \dots \& p_n = v_n$ ” and use HTTP POST method to access it. If the “SELECT” query includes the sub select query, it will be parsed and be processed recursively.

3.4.2 Filtering Results

When our proxy got the results from the API server, our proxy will filter the result by specific fields in the “SELECT” query. If the AQL query is start by “SELECT *”, our proxy will return the entire result. If proxy got the result in XML format, the white space between different XML tag will be removed. The comments of XML will be removed,

⁶Bitmap. URL<<http://developer.android.com/reference/android/graphics/Bitmap.html>>

too. Finally, the first line of XML (`<?xml version="1.0" encoding="UTF-8" ?>`) will be removed because we can add it in our Client-Side Library.

3.4.3 Convert Response Format

If there exists a parameter “format” in the “SELECT” query, our proxy will convert the format to the related value. For example, If the service provider just has XML format to response, we can add the parameter “format=JSON” after “WHERE” in AQL “SELECT” query. Our Proxy-Side Library will convert the format to JSON format. JSON format can convert to XML, too.

3.4.4 Combining Images

If our AQL parser meet term “[IMG]” is “SELECT” query, the IMG module will be launch. Proxy will download all the images by parallel scheme (multi-thread). The IMG query format is

SELECT[IMG].[Quality].[Resize].TagName_of_Url//AttributeName_of_Url

. When proxy download all the images, proxy will combine all images into one byte array output stream and set the HTTP response header “Size” that record all the image’s byte length. Finally, return the combined image to the client.

3.4.5 OAuth Authentication

Our proxy supports OAuth authentication if users want to get their private data through our proxy. The authentication flow had been shown in Figure 2.2. Figure 3.8 shows the example about the flow of OAuth authentication in the simple Twitter Android application. When user allows our proxy to access their private data, we will get their access token and then return to Client-Side Library. Client-Side can save this access token and use this access token to send requests to proxy.

