Characterizing RESTFul Web Services Usage on Smartphones: A Tale of Native Apps and Web Apps

Yi Liu[1], ‡Xuanzhe Liu[1], Yun Ma[1], Yunxin Liu[2], Zibin Zheng[3], Gang Huang[1], and M. Brian Blake[4]

[1]*Key Lab of High-Confidence Software Technology (Peking University), Ministry of Education, Beijing China,* [2]*Microsoft Research, Beijing, China*
[3]*School of Advanced Computing, Sun Yat-Sen University, Guangzhou, China*[4]*University of Miami, Coral Gables, USA*
[1]{*liuyi14, liuxuanzhe, mayun*}*@pku.edu.cn,*[2]*yunliu@microsoft.com,* [3]*zibin.gil@gmail.com, hg@pku.edu.cn,* [4]*m.brian.blake@miami.edu*

***Abstract*—The burst of Web-based RESTful services brings us a number of facilities in our life and work. We are used to take smartphones to access these Web services, like location-based services, weather search, mapping, social networking, et al. On smartphones, we have two options of service consumers, a.k.a, native apps and Web apps. Despite the platform-independence, Web apps are claimed to provide the same features and comparable user experiences with native apps. However, one fact is that more and more people prefer native apps rather than Web apps. In this paper, we make an empirical study on characterizing the performance disparity of native apps and Web apps. Given the same functionalities provided by the same service providers, we explore the RESTful Web services that are used by native apps and Web apps. With HTTP-level trace analysis, we demystify the workflows on how native apps and Web apps use Web services and summarize different service usage patterns from architectural style perspective. Then we characterize the performance differences between native apps and Web apps on realizing RESTful Web services including GET, DELETE, PUT & POST, in terms of number of network connections, response time, and data drain, given the same functional features. Our observations reveal that Web apps do not always perform worse than native apps using RESTful Web services under the same context. We further propose some implications to improve both native apps and Web apps on smartphones.**

***Index Terms*—[1]Mobile; native apps; Web apps; RESTful Web services**

I. Introduction

Services computing paradigm provides a foundation to make software functionalities published and accessed in standard way. Numerous Web service providers, like *Google, Facebook*, and *Amazon*, have provided a large number of Web services to facilitate our life and work. Compared to the early age of services computing when Web services were mainly delivered by SOAP and WSDL, most of currently popular Web services are more "purely Web oriented, i.e., published and accessed by means of RESTful Web services APIs.

In the past decade, mobile devices like smartphones and tablet computers become popular. People are used to access various RESTful Web services via their devices, e.g., weather query by *Yahoo!*, web search by *Google*, shopping by *Amazon*, and social networking by *Facebook*. From services computing perspective, mobile devices, or more specifically, mobile apps, actually play the role as Web services consumers. Numerous mobile apps including native apps and Web apps, are consuming a large number of Web services.

One fact is that, a lot of popular service providers often maintain two forms of clients as their service consumers on mobile devices, i.e., native apps and Web apps. Conceptually, both of the two app forms fulfill specific tasks and features for end-users. For example, we can view the timeline through either the native app or the Web app provided by *Facebook*. According to our investigation, by the end of June, 2014, 132 of the top 200 websites in Alexa have maintained both a native app and a Web app.

Although these two app forms can provide similar or even the same functionalities, they are quite different for software developers. Native apps are claimed to better leverage device hardware, but the development of native apps requires efforts to deal with platform-specific features by using different programming languages. For example, iOS needs Objective-C and Android needs Java. On the other hand, Web apps developed by standard HTML, JavaScript, and CSS can promise cross-platform development and deployment. Users can access the corresponding Web apps in popular web browsers like Chrome, Safari, and Firefox on any platform. However, the distinction between a Web app and the corresponding native app is claimed to be blurry with the advancement of HTML5. Nevertheless, a recent survey by Flurry [1], found that mobile users in the US spend 86% of their time using native or hybrid apps, while only 14% of the customers time is spent in the browser using mobile websites (mobile apps).

There have been several debates on the comparison of native apps and Web apps. Compared to existing studies which mainly focus on programming language supports, we are motivated by the preceding fact that native apps are dominant consumers of Web services on mobile devices. We want to explore how native apps and Web apps consume Web services, and to what extent their performances vary when realizing the same functionalities. In this paper, we conduct a trace-based empirical study to analyze features and performance between the native app and the Web app implemented by popular service providers, including *Facebook, Sina Weibo, Google*

[1]‡: corresponding author: liuxuanzhe@pku.edu.cn

Map, Baidu Map, Amazon, and *Dianping*[2]. Our study was built on 47 features and 52 Web services (APIs), covering typical RESTful operations including `GET`, `DELETE`, `PUT`, and `POST`. In our experiment, we use two smartphones of the same model to run the native app and the Web app concurrently in the same environment, and explore how they perform with the same operation. For example, we input the same content and click the post button simultaneously when we test the feature of posting a timeline in *Facebook*. We use Fiddler [3] as a proxy between the Web service provider and mobile devices to capture the HTTP and HTTPS traffic and logs. Then we make trace analysis by demystifying the different workflows on how native apps and Web apps consume Web services. Meanwhile, we investigate the number of network connections, response time and traffic volume per Web service invocation, and summarize the different performance.

