ReWAP: Reducing Redundant Transfers for Mobile Web Browsing via App-Specific Resource Packaging

Xuanzhe Liu, *Member, IEEE,* Yun Ma, *Student Member, IEEE,* Shuailiang Dong, Yunxin Liu, *Member, IEEE,* Tao Xie, *Senior Member, IEEE,* and Gang Huang, *Member, IEEE*

Abstract—Redundant transfer of resources is a critical issue for compromising the performance of mobile Web applications (a.k.a., apps) in terms of data traffic, load time, and even energy consumption. Evidence demonstrates that the current cache mechanisms are far from satisfactory. With lessons learned from how native apps manage their resources, in this article, we present the ReWAP approach to fundamentally reducing redundant transfers by restructuring the resource loading of mobile Web apps. ReWAP is based on an efficient resource-packaging mechanism where stable resources are encapsulated and maintained into a package, and such a package shall be loaded always from the local storage and updated by explicitly refreshing. By retrieving and analyzing the update of resources, ReWAP maintains resource packages that can accurately identify which resources can be loaded from the local storage for a considerably long period. ReWAP also provides a wrapper for mobile Web apps to enable loading and updating resource packages in the local storage as well as loading resources from resource packages. ReWAP can be easily and seamlessly deployed into existing mobile Web architectures with minimal modifications, and is transparent to end-users. We evaluate ReWAP based on continuous 15-day access traces of 50 mobile Web apps randomly chosen from Alexa top 500 ranking list. Compared to the original mobile Web apps with cache enabled, ReWAP can significantly reduce the data traffic, with the median saving up to 51%. In addition, ReWAP can incur only very minor runtime overhead of the client-side browsers and thus does not compromise user experiences.

Index Terms—Mobile Web Browsing, Redundant Transfer, Resource Package.

1 Introduction

Redundant transfer of web resources[1] refers to the case where a previously fetched resource is downloaded again from the network before the resource is actually updated. For mobile Web applications (a.k.a. apps) [1], redundant transfers remain as a critical performance issue leading to duplicated data transmission, long page load time, and high energy drain [2], [3].

Redundant transfers originate from apps' resource-management mechanism, which is to determine whether a resource should be loaded locally or remotely. Web cache is a conventional resource-management mechanism adopted by Web apps. Web developers can configure cache policies, such as expiration time and validation flag, on those resources that are likely to be loaded from the local storage. The browser maintains a cache space and deals with the cache logic for all the Web apps running in it. However, our previous work [4], [5] indeed found that there is a big gap between the ideal and actual cache performance of mobile Web apps. For example, for the mobile versions of top-100 websites of Alexa, although more than 70% of resources can be loaded from the cache when these websites are revisited after one day, less than 50% of these cacheable resources are actually loaded from the cache. Surprisingly, all resource transfers are redundant for some well-known websites when they are revisited after one day. We also revealed two major causes for redundant transfers: (1) the imperfect cache configuration, such as heuristic expiration and conservative expiration time; and (2) the undesirable Web development practice, such as using random strings to name resources for enforcing their refresh.

Due to the dynamics of mobile Web apps, it is difficult for Web developers to properly configure the apps' cache policies. Short expiration time may lead to redundant transfers, while long expiration time may result in the usage of stale resources. As a result, simply using cache policies is not a desirable mechanism to accurately determine whether resources should be loaded locally or remotely [5].

To fundamentally reduce redundant transfers for mobile Web apps, the resource management in native apps can provide some useful inspirations. Resources of native apps are managed directly and explicitly by app-specific logics to control where to load resources and when to update the local resources. Intuitively, native apps explicitly distinguish their static resources from dynamic ones, and encapsulate the static resources into a package that is installed into a dedicated space allocated by the underlying operating system. When a native app is running, the app logic controls that only its dynamic resources are downloaded on demand to provide the fresh data to users, while the static resources in the installed package are always fetched locally. When the static resources have to be updated, a new resource package is downloaded and installed to refresh all the static resources.

- *Xuanzhe Liu, Yun Ma, Shuailiang Dong, Gang Huang are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871. Email: {liuxuanzhe, mayun, sldong, hg}@pku.edu.cn*
- *Yunxin Liu is with Microsoft Research, Beijing, China, 100084. Email: yunxin.liu@microsoft.com*
- *Tao Xie is with the University of Illinois at Urbana-Champaign. Email: taoxie@illinois.edu*
- *Xuanzhe Liu and Yun Ma contribute equally to this article.*

Manuscript received 1 June. 2016; revised 28 Sept. 2016; accepted 15 Nov. 2016.