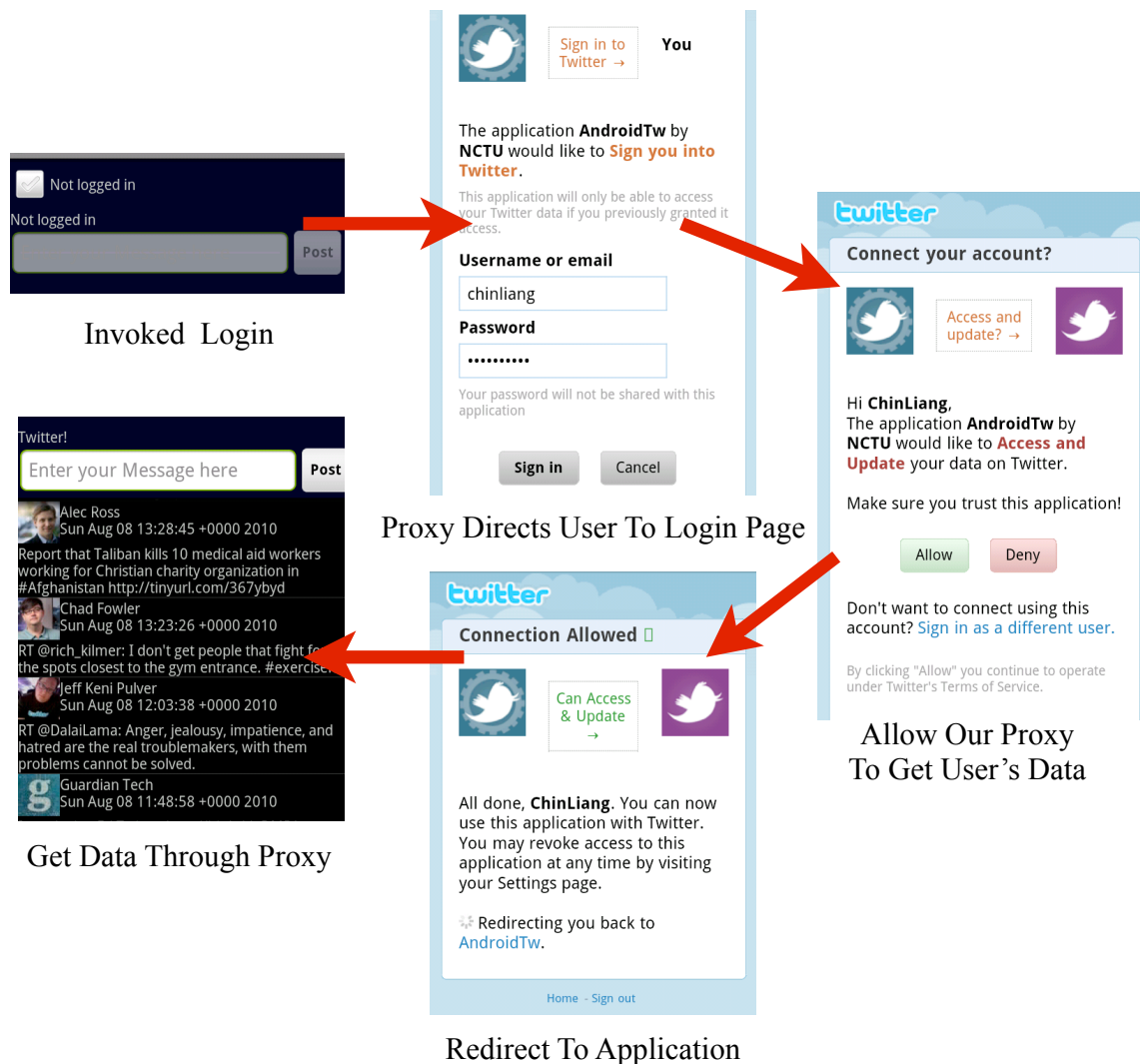


Figure 3.8: Example of OAuth Authentication

Chapter 4

Experimental Results

In this chapter, we will show our experimental results in following sections. First, we show the results that we call the Yahoo! LifeStyle API by different ways. One is to get results from the API server directly and another is through our proxy. We also show the result of different formats (XML and JSON). Second, we compared the two different implementations about common Picasa album application.

4.1 Experimental Setup

Our client side device is “HTC Desire A8181”¹. The specifications show in Table 4.1.



Figure 4.1: HTC Desire A8181

Our client side device use Wi-Fi to access the RESTful APIs and capture the packets by Wireshark. In order to simulate the low-bandwidth wireless network, our Wi-Fi access point was connecting a wireline that just has 2M downlink and 128k uplink. Our proxy

¹HTC Desire Specification<<http://www.htc.com/www/product/desire/specification.html>>

Table 4.1: Specifications of Our Client Device (HTC Desire A8181)

| | |
|-----------------|---|
| Model | A8181 |
| CPU Speed | 1 GHz |
| Platform | Android TM 2.1 (Eclair) with HTC Sense TM |
| Memory | 512 MB ROM and 576 MB RAM |
| Size and weight | 119 x 60 x 11.9mm, 135 grams |
| Network | HSPA/WCDMA (900/2100 MHz) GSM/GPRS/EDGE (850/900/1800/1900 MHz) |
| Display | 3.7-inch AMOLED touch-sensitive screen |
| Resolution | 480 X 800 WVGA |
| Internet | 3G: Up to 7.2 Mbps download speed Up to 2 Mbps upload speed GPRS: Up to 114 kbps downloading EDGE: Up to 560 kbps downloading Wi-Fi TM : IEEE 802.11 b/g |

was implemented with PHP Version 5.3.2-1ubuntu4.2 on Ubuntu 10.04.1 LTS and the memory size is 4GB, processor is “AMD Athlon(tm) II X4 630 Processor 2. 8 GHz”.

4.2 Common Plain Text

In the section, we choose the Yahoo! LifeStyle API to show our improvement. The response of RESTful APIs always be plain text just like Yahoo! LifeStyle API. Our experimental application will normally call the Yahoo! LifeStyle API with the parameters that show in Table 4.2. We use the same parameters to call the Yahoo! LifeStyle API by AQL. Table 4.3 is shows our reduce ratio. The original size is to call the method normally. The fields of content received is the size of response content (including response header length). The fields of Total is all the transmission traffic that including TCP three-way handshake (about 200 bytes), TCP ACK (per packet 66 bytes), HTTP request headers and response headers. The original HTTP request header length in this experiment is about 305 358 bytes. Our reduced HTTP request header length is about 180 to 245 bytes.

Table 4.2: Experimental Parameters in Yahoo! LifeStyle API

| Method Name | Parameters |
|----------------------|--|
| Addr.listCity | (NULL) |
| Addr.listDistrict | city=A |
| Addr.listArea | city=A&dist=02 |
| Biz.search | BizName=coffee&address=&page=1 |
| Biz.getDetail | ID=N96KJRS68418 |
| Biz.listReviews | ID=N97K68S30418&begin=0&limit=100 |
| Biz.getPhotos | ID=P87YBWS38512&begin=1&limit=100 |
| Class.listClasses | id=0 |
| Class.listBizInRange | lon=121.548030&lat=25.036608&class=152979953 |
| Search.getTopQuery | (NULL) |
| Theme.getList | count=20 |
| Theme.getDetail | id=30 |