More specifically, this paper tries to answer the following three research questions:

- **RQ1: How do native apps and Web apps use Web services on smartphones respectively?** We summarize the Web service consumption behavior of native apps and Web apps from architectural style perspective.
- **RQ2: How do native apps and Web apps differ from performance when providing the same functionality?** We show how performance varies between native apps and Web apps in terms of network connections, response time and traffic volume.
- **RQ3: How to optimize Web apps and native apps for comparable performance?** We analyze the bottleneck and propose some implications to improve both native apps and Web apps on smartphones in Web services usage.

The remainder of this paper is organized as follows. Section II introduces the measurement methodology and how we conduct our experiments. Section III describes the architecture of native app and Web app in the respective of Web service. Section IV presents the results and analysis of our experiments. Section V provides our findings and implications. Section VI presents related work, and Section VII concludes this paper.

II. MEASUREMENT METHODOLOGY

Due to the tremendous growth of mobile devices over past a few years, mobile apps have become the big consumers of Web services. As is known to all, Web services can follow either a Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) pattern [11], However, although mobile computing has improved so much in recent years, consuming the SOAP-based Web services is still too heavyweight and may result in unacceptable performance overheads. In contrast, the RESTful Web service offers a lightweight and flexible architecture, and is much more suitable for mobile applications [11]. From our study, almost all popular Internet-scale service providers, like *Google, Amazon, Facebook*, use RESTful fashion to publish their Web services. Therefore, all apps we use in our experiments are consumers of RESTful Web services.

[2]DianPing (http://www.dianping.com/) is a provider of life style service in China to search resturants just like Yelp.

A. Data Set

In our experiment, we make trace-based analysis to anatomize the workflows of native apps and Web apps as service consumers by performing the same functionality. To collect the experimental data for our empirical study, we study six popular Web service providers with their native apps and Web apps. The service providers are *Facebook, Amazon Shopping, Google Map, Baidu Map, Sina Weibo*, and *Dianping*, as shown in Table I. We check 47 features with 52 RESTful Web services. We also record the network traces including the number of network connections, response time, and traffic data drain of the same feature on the two forms of apps. For example, as shown in Figure 1, the left is the native app of *Amazon Shopping*, and the right one is the corresponding Web app. We list three respective traces of both the native app and the Web app, responding to three features provided by *Amazon shopping*. For example, when we add a goods to cart, the native app initiates a request to "*/aw/detail/ajax/add-to-cart/*" service, while the Web app initiates a request to "*/aw/c/*" service. **An interesting finding is that, the target URLs of RESTful Web services on native apps and Web apps are not the same, given by the same functionality.** It implies that Web service providers have different API semantics for native apps and Web apps, respectively, though both of them are in form of RESTful. Subsequently, developers need to implement and maintain different Web service invocation client codes to initiate service requests.

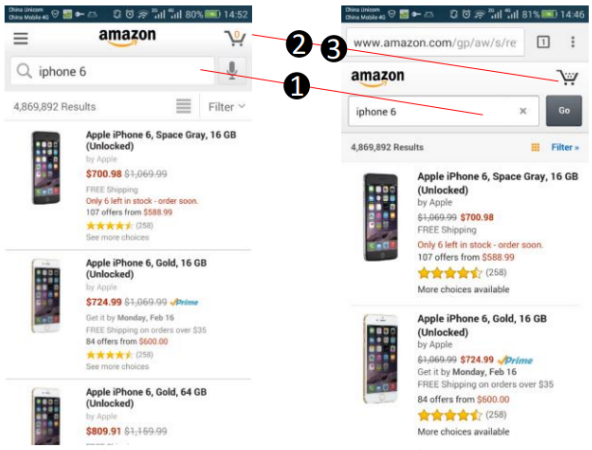
TABLE I: Selected Apps of Targeted Web services

APP Name	Description	# of Features	# of Web services
Facebook	Social Networking	10	12
Sina Weibo	Social Networking	10	12
Amazon	Onlining Shopping	9	9
Google Map	Map	4	4
Baidu Map	Map	4	4
Dianping	Lifestyle like Yelp	10	11

B. Definition

We set out to measure the metrics of the two forms of apps developed by a specific Web service provider. For each feature, we would collect the corresponding network traces from Fiddler [3]. We define some terms used for describing our approach as following:

1) **Service.** A service is an atomic functionality provided by the vendors. For example, adding goods to cart and deleting goods from cart in *Amazon Shopping* should be viewed as services. Technically, a service has RESTful operations, e.g., `GET`, `DELETE`, `PUT`, or `POST`.