1. In this article, Web resources, in short as *resources*, refer to resource objects constituting an app (such as HTML, JavaScript, CSS, and images of a Web app; native code, media files, and layout files of a native app).

However, there are two main challenges for Web developers to adopt such a package-based resource management specific to a Web app. First, it is hard to maintain the resource package. Resources of Web apps are loosely coupled, and usually updated independently and casually without influencing each other. As a result, it is tedious and error-prone to decide which resources should be put into the package and when to update the package. Second, it is hard to enable mobile Web apps to use the package. Modern Web apps are complex and there are many mature Web development frameworks. As a result, a lot of manual efforts are needed to realize or refactor mobile Web apps to benefit from the package-based resource management.

To address these challenges, in this article, we present the ***ReWAP*** approach to restructuring mobile Web apps to be equipped with package-based resource management while requiring minimal developer efforts. The key rationale of ReWAP is to provide more efficient and app-specific control of resource management rather than relying on only the current mechanisms such as Web cache, to avoid the caused unnecessary redundant resource transfers. By retrieving the update of resources of mobile Web apps, ReWAP automatically maintains resource packages by accurately identifying which resources should be loaded from the local storage for a considerably long period. Based on the package information, the ReWAP-enabled Web app automatically checks the update of resource packages, refreshes the resources in the package when the resource package is updated, and loads resources from the resource package. To integrate ReWAP with existing mobile Web apps, Web developers need to conduct only minor modifications to their existing implementation. In summary, *ReWAP shares the same spirit of resource management mechanisms as those in the installation package of native apps but in the way of Web.*

To the best of our knowledge, our work is the first to facilitate Web developers to effectively reduce redundant transfers of mobile Web apps by conducting resource management in a similar way as native apps. More specifically, this article makes the following main contributions:

- We design ReWAP, a packaging approach for a mobile Web app to accurately identify resources that can be loaded from the local storage for a considerably long time. The maintained package can maximize the benefit of data-traffic saving by considering all the users of the mobile Web app.
- We implement ReWAP with the goal of minimizing developer efforts of restructuring existing mobile Web apps. Web developers can easily integrate ReWAP in their current implementation of a mobile Web app, and the end-users are completely unaware of the existence of ReWAP when they access the Web apps.
- We conduct experiments based on 15-day access logs of 50 mobile Web apps randomly chosen from Alexa top 500 rank list to evaluate the effectiveness of ReWAP. Compared to the original mobile Web apps with default browser cache enabled, ReWAP can significantly save the data traffic with the median up to 51% and the maximum of almost 100%. In addition, ReWAP incurs only quite small runtime overhead of the client-side browsers.

The remainder of this article is organized as follows. Section 2 illustrates the problem of redundant transfers with an example and compares the resource-management mechanisms of Web apps and native apps. Section 3 presents the overview of the ReWAP approach. Sections 4 and 5 show the details of ReWAP's key components, i.e., Package Engine and Wrapper, respectively. Section 6 describes the implementation of ReWAP and demonstrates its easy deployment. Section 7 presents the evaluations of ReWAP based on top Web apps of Alexa. Section 8 discusses limitations of ReWAP and possible solutions. Section 9 presents the related work and Section 10 concludes this article.

2 Background and Motivation

In this section, we present the background and motivation for leveraging the resource-management mechanisms of native apps to improve Web apps. We first describe a motivating example to illustrate the problem of redundant transfers. Then we compare the resource-management mechanisms of Web and native apps.

2.1 Redundant Transfer in Mobile Web Apps

Although the resource management of Web apps is flexible enough to achieve easy-to-access and always-updated features, it could lead to redundant transfers of resources. We illustrate redundant transfers via an example shown in Figure 1.

Figure 1(a) shows resource excerpts of a mobile Web app "http://m.foo.com/". The HTML resource indicates that the app includes a layout resource "a.css" and a JavaScript resource "b.js". When "a.css" is being evaluated, a background image "bg.png" is identified. When parsing the HTML resource is finished and the `onload` event is triggered, JavaScript function `f` is executed to get the address of an image by requesting a service "/image/address". Suppose that the returned address is "d.jpg?0.892", and then the image is retrieved. Therefore, when the app is visited at the first time, 6 resources are actually retrieved. Figure 1(b) shows the cache configuration of these resources. The HTML is not configured with an explicit expiration time so the browser assigns a random time that is usually not very long, e.g., 30 minutes. The expiration time of CSS, JavaScript, and images is configured as 1 day, 5 minutes, and 1 year, respectively. The service "/image/address" is configured as no-cache and no-store to ensure obtaining the latest address at every visit. The top table in Figure 1(c) shows the resources in the browser cache after the first visit.