The compress ratio was calculated by the formula:

$$CompressRatio = 1 - ContentReceived(AQL) / ContentReceived(Original)$$

The total reduce ratio was calculated by the formula:

$$TotalReduceRatio = 1 - Total(AQL) / Total(Original)$$

We can see the average compress is about 74% because the gzipping is generally reduces the response size by about 70%. In addition, the total reduce ratio is over 60% for average. Figure 4.2 is a bar chart of this experiment. The reduce ratio of method “Search.getTopQuery” is only 44.12% because the response of this method is small (825 bytes by original). Table 4.4 is the data of the same API method but the response format is JSON. We can see the size of JSON format usually small than the XML format because JSON format is lightweight data-interchange format. But the original response of method “Theme.getDetail” in JSON format is larger than XML format. The reason is the content of the Yahoo LifeStyle API has many Chinese words. When JSON encode the Chinese words, it will become a long code (e.g., “美食” after json encode is “\u7f8e\u98df”).

We observed the response time in two types of API calling. The response time is start from sending the request and displaying on screen in the end. We set the format to XML

Table 4.3: Ratio of XML Content Compressed by Gzip in LifeStyle API (Bytes)

| Method Name | Content Received (Original) | Content Received (AQL) | Total (Original) | Total (AQL) | Compress Rate (HTTP) | Compress Rate (TCP) | Total Reduce Size |
|----------------------|-----------------------------|------------------------|------------------|-------------|----------------------|---------------------|-------------------|
| Addr.listCity | 3,143 | 491 | 4,446 | 1,261 | 84.38% | 71.64% | 3,185 |
| Addr.listDistrict | 2,044 | 398 | 3,226 | 1,187 | 80.53% | 63.21% | 2,039 |
| Addr.listArea | 2,988 | 586 | 4,306 | 1,383 | 80.39% | 67.88% | 2,923 |
| Biz.search | 7,697 | 2,515 | 9,094 | 3,461 | 67.32% | 61.94% | 5,633 |
| Biz.getDetail | 2,438 | 779 | 3,626 | 1,574 | 68.05% | 56.59% | 2,052 |
| Biz.listReviews | 7,678 | 1,999 | 9,084 | 2,953 | 73.96% | 67.49% | 6,131 |
| Biz.getPhotos | 3,182 | 620 | 4,519 | 1,440 | 80.52% | 68.13% | 3,079 |
| Class.listClasses | 1,926 | 501 | 3,106 | 1,288 | 73.99% | 58.53% | 1,818 |
| Class.listBizInRange | 4,321 | 929 | 5,479 | 1,768 | 78.50% | 67.73% | 3,711 |
| Search.getTopQuery | 825 | 232 | 1,869 | 1,007 | 71.88% | 46.12% | 862 |
| Theme.getList | 8,410 | 3,055 | 9,788 | 4,106 | 63.67% | 58.05% | 5,682 |
| Theme.getDetail | 17,886 | 3,988 | 20,187 | 5,038 | 77.70% | 75.04% | 15,149 |
| Overall | 62,538 | 16,093 | 78,730 | 26,466 | 74.27% | 66.38% | 52,264 |

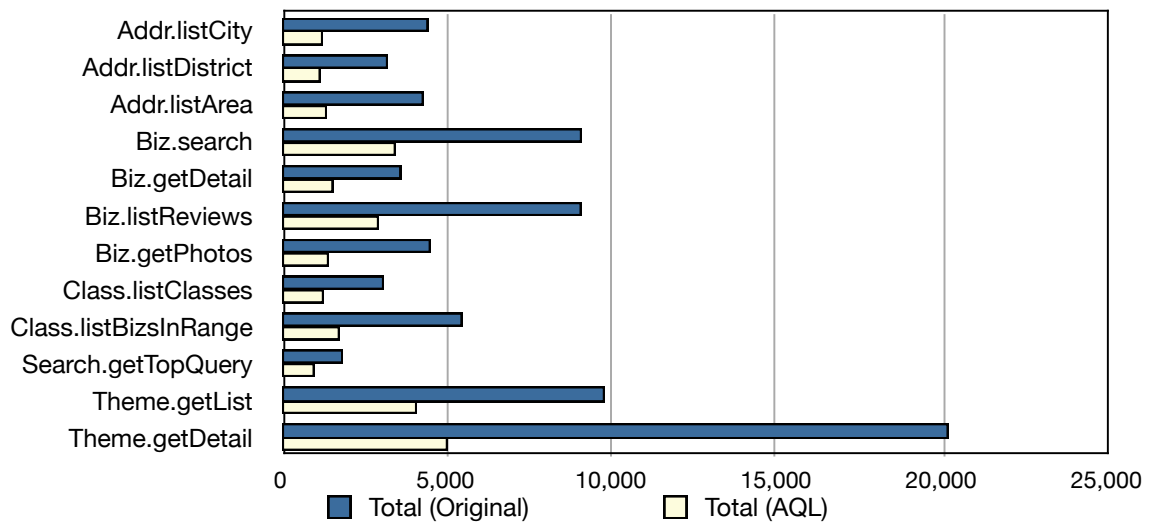


Figure 4.2: Gzip Compression in LifeStyle API (XML)