Native Apps

Feature	Trace	Content-type
①	/s?k=ipnone6&rh=i%3Aaps%2Ck%3Aiphone6…	sapplication/json
	/images/I/51wcd%2Bz24TL._SL110_AC.jpg	image/jpeg
	/images/I/51zcjkPXWYL._SL110_AC.jpg	image/jpeg
②	/gp/aw/detail/ajax/add-to-cart/ref=mw_pa_dp_buy_cart	text/json;charset=UTF-8
③	/gp/cart/ajax-update.html/ref=ox_sc_cart_delete_1	application/x-javascript

Web Apps

Feature	Trace	Content-type
①	/gp/aw/s/ref=nb_sb_noss/478-4950856-6482801?k=iphone6	text/html;charset=UTF-8
	/images/G/01/AUIClients/AmazonUI-afd76f38adbc3d6c…	application/x-javascript
	/images/G/28/nav2/images/sticker/sx-popular-elipse._CB…	image/png
②	/gp/aw/c/ref=mw_dp_buy_crt	text/html;charset=UTF-8
③	/gp/cart/ajax-update.html/ref=ox_sc_cart_delete_1	application/x-javascript

Feature①:Search keywords; Feature ②: Add goods to cart; Feature③: Delete goods from cart

Figure 1: Trace Logs of the Native App and the Web App of Amazon Shopping generated by the Same Features

2) **Feature.** A feature is a specific set of Web services in a specific order. It corresponds to a specific functionality offered by a Web service provider, such as posting a blog, adding a comment, and login. A feature may contain more than one services.
3) **Trace.** A trace is a sequence of HTTP/HTTPS requests and responses. Figure 1 shows three traces of *Amazon Shopping*, respective to features of searching keywords, adding goods to cart, and deleting goods from cart.
4) **Number of network connections.** The number of network connections is actually the number of requests and respective responses in a trace. For the feature of adding goods to cart in *Amazon Shopping*, there is only one request "*/aw/detail/ajax/add-to-cart/*" initiated from native app as show in Figure 1, so the number of network connections is one.
5) **Traffic volume.** The traffic volume means the total network traffic of a trace, including the bytes sent and bytes received.
6) **Response time.** The response time is from the time of the first network request initiated to the time of the last response finished in a trace. As shown in Figure 1, the response time of searching keywords feature means the time between the start of first request "*s/k=iphone...*" and the end of last response "*/images/...*".

C. Experiment Setup

In this part, we introduce the infrastructural platforms used for experiments and how we conduct our experiments.

Hardware. We use two smartphones of the same model for our investigation. The mobile devices are Samsung Galaxy S4 with 2 GB RAM and Android 4.2 OS. We deploy native apps and Web apps on them, respectively, and test the service invocation under the same experimental condition. In this way, we could operate concurrently for testing both apps. The Android devices are rooted so that we could prevent other apps from connecting network so as to reduce noise. At the same time, both smartphones visit the network via the same Wi-Fi to maintain the same network condition.

Software. Our experiment is conducted by using two forms of apps for each targeted Web service provider. We install the native app in one handset, and open the Web app through Android Chrome 40.0.2214.89 in the other one.

We set out to fulfill each feature of targeted Web service in one of two modes:

1) **No Cache.** First, we operate concurrently in the both forms of apps for a specific feature without cache. The native app is manually re-installed before we restart our experiment, and browser cache is manually cleared before loading the Web app again. Each time we record the number of network connections, traffic volume, loading time of providing the same feature.
2) **With Cache.** Second, we test the same feature again for the both apps. This enables us to characterize the performance of the two apps with cache.

D. Experiment Rationale

To record the traces and traffic information of requests and responses between the mobile device and Web services, we use Fiddler [3] as a proxy between them. Fiddler can capture HTTP and HTTPS traffic and logs for us to review the network details. Then, we compare the traces of the two respective versions of each application, in terms of the same feature. For example, we analyze the traces of searching a person feature offered by *Facebook* in both the native app and the Web app, with the same keyword query.

III. Architecture Anatomy

A. Trace-based Service Analysis

We then describe how native apps and Web apps consume the Web services by providing the same feature, respectively.

For better illustration, we take a popular Web service, *Amazon* as our example to illustrate the experiment. Figure 1