Assume that the app is revisited after one hour and all the related resources have not been updated except the "/image/address". The bottom table in Figure 1(c) shows the cached resources before the second visit. It can be seen that the background image "bg.png" has been removed out of the cache due to the limited size of cache on mobile devices because all the Web apps accessed by a browser share a fixed size of cache space.

Given the current status of cache, when revisiting this app, several resources that could have been loaded from the cache are actually re-downloaded from the network, leading to redundant transfers of resources falling in the following main categories [2], [4].

RT1: Resources that are moved out of the cache.

Due to the imperfect implementation of cache on mobile Web browsers such as limited size and non-persistent storage, resources in the cache may be removed out of the cache after some time. In the preceding example, the background image "bg.png" is removed out of the cache and has to be re-downloaded when the Web app is revisited.

http://m.foo.com/index.html

```
<html>
 <head>
  <link rel="stylesheet", href="a.css">
  <script src="b.js">
 </head>
 <body onload="f()">
  <img id="c">
 </body>
</html>
```

http://m.foo.com/a.css

```
body {
  background-image: url ("bg.png");
}
```

http://m.foo.com/b.js

```
function f() {
  var c = document.getElementById("c");
  c.src = ajax.get("/image/address");
}
```

(a)

URL	Cache Configuration
index.html	cache-control : public
a.css	cache-control : public max-age : 86400
b.js	cache-control : public max-age : 300
bg.png	cache-control : public max-age : 31536000
/image/address	cache-control : no-cache no-store
d.jpg?0.892	cache-control : public max-age : 31536000
d.jpg?0.157	cache-control : public max-age : 31536000

(b)

after 1st visit

index.html
a.css
b.js
bg.png
d.jpg?0.892

before 2nd visit

index.html
a.css
b.js
d.jpg?0.892

(c)

Fig. 1. Motivating example. (a) Resource excerpts of a Web app; (b) Cache configuration; (c) Cache entries between two visits.

RT2: Resources that are wrongly judged as expired.

Each resource has to be configured by developers with a cache policy. Due to the imperfect cache configuration of resources whose expiration time is either configured to be too short or not configured but assigned heuristically by browsers, many resources are incorrectly judged by browsers as expired ones, and have to be validated or re-downloaded. In the example, the HTML resource has not been assigned an explicit expiration time, and the expiration time of the JavaScript resource is configured to be too short. As a result, these two resources cannot be loaded from the local environment when the Web app is revisited.

RT3: Resources that are requested by new URLs but have the same content with cached ones.

Resource Loader of browsers uses URLs to uniquely distinguish resources. Resources with different URLs are regarded as totally different ones. In the example, the same image "d.jpg" has different URLs at two visits, resulting in being fetched twice. URL changing is usually adopted to realize backend load balance according to URL routing. Although such a practice can improve the performance of backend servers, it actually harms the loading process of mobile Web apps.

2.2 Resource Management of Web Apps and Native Apps

One key reason for the preceding redundant transfers is the inefficient resource management of Web apps. Figure 2(a) illustrates the resource-management mechanism of Web apps. Web apps rely on the underlying browsers to manage their resources. All Web apps in a browser share a common cache space whose size is usually small on mobile devices. When a user launches a Web app in the browser (①), resources for rendering the Web app are dynamically identified and all the resource requests are handled by the Resource Loader component in the browser (②). Based on the Web cache mechanism [6], the Resource Loader determines whether to load the resource from the cache (③) or download it from the server (④). After retrieving the resource, the Resource Loader returns it to the Web app (⑤). In summary, Web apps rely on the app-independent browser logics to manage resources. Such a mechanism makes Web apps flexible for resource management so that Web apps can be always up-to-date. However, Web apps cannot have the full control of resources to be loaded from the local storage and when to update the local resources. As a result, redundant transfers arise when the cache policies are not configured properly or the browser removes cached resources.

In contrast, the resource management of native apps works in a different fashion and can be more efficient.