Table 4.4: Ratio of JSON Content Compressed by Gzip in LifeStyle API (Bytes)

| Method Name | Content Received (Original) | Content Received (AQL) | Total (Original) | Total (AQL) | Compress Rate (HTTP) | Compress Rate (TCP) | Total Reduce Size |
|----------------------|-----------------------------|------------------------|------------------|-------------|----------------------|---------------------|-------------------|
| Addr.listCity | 1,602 | 464 | 2,785 | 1,254 | 71.04% | 54.97% | 1,531 |
| Addr.listDistrict | 1,171 | 377 | 2,233 | 1,182 | 67.81% | 47.07% | 1,051 |
| Addr.listArea | 2,052 | 583 | 3,250 | 1,396 | 71.59% | 57.05% | 1,854 |
| Biz.search | 7,505 | 2,618 | 8,914 | 3,580 | 65.12% | 59.84% | 5,334 |
| Biz.getDetail | 1,588 | 684 | 2,788 | 1,495 | 56.93% | 46.38% | 1,293 |
| Biz.listReviews | 5,910 | 1,985 | 7,196 | 2,955 | 66.41% | 58.94% | 4,241 |
| Biz.getPhotos | 2,261 | 558 | 3,478 | 1,394 | 75.32% | 59.92% | 2,084 |
| Class.listClasses | 1,355 | 469 | 2,415 | 1,272 | 65.39% | 47.33% | 1,143 |
| Class.listBizInRange | 2,929 | 893 | 4,297 | 1,748 | 69.51% | 59.32% | 2,549 |
| Search.getTopQuery | 698 | 229 | 1,754 | 1,024 | 67.19% | 41.62% | 730 |
| Theme.getList | 9,697 | 3,300 | 11,219 | 4,367 | 65.97% | 61.07% | 6,852 |
| Theme.getDetail | 18,133 | 4,184 | 20,446 | 5,250 | 76.93% | 74.32% | 15,196 |
| Overall | 54,901 | 16,344 | 70,775 | 26,917 | 70.23% | 61.97% | 43,858 |

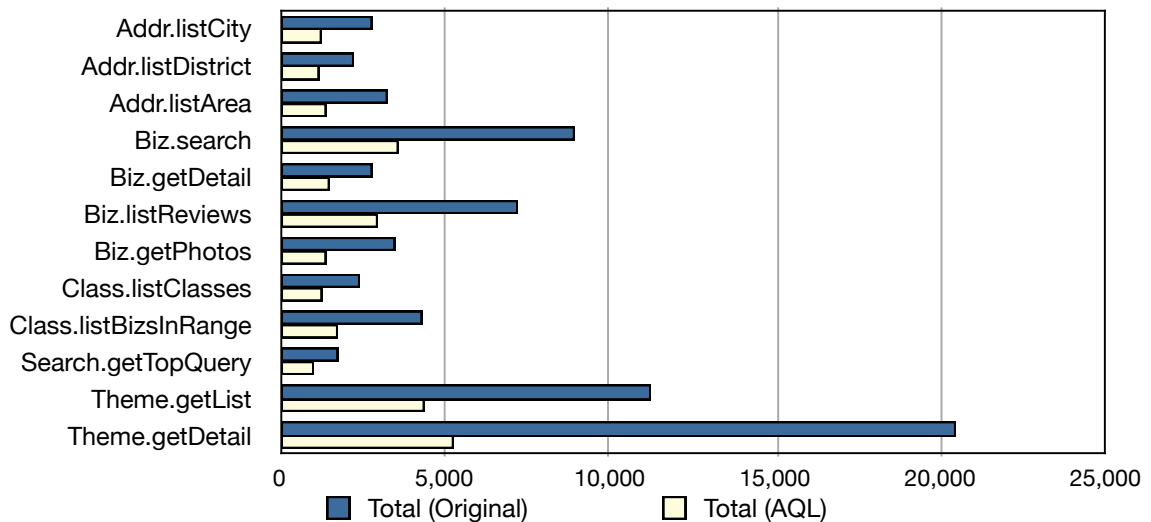


Figure 4.3: Gzip Compression in LifeStyle API (JSON)

and JSON and then called all the methods one hundred times respectively. Table 4.5 shows the average response time in Yahoo! LifeStyle API. The bold number is stand for “more fast”. We can found that we call API by AQL is more fast than call it directly,