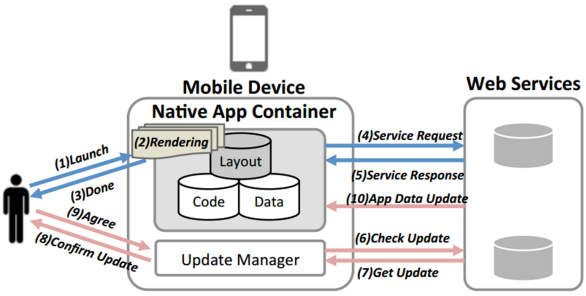


Figure 2: The Architecture of Native App

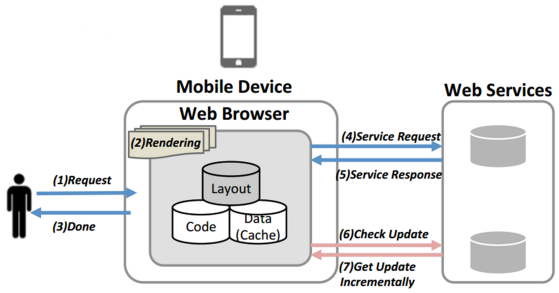


Figure 3: The Architecture of Mobile Web App

shows some traces of *Amazon* on both the native app (left) and the Web app (right). When users fill in basic information and click some buttons, it will lead to some consequent requests and respective responses. We list three features, including `search_keywords`, `add_goods_to_cart`, and `delete_goods_from_cart`. The left is the native app, and the right is the Web app. We list the traces aside. We notice that the contents of services responses are always formatted with JSON in the native app, then parsed and exhibited to the user by the processing logic of native code. We can see that these two forms of apps have similar user interface, but request different Web services URLs for the same operation. According to the operation and traces, we could manually characterize the feature and choose the same feature of the native app and the web app for subsequent analysis.

We can find that all the requests are HTTP/HTTPS requests, and the provided Web services are RESTful. Especially, *Facebook* provides the pure RESTful API for the developers to get the social graph [2]. It is a low-level HTTP-based API that we can use to query data, post new stories, upload photos, and a variety of other tasks. We can conduct reading, publishing, updating and deleting operation of the *Facebook's* social graph, well corresponding to the `CRUD`[3] operations of HTTP. To support clients that do not support pure RESTful methods, developers can alternatively issue a `POST` request to override the HTTP method. We also find that the *Facebook's* official native app in Android platform realizes almost all the requests in `POST` method. For example, according to the API guide, developers could use HTTP/HTTPS `DELETE` request to delete a comment. However, its official native client in Android platform just initials a `POST` request with additional parameters in HTTP header. The `add_good_to_cart` refers to a `POST` operation, the `delete_goods_from_cart` refers to a `DELETE` operation, and the `search_keywords` refers to a `GET` operation.

B. Architecture of Native App of using Web Services

When requesting Web services, native apps come up with lots of elements pre-loaded and only need to fetch user data from the web server rather than the entire application. However, users have to keep downloading updates manually. The native apps always have their own update manager, which is responsive for checking updating and downloading the latest version to install. When user launches the *Weibo's* native app, the update manager would send a `GET` request to "*client/version*" service, checking for updating. Once the newer version is released, the update manager would alert user to update since the latest one might be out of support.

Figure 2 shows the architecture of a native app on using Web services. Once a native app is installed onto the device, the native code and some resources have been stored, and therefore the processing logics consisting of data parsing, transaction processing and layout are maintained locally on the mobile client. When users launch the native app or click some buttons, the processing logic of native code starts. It may trigger the resources requests to the targeted Web services, and issue asynchronous requests to fetch the user data. The native app would create a new thread to issue an asynchronous request when resource is needed, therefore leading to higher concurrency. When the data is downloaded, the native code would parse the data (for example, encoded in JSON format), process as the pre-setup logic by the native code and re-render the new user interface. It needs to be stressed that some resources have been stored in the mobile client after the app is installed, and developers always use cache mechanism to reuse the downloaded resources, especially the images.

We also find that, some native apps are actually so-called "hybrid" ones, i.e., embedding Web pages in native apps. The hybrid can take advantage of many device features available, but some pages rely on HTML rendered in the embedded Android WebView component. Typically, the native app of *Amazon Shopping* is a hybrid one.

C. Architecture of Web App of using Web serivces

In contrast, Web apps are accessed in the mobile devices web browser, and update themselves without the need for user intervention. Web apps are built in standards-based technologies such as HTML5, Cascading Style Sheets (CSS), JavaScript, and other modern web tech. It has to download all resources such as page files, images, JavaScript files, and CSS files to render the user interface in the mobile browser when users visit the Web app. Each time users visit a Web app, they are displayed by the latest version of the app. Figure 3

[3]CRUD means create, read, update and delete

shows the architecture of a Web app. Due to the lack of local code and resources, the Web app needs to download all the resources from the Web services, including HTML files for page layout, CSS for page formatting, JavaScript files for interacting with the users and other web contents. When users first load the Web app, the browser initiates the first request to fetch the main page of the Web service, if the Web service has user interface. After parsing the page in form of HTML, consequent resource requests for JavaScript, CSS files and other web contents are initiated. When users interact with the Web app, there are two forms of processing. The first one is to input some forms, then the Web app would jump to another page. The other one is that the Web app initiates an AJAX request, and deals with the callback methods without page jumping.

IV. PERFORMANCE EVALUATION

Following the experiment setup described in the previous section, this section focuses on the performance evaluation of native apps and Web apps.

Developers can call RESTful Web services via HTTP or HTTPS. For each app, we select some features, and collect the respective network trace logs for both forms to differentiate the native app and Web app. We categorize all the tested features into three groups corresponding to the RESTful operations, i.e. `GET`, `DELETE`, `PUT`, and `POST`. Here we aggregate the `PUT` and `POST` features into one group since they both result in resources updating.