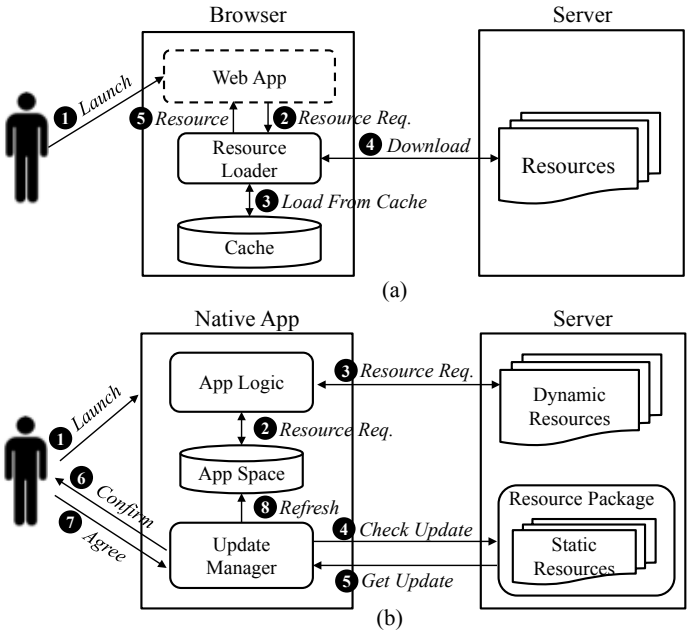

Fig. 2. Resource management of (a) Web apps and (b) native apps.

Figure 2(b) illustrates the resource-management mechanism of native apps. Native apps separate resources into two sets, i.e., the dynamic resource set and the static resource set. Static resources are encapsulated into a resource package. Before using a native app, the resource package has to be installed on the device. When a user launches the app (①), the App Logic controls to load static resources from the App Space (②) and dynamic resources from the server (③). Usually, there is a built-in Update Manager for updating the static resources. The Update Manager checks update with the server (④) in some situations (e.g., every time when the app is launched) to find whether the resource package has been updated (⑤). If a new package is retrieved, the Update Manager confirms with the users whether to update the app (⑥). If agreed (⑦), then the Update Manager refreshes the static resources with the new resource package (⑧). In summary, native apps have app-specific logics to control the resources loaded from the local environment and the update of local resources.

Comparing the two resource-management mechanisms indicates that native apps can manage their resources based on app-specific logic along with resource packages while Web apps cannot precisely manage their resources.

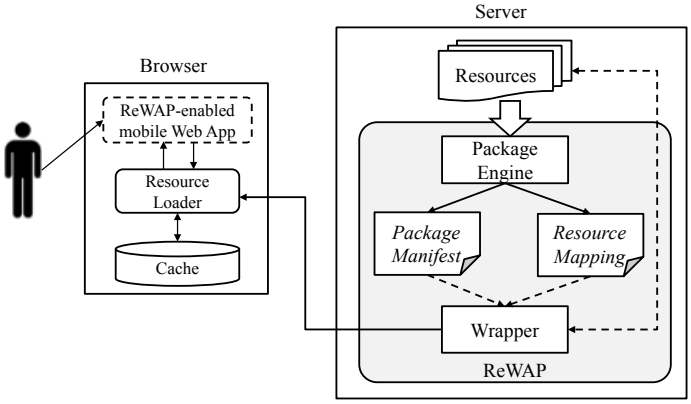

Fig. 3. The ReWAP approach.

The insight underlying our new approach is that redundant transfers originate from the principle of the resource-management mechanism adopted by Web apps.

3 Approach Overview

To fundamentally reduce the redundant transfers, we propose our new solution with the key rationale of lessons learned from the resource-management mechanism used by native apps, while reserving the advantages of the mechanism used by Web apps. More specifically, mobile Web apps can encapsulate stable resources into a package and make the resources in the package always loaded locally rather than being fetched from the servers, while other resources are regularly loaded by the browser's default mechanism. All the resources in the package are refreshed together also by the default mechanism only when the resource package gets updated. The update of the package should follow the way of the Web as well without the intervention of end-users.

To this end, we present the ReWAP approach to restructuring mobile Web apps to be equipped with package-based resource management. ReWAP can accurately identify the resources that should be loaded from the local storage for a considerably long time and that can be refreshed together with minimal cost when the package is updated. Other than the native apps, such a packaging mechanism follows the conventional way of the Web, i.e., the updating and refreshing of packaged resources still use the browser's default cache mechanism. The Web developers can simply integrate ReWAP into their existing mobile Web apps with only minor modifications. As is shown later in this article, the Web developers need only to redirect the entrance of the app to a Wrapper that delegates the resource loading. Meanwhile, the client-side browser performs as usual without additional modifications.