Table 4.5: Average Response Time in Yahoo! LifeStyle API (ms)

| Method Name | XML | | JSON | |
|----------------------|---------------|---------------|---------------|---------------|
| | Directly | AQL | Directly | AQL |
| Addr.listCity | 710.67 | 666.88 | 569.20 | 672.05 |
| Addr.listDistrict | 642.24 | 636.02 | 629.54 | 617.28 |
| Addr.listArea | 667.81 | 650.30 | 633.03 | 633.31 |
| Biz.search | 996.39 | 706.50 | 715.64 | 671.55 |
| Biz.getDetail | 646.60 | 656.11 | 626.11 | 623.76 |
| Biz.listReviews | 979.13 | 690.93 | 916.66 | 619.21 |
| Biz.getPhotos | 655.08 | 660.99 | 639.76 | 642.24 |
| Class.listClasses | 643.52 | 638.61 | 635.34 | 623.32 |
| Class.listBizInRange | 967.16 | 664.23 | 646.59 | 641.07 |
| Search.getTopQuery | 619.92 | 612.58 | 611.77 | 615.19 |
| Theme.getList | 982.86 | 674.66 | 993.36 | 676.78 |
| Theme.getDetail | 1,317.58 | 738.82 | 1,239.19 | 648.73 |
| Average | 819.08 | 666.39 | 738.02 | 640.37 |

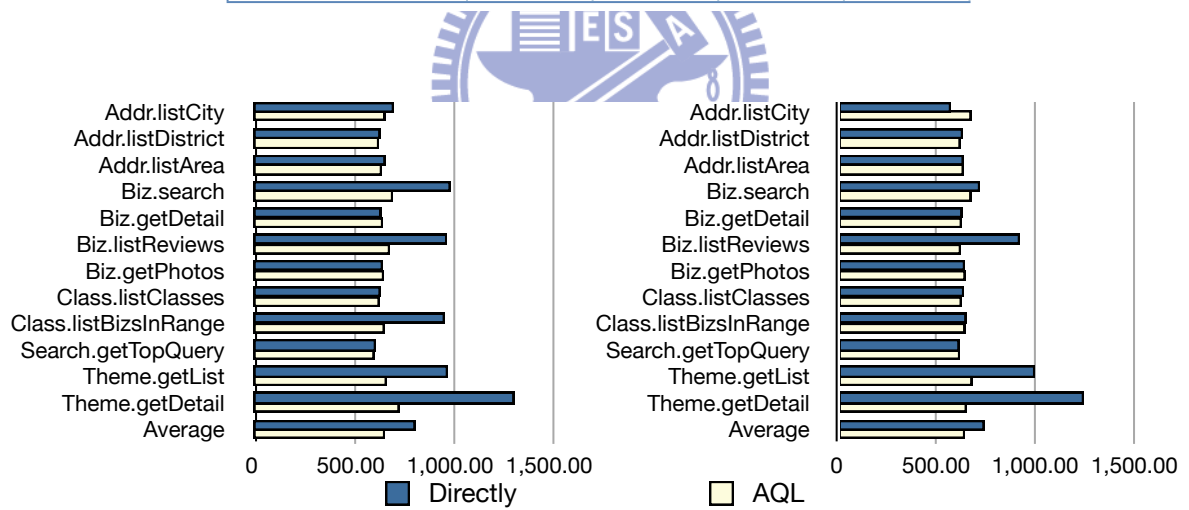


Figure 4.4: Line Chart of Average Response Time in Yahoo! LifeStyle API (ms)

especially in large data. The reason is large data will be reduce more bytes by Gzip compression. So it needs less time to transfer. According to Table 4.5, AQL is more efficient because the average time is less then another.

4.3 Multiple Images

In this section, we will show the effect when we using “IMG” module or not. First, we observed the impact between quality and total image size. Second, we observed the effect of resize ratio. And then, we compared different applications’ response time and total packet length in the end .

4.3.1 Image Quality

The image quality and the total image size is a trade off. More smaller image size is less quality. We choose ten images to observed the quality parameter in “IMG” module. The size of original image is 160 x 160. Table 4.6 shows the quality parameter related to the total image size (including TCP packet data). The “Original” quality is the images

Table 4.6: Quality Parameter versus Total Image Size

| Quality | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 | 20 | 10 |
|-------------|---------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| Total Bytes | 142,714 | 68,385 | 49,278 | 41,150 | 35,682 | 32,185 | 28,809 | 25,000 | 20,399 | 14,555 |

that we do not do any compression and download it in normal HTTP connection (one connection per image). Quality in Figure 4.5 (a) shows the line chart of Table 4.6. We found that the total size of combined image is large than the sum of all original size when the quality parameter is “100” because the combined image is TrueColor image. When we set the quality parameter to “90”, the total image size is decrease very soon.