We find the *Facebook's* features because of its well-designed API following the principles of RESTful Web services and plentiful documentation [2]. However, other Web services providers do not always design their APIs strictly according to the standard of RESTful fashion, so we manually group these features according to the definition of CRUD and the respective RESTful Web services from the trace logs. For example, we group the feature of adding goods to shopping cart in *Amazon Shopping* into the `POST` category.

In the following sections, we are going to analyze the performance variations of native apps and Web apps when performing the same feature. In particular, we investigate the network connections, response time, and traffic volume by calling the corresponding RESTful Web services. We treat the situations ***with*** and ***without*** cache respectively.

Due to space limit, this paper only shows the performance evaluation of one typical Web service per category, i.e., 20 features in total. The full experimental evaluation could be found at our lab server via http://www.sei.pku.edu.cn/~liuxzh/ICWS2015/RESTonMobile).

A. PUT&POST

In RESTful Web services semantics, the `POST` refers to the request that send a representation of a new data entity to the server so that it will be stored as a new subordinate of the resource, and `PUT` refers to the request that modifies the existing resource. We examine seven typical features that need `PUT/POST` operations, `post_timeline` feature and `set_like` feature in *Facebook*, `set_like` feature and `add_comment` feature in *Weibo*, `collect_restaurants` in *DianPing*, `add_goods_to_cart` and `collect_goods` in *Amazon Shopping*. Figure 4, Figure 5, and Figure 6 show the number of network connections, the traffic volume, and the response time in `PUT&POST` operations, respectively.

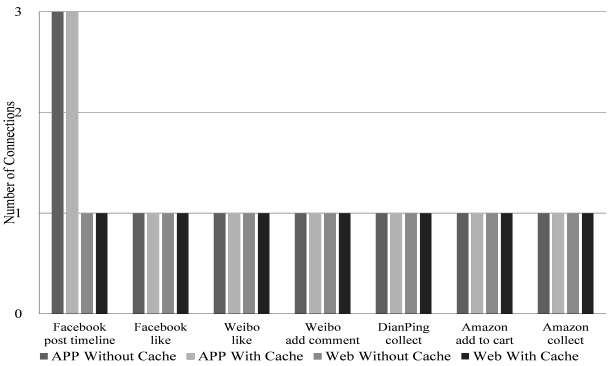


Figure 4: Number of connections in PUT&POST

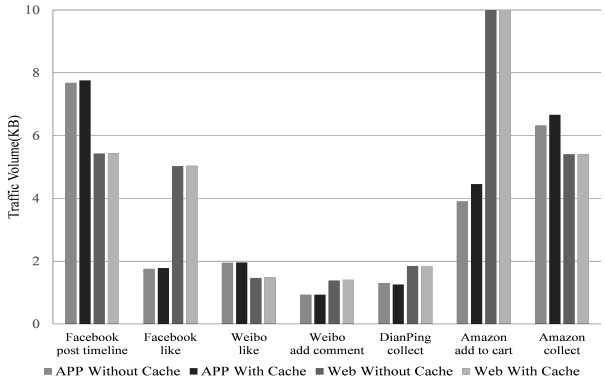


Figure 5: Traffic volume in PUT&POST

Network Connections. We can find that the network connections for the same feature request do not differ quite much. Both the native app and Web app initiate the same number of requests with or without cache. In term of the `post_timeline` feature of *Facebook*, the native app initiates more requests. The reason is that it programmatically refreshes the whole timeline after the `POST` operation is finished, and triggers some additional requests.

Traffic. The traffic volume may differ significantly between the two apps. Generally, Web apps cost more traffic volume than native apps for the same `PUT&POST` operation. It is not very surprising since Web apps always send extra information, such as cookie and longer User-Agent. But, we find there are exceptions. For the *Facebook's* `post_timeline` feature, it refreshes the whole timeline afterwards. For the *Weibo's* like feature, the response data contains extra user information so that the native app consumes a little more network traffic than the Web app. For the *Amazon Shopping's* `collection` feature, the native app sends and receives a little more traffic

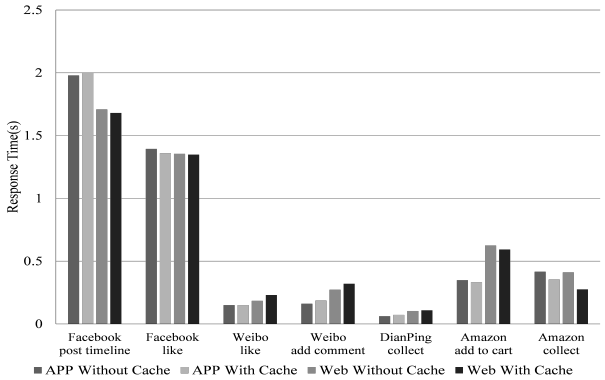


Figure 6: Response time in PUT&POST

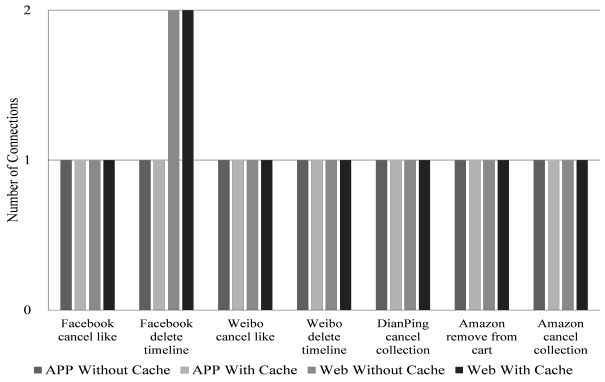


Figure 7: Number of connections in DELETE

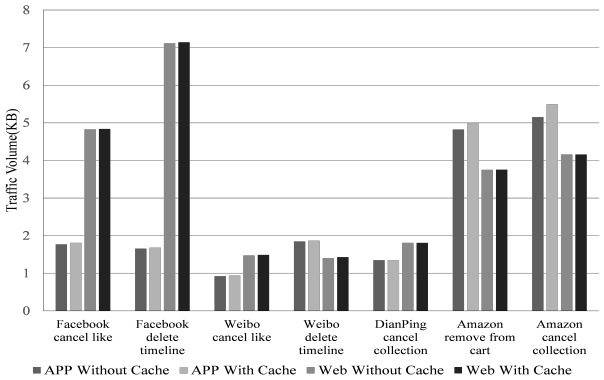


Figure 8: Traffic Volume in DELETE

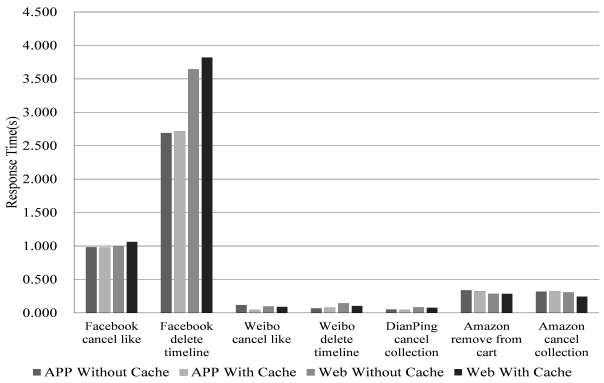


Figure 9: Response time in DELETE

volume. We can also find that the cache mechanism marginally works in the `PUT&POST` operations, i.e., the traffic volume changes less than 2% for Web app and less than 4% for native app. Instead, for the two tested features of *Amazon*, the native app sends extra information such as cookie when we conduct our experiment with cache, then the native app consumes a little more network traffic.

Time. The same observation could be found in response time metric and the traffic volume metric. In most situations, more traffic volume means more response time. It complies with our expectation.

B. DELETE

In RESTful Web services semantics, `DELETE` refers to the request that deletes an existing resource. We examines seven typical features that need `DELETE` operations, `delete_timeline` and `cancel_like` in Facebook, `cancel_like` and `delete_timeline` in *Weibo*, `cancel_collection` in *DianPing*, `delete_goods_from_cart` and `cancel_collection` in *Amazon Shopping*. Figure 7, Figure 8, and Figure 9 show the number of network connections, the traffic volume, and the response time in `DELETE` operations, respectively.

Network Connections. For both the native apps and Web apps, they initiate the same number of requests to convey the deletion command. In term of the `delete_timeline` feature of *Facebook*, the Web app initiates more requests. This is because it refreshes the timeline after prior operation.

Traffic. Web apps cost more traffic volume than Web apps for the same `DELETE` operation except some apps. Similar to `PUT` and `POST`, native apps of *Weibo* and *Amazon* consume more traffic volume than the Web app because the native apps send more data and receive extra information. For example, when we consider `delete_timeline` feature of *Weibo*, the service returns the whole user information for native app, while we only need the hint of successful deletion. For other apps, the Web apps consume more network traffic since they need to download all necessary resources. The cache mechanism do not work well, either. The traffic volume of Web apps fluctuates within a narrow range of 2%, and native apps differ as much as 2.3% except the native app of *Amazon Shopping*. We need to mention that the native app of *Amazon Shopping* is a hybrid app, and it may act a little awkwardly.

Time. The same observation could be found in response time metric and the traffic volume metric. In most situations, more traffic volume means more response time, but the difference between native app and Web app provided by the same service provider is not obvious. Usually, the `DELETE` operation can be finished within one second. The `delete` timeline feature in *Facebook* is an exception.