4.3.2 Image Resize

The total image size is proportional to resize rate. We can resize a big image to a smaller one to reduce the total image size. We choose the same images as “Image Quality” experimental. In this experimental, we set the “Quality parameter” to “100”. Table 4.6 shows the resize rate related to the total image size. Resize in Figure 4.5 (b) shows the line chart of Table 4.7.

Table 4.7: Resize Rate versus Total Image Size

| Resize Rate | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 | 20 | 10 |
|-------------|---------|---------|---------|---------|--------|--------|--------|--------|--------|-------|
| Total Bytes | 168,573 | 150,029 | 122,900 | 105,444 | 84,138 | 63,598 | 47,266 | 32,283 | 18,716 | 7,944 |

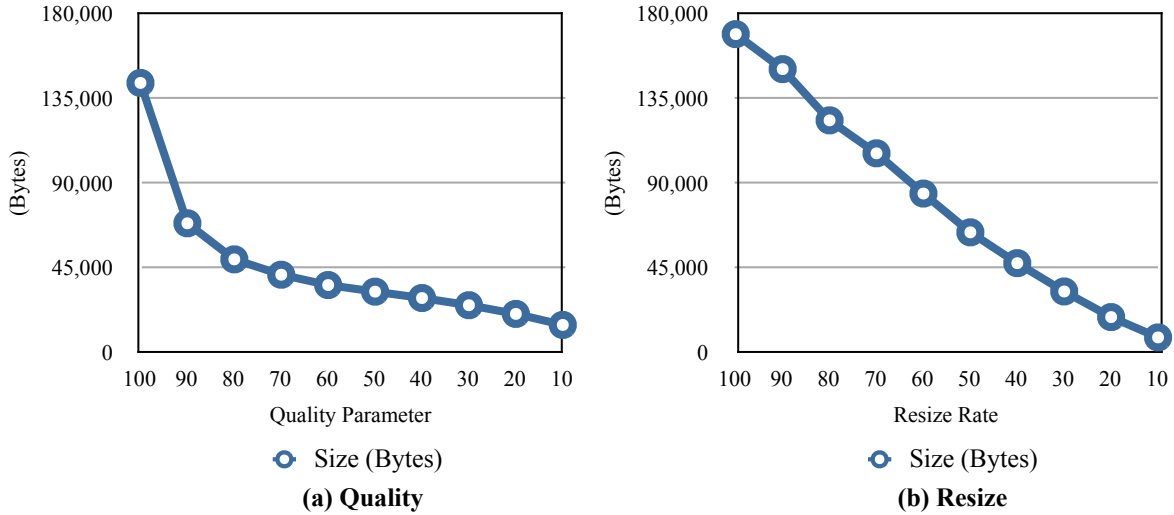


Figure 4.5: Line Chart of Quality Parameter and Resize Rate versus Total Image Size

4.3.3 Normal versus AQL Picasa Application

We implemented the Picasa album application normally and by AQL respectively. Figure 4.6 is a screen capture of our application. User enter a user ID and then click the “Enter” button, and then it will show the album list of that user’s ID. The normal Picasa application get the entire RSS from Picasa album list without compression. And then, parse the XML to get the album title and cover image urls. Finally, download all the “on screen”² images. On the other hand, AQL Picasa application (PicasaAQL) one set the quality parameter to “50” (PicasaAQL-50) another set quality parameter to “95” (Picasa-95), the other set quality parameter to “100” (Picasa-100) for worst case in this experimental. We set the resize rate of all the PicasaAQL to “50” to get the images that is 80 x 80 pixels because original image size is 160 x 160 pixels, our application just show 60 x 60 pixels so 80 x 80 pixels is more better user interface looking. First, PicasaAQL get the necessary XML data by AQL query. For example, It will decrease XML size from

²On screen image is stand for the images that will initial show on Android phone’s screen. For example, if the screen can only contain 18 images, normal Picasa application will just download these 18 images until the user scroll the screen.