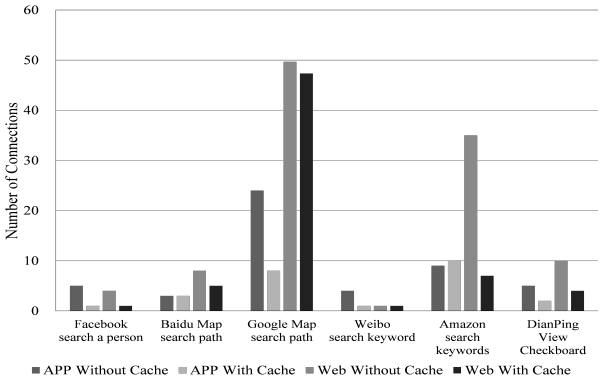


Figure 10: Number of connections in GET

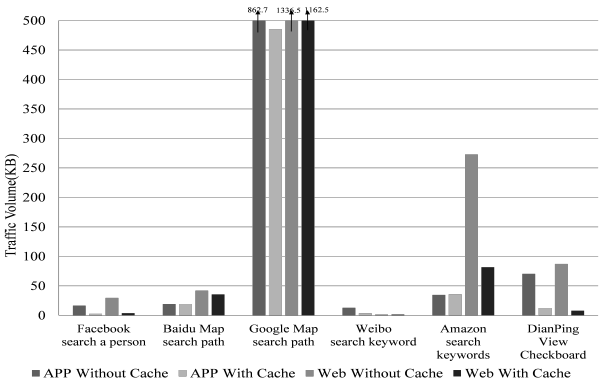


Figure 11: Traffic volume in GET

C. GET

In RESTful Web service semantics, `GET` refers to the request that retrieve a representation of a resource from Web server. We examine seven typical features that need `GET` operations, `search_a_person` in *Facebook*, `search_keywords` in *Weibo*, `view_checkboard` in *DianPing*, `search_keywords` in *Amazon Shopping*, `search_path` in *Baidu Map*, and *Google Map*. Figure 10, Figure 11, and Figure 12 show the number of network connections, the traffic volume, and the response time in `GET` operations, respectively.

Network Connections. In the `GET` operations, the numbers of network connections differ much more than native apps for the same feature. This is because the Web app needs to download all the needed resources including the CSS and images, which have nothing to do with the functionality but for layout. However, when we consider the `search_keywords` in *Weibo* and the `search_persons` in *Facebook*, the native apps initiate more requests. We examine the traces, and find that the native apps receive the user's friends with their head portraits according to the key words, and the user's photo with different sizes in *Facebook*. Apps with cache could save as much as 80 percents of request, and the Web app's performance may be comparable with the native app in terms of network connections.

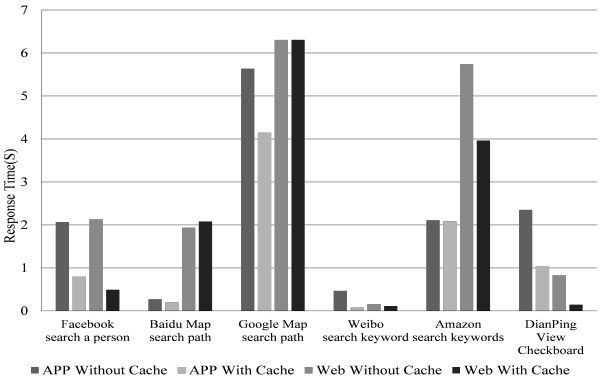


Figure 12: Response time in GET

Traffic. For traffic, the performance of Web app is usually worse than the native app in the most situations without cache. However, the native app of *Weibo* consumes much more network traffic because it requests more resources. We need to mention that the native app of *Facebook* also initiates more requests in the `search_persons` feature, but it consumes less network traffic volume. The cache mechanism in `GET` features works much better. Native apps on *Google-search* with cache could save as much as 90.9% traffic volume. *Baidu Map* disables cache for most resources, so traffic volume only varied a little. When we consider the network traffic in the with-cache situation, we are surprised to find that the performance of some Web apps may be better than the native ones. For the `search_persons` feature in *Facebook* and `view_checkboard` feature in *DianPing*, the Web apps consume less network traffic than the native ones with cache. We could find that the native ones may return more information to provide better user experience. It implies that the developers need to keep the balance of performance and user experience. It is a good implication that we could improve the performance of Web app by taking full advantage of cache mechanism.

Time. The response time of `GET` operations are affected by many factors. Mobile web browsers always allow a limited number of simultaneous connections to a single host, but the native app can fully use the ability of concurrency. Thus, a certain sequence existed in the `GET` operations, especially in the Web app. We can find that the tendency of response time metric is similar with the traffic volume metric. The performance of "*with-cache*" test is much better than no-cache test, and the performance of Web app with cache could be close to that of native app.

V. FINDINGS AND IMPLICATIONS

Based on above analysis, we are moving to **RQ3**. We then make analysis to understand the bottleneck and propose some implications to improve web apps on smartphones with comparable performance with native apps.

In most situations of RESTful Web service operations, the Web apps initiate more requests and consume more traffic and need longer response time. The reason is that Web apps usually

TABLE II: Summary of Findings and Implications

Findings	Implications
Given by the same feature, Web service providers have different RESTful Web service semantics for native apps and Web apps. As a result, the number of HTTP/HTTPS requests sometimes varies between the two app forms, e.g., the `post_timeline` in *Facebook*.	Though apps developers could implement the client logics manually or by some third-party HTTP libraries, Web service providers had better uniformly organize the RESTful APIs to reduce the HTTP connections.
Overall, given the same features, the Web apps perform worse than native apps, by consuming more network traffic volume and requiring longer response time.	Mobile users need to make a choice when they want to consume a specific Web service. In term of performance, users are recommended to use the native one.
For GET operation, sometimes the performance of Web apps may approach or even exceed the performance of the corresponding native apps when considering cache, e.g., `search_a_person` feature in *Facebook* and `view_checkboard` feature in *DianPing*.	Both Web service providers and Web apps developers should focus on the cache mechanism, since better cache requires the cooperation of client-side code, browser runtime, and server-side configuration, e.g., longer cache expiration time. Also, recently HTML5 localstorage and appcache are supposed to be promising solutions.
For some `POST`, `PUT`, and `DELETE` operations, native apps may perform worse than Web apps, e.g., `collect_goods` in *Amazon* and `delete_timeline` in *Weibo*. The reason is that Web services providers enforce their APIs to return extra data for refreshing the UI. However, such returned data is not always useful and necessary.	Web service providers should give developers the option to programmatically filter the returned data, and developers should be careful to add parameters to filter response data.
The native app of *Amazon Shopping* is a hybrid app. It reuses the pages of the Web app and take full advantage of the native features, for example, more choice of local storage, then perform better for the same feature `search_keywords`.	Hybrid app is a good choice for developers to develop a native app. It can reduce work and gain better performance than Web app.

need to download the extra CSS files, JavaScript files, and other web contents to build up user interfaces, while native apps can store them locally. However, from our observations, we can also find some implications to potentially improve the Web apps performance, and optimize native apps. We list the findings and implications in Table II.

VI. Related Work

Web services research on desktop computing has been well studied by a lot of literatures [13] [9], efforts on mobile computing contexts are just at the beginning. Due to the limited computation resources, RESTful Web services are considered to be a promising solution on mobile devices. Wagh et al. [12] made comparison study on the performance evaluation on handheld devices. Fokaefs et al. [5] focused on the performance of Web services on mobile platforms, and showed that the RESTful pattern was much more suitable for resources-constrained mobile devices. In practice, it is reported that most major Web service providers have adopted RESTful fashion [12] [7].

To improve Web services performance on mobile devices, some efforts are also made. Jamal et al. [8] used a cloud-based proxy and cache mechanism to enhance the performance of RESTful Web services for mobile devices. Our previous work also targets at how to make Web services invocation more energy-efficient [4].

Though there have been a long debate on native app and Web app on mobile devices [6], very little work has been done to understand the differences on Web services client evaluation. To the best of our knowledge, we make the first step in this direction. We used a trace-based analysis to uncover the workflows and performance variations among native apps and Web apps when requesting RESTful Web services for the same feature.

VII. Conclusion

This paper conducts an empirical study on two kinds of consumers of RESTful Web services: mobile native apps and mobile Web apps. Based on our analysis, the future work includes how to optimize Web apps to provide comparable performance with the native apps. More specifically, we plan to realize some mobile web browser cache configurations optimization based on our previous study[10].

Acknowledgement

This work is supported by the High-Tech Research and Development Program of China (Grant No. 2013AA01A605), the National Basic Research Program of China (Grant No. 2011CB302604), and the National Natural Science Foundation of China (Grant No. 61370020, 61222203, 61421091), Guangdong Natural Science Foundation (Project No. 2014A030313151).

References

[1] Apps solidify leadership six years into the mobile revolution. http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution#.VMdCU7H9PeM/.

[2] Facebook graph api rereference. https://developers.facebook.com/docs/graph-api/reference//.

[3] Fiddler, the free web debugging proxy for any browser, system or platform. http://www.telerik.com/fiddler/.

[4] P. Bartalos and M. B. Blake. Green web services: Modeling and estimating power consumption of web services. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 178–185. IEEE, 2012.

[5] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau. An empirical study on web service evolution. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 49–56. IEEE, 2011.

[6] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. *ACM SIGPLAN Notices*, 46(10):695–712, 2011.

[7] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2010.

[8] S. Jamal and R. Deters. Using a cloud-hosted proxy to support mobile consumers of restful services. *Procedia Computer Science*, 5:625–632, 2011.

[9] O. Kondratyeva, A. Cavalli, N. Kushik, and N. Yevtushenko. Evaluating quality of web services: A short survey. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 587–594. IEEE, 2013.

[10] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie. Measurement and analysis of mobile web cache performance. In *24th International World Wide Web Conference (WWW 2015)*, Accepted to appear.

[11] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[12] K. Wagh and R. Thool. A comparative study of soap vs rest web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications*, 2(5):12–16, 2012.

[13] S. Wang, W. A. Higashino, M. Hayes, and M. A. Capretz. Service evolution patterns. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 201–208. IEEE, 2014.