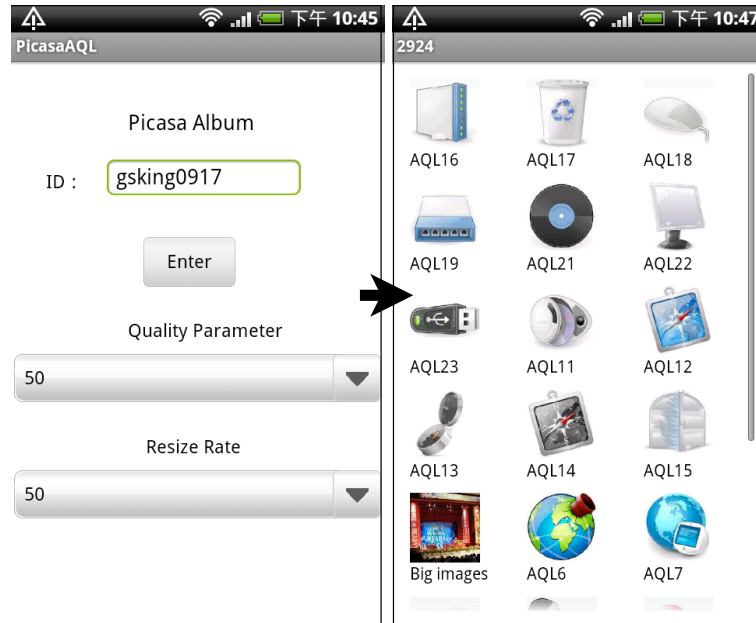


Figure 4.6: Application Screen Capture

19.443 bytes to 884 bytes (about 95.3% reduction) when there are 10 albums.

Table 4.8 shows the response time and the total transmission bytes (Request and Response). Figure 4.7 shows the line chart of Table 4.8. The response time is start from user click the “Enter” button to user see the album list. The total transmission bytes is the total packet length that send and receive by application during the response time. According to our experimental, we can found the normal Picasa application is little faster (200ms to 600ms) than PicasaAQL when the album image count is less (about 4 in Figure 4.7 (a)). The reason is the fewer images need fewer request headers and it directly get images from Picasa. But the total transmission bytes is larger than PicasaAQL. When the image count increase, normal application’s response time is raise very fast but PicasaAQL-50 and PicasaAQL-100 are increase slowly.

Table 4.8: Normal Picasa APP. versus PicasaAQL APP.

| | | Image Count | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
|-----------------|--------------------|-------------|--------|--------|--------|---------|---------|---------|---------|---------|---------|
| Normal Picasa | Response Time (ms) | | 1,441 | 2,025 | 3,040 | 3,462 | 3,821 | 4,959 | 5,576 | 5,938 | 7,345 |
| | Total Bytes | | 32,725 | 52,039 | 86,291 | 103,362 | 121,311 | 130,627 | 147,519 | 159,747 | 172,012 |
| PicasaAQL (50) | Response Time (ms) | | 2,278 | 2,322 | 2,627 | 2,725 | 2,635 | 2,896 | 2,938 | 2,969 | 3,013 |
| | Total Bytes | | 4,798 | 7,356 | 10,193 | 13,102 | 14,634 | 17,608 | 19,525 | 21,302 | 22,717 |
| PicasaAQL (95) | Response Time (ms) | | 2,131 | 2,477 | 2,583 | 3,011 | 3,156 | 3,190 | 3,348 | 3,669 | 3,722 |
| | Total Bytes | | 8,715 | 16,339 | 25,240 | 32,634 | 38,568 | 46,300 | 52,766 | 57,883 | 62,485 |
| PicasaAQL (100) | Response Time (ms) | | 2,260 | 2,736 | 2,824 | 3,266 | 3,375 | 3,442 | 3,551 | 3,827 | 3,915 |
| | Total Bytes | | 12,653 | 26,856 | 40,842 | 53,409 | 63,746 | 77,181 | 87,786 | 96,120 | 104,091 |

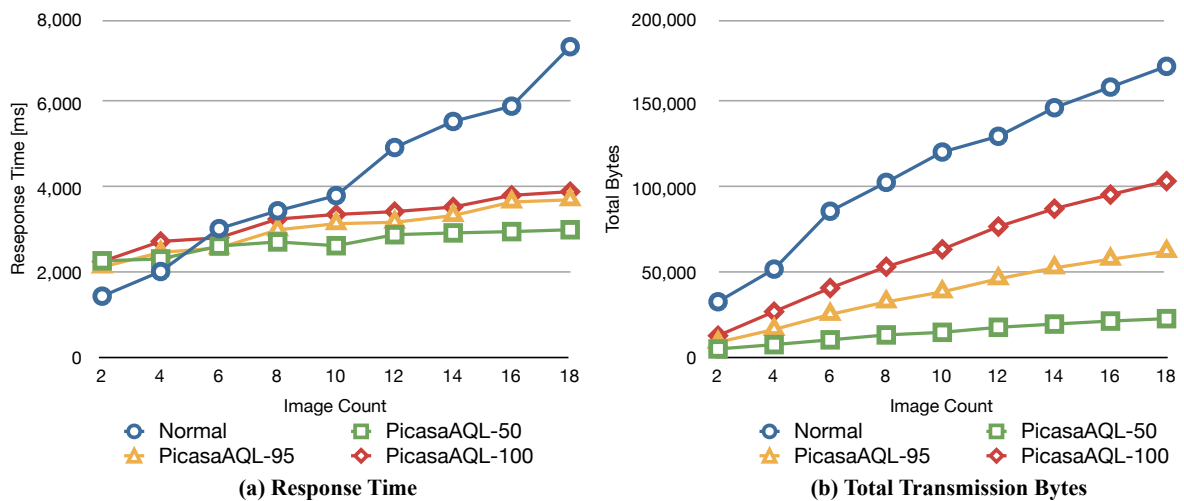


Figure 4.7: Line Chart of Response Time and Total Transmission Traffic

Chapter 5

Conclusion

In this thesis, we observed the overhead when Android mobile phone application calling the RESTful API in wireless environment. The Android mobile phone has very fantastic user interface and a large number of application to use. More and more mobile phone users using the Android REST client application to take the place of using Web browser to access the internet. The wireless network is more weaker than wired network so we proposed a system architecture to reduce the wireless transmission overhead when calling RESTful APIs. Our system architecture has two parts. Client-Side Library in Android application is to send the API query language to Proxy-Side Library and handle the response result from our proxy. It can reduce the HTTP request headers and use fewer request to get more results by AQL. Proxy-Side Library is to parse the query from Client-Side Library. It can filter the result according to AQL and compress the XML, JSON or any plain text format data to reduce the transmission traffic. Proxy-Side Library also have “Image Multiple Get” module to provides image compression, image resize and image combined function to reduce the Image transmission overhead. Experimental results show our system architecture can reduce the transmission traffic and improve the response time. For the common plain text data, it will reduce over 61% of original data. For the images, according to image count and parameter setting, it could reduce the total transmission bytes about 80% and speed up the response time to about 50% when there were over 10 small images and the quality parameter and resize rate was set to 50.

Bibliography

- [1] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.
- [2] I. Kilanioti, G. Sotiropoulou, and S. Hadjiefthymiades, “A client/intercept based system for optimized wireless access to web services,” in *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pp. 101–105, 26-26 2005.
- [3] T.-Y. Chang, Z. Zhuang, A. Velayutham, and R. Sivakumar, “Webaccel: Accelerating web access for low-bandwidth hosts,” *Computer Networks*, vol. 52, no. 11, pp. 2129–2147, 2008.
- [4] *Open Mobile Alliance Inc., WAP FORUM*. <http://www.wapforum.org>.
- [5] B. C. Housel, G. Samaras, and D. B. Lindquist, “Webexpress: A client/intercept based system for optimizing web browsing in a wireless environment,” *Mobile Networks and Applications*, 1998.
- [6] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas, “Dynamic adaptation in an image transcoding proxy for mobile web browsing,” *IEEE Personal Communications*, pp. 8–17, 1998.
- [7] Y. Hwang, J. Kim, and E. Seo, “Structure-aware web transcoding for mobile devices,” *IEEE Internet Computing*, vol. 7, no. 5, pp. 14–21, 2003.
- [8] T. Bickmore and B. N. Schilit, “Digester: Device-independent access to the world wide web,” in *Proc. WWW-6*, pp. 655–663, 1997.
- [9] O. Buyukkokten, H. Garcia-molina, and A. Paepcke, “Seeing the whole in parts: Text summarization for web browsing on handheld devices,” pp. 652–662, 2000.
- [10] J. Chen, B. Zhou, and H. Zhang, “Function-based object model towards website adaptation,” in *In Proceedings of the 10th International World Wide Web Conference*, pp. 587–596, ACM Press, 2001.

- [11] Y. Chen, “Detecting web page structure for adaptive viewing on small form factor devices,” in *In Intl. World Wide Web Conf. (WWW)*, pp. 225–233, ACM Press, 2003.
- [12] X.-D. Gu, J. Chen, W. ying Ma, and G. liang Chen, “Visual based content understanding towards web adaptation,” in *In Second International Conference on Adaptive Hypermedia and Adaptive Web-based Systems (AH2002)*, pp. 164–173, 2002.
- [13] Z. Hua, X. Xie, H. Liu, H. Lu, and W.-Y. Ma, “Design and performance studies of an adaptive scheme for serving dynamic web content in a mobile computing environment,” *IEEE Trans. Mob. Comput.*, vol. 5, no. 12, pp. 1650–1662, 2006.
- [14] *Yahoo! Query Language*. <http://developer.yahoo.com/yql/>.
- [15] *Facebook Query Language*. <http://developers.facebook.com/docs/reference/fql/>.
- [16] *Yahoo!奇摩生活 + API*. <http://tw.developer.yahoo.com/lifestyle/>.
- [17] P. Deutsch, “Gzip file format specification version 4.3,” *RFC 1952*, May 1996.
- [18] *Twitter API Documentation*. <http://dev.twitter.com/doc>.
- [19] *Yahoo!奇摩知識 + API*. <http://tw.developer.yahoo.com/knowledge/>.
- [20] *Best Practices for Speeding Up Your Web Site by Yahoo! Developer Network*. <http://developer.yahoo.com/performance/rules.html>.