Figure 3 illustrates the overview of the ReWAP approach. ReWAP consists of two major components. The ***Package Engine*** automatically generates and maintains the resource packages of Web apps. The ***Wrapper*** enables the Web apps to use and update resource packages at the local storage. Each Web app has multiple pages and we maintain a dedicated resource package for each page in our current design. In the rest of this article, the term "Web app" actually refers to a single Web page referenced by an HTML document.

By retrieving the update of resources constituting a mobile Web app, the Package Engine generates and maintains a resource package with two configuration files: *Package Manifest* and *Resource Mapping*. The Package Manifest specifies which resources are in the resource package. The update of Package Manifest indicates the update of the corresponding resource package. The Resource Mapping keeps the relationship between URL patterns and unique resource entities. Resources that have the same content but are identified by different URLs are mapped into one resource entity according to Resource Mapping. Therefore, the generated package is highly accurate to cover more resources.

The Wrapper is essentially a separate HTML page where the Web developers can easily enable their mobile Web apps with the package-based resource management. When a ReWAP-enabled mobile Web app is launched, the Wrapper is first fetched from the server. Then the Wrapper controls the loading process on the browser (we use dotted lines to represent the flow taking place on the client side). The Wrapper checks whether the resource package has been updated according to the Package Manifest. If updated, all resources in the package are refreshed and stored into an App-Specific Space according to Resource Mapping. Note that the resource refreshing follows the regular mechanism of Web resource loading so that only new or changed resources incur network traffic to be refreshed. After the refreshing, the Wrapper loads the app and intercepts all the resource requests to determine whether to load a resource from the App-Specific Space or as usual based on the Package Manifest.

ReWAP is deployed as a service on the same server with the target mobile Web app. For example, to integrate ReWAP with the motivating mobile Web app in Section 2, a developer can specify the app's URL http://m.foo.com/index.html in ReWAP and then launches ReWAP service on the server m.foo.com. The Package Engine is then automatically started as a background process at the server side, while the Wrapper is also generated on the server with a URL, e.g., http://m.foo.com/index/wrapper.html. At last, the developer configures the server, making the requests to index.html redirected to the URL of the Wrapper. When a user visits the ReWAP-enabled app, the Wrapper is loaded first to the browser, dealing with resource packages. Then the Wrapper loads the index.html by an AJAX call and intercepts all the resource requests. Overall, deploying ReWAP requires only minimal modifications to existing mobile Web apps.

In the next two sections, we present the technical details on how the Package Engine maintains the resource package and how the Wrapper supports the resource package at the runtime of Web apps.

4 The Package Engine

Figure 4 illustrates the four phases to maintain the resource package: (1) ***Retrieving Resources***; (2) ***Normalizing Resources***; (3) ***Predicting Update Time***; and (4) ***Generating Package***. We then illustrate the details of each phase.

4.1 Retrieving Resources

In the first phase, the Package Engine loads the Web app, retrieves all the required resources, and stores them into the Resource Repository. We define a "*concrete resource*" structure to represent each retrieved resource. A "concrete

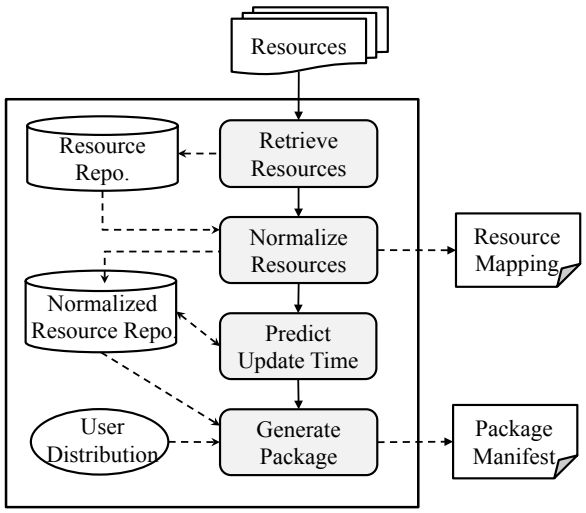

Fig. 4. The workflow of the Package Engine.

resource" has four fields: (1) *URL*, which is the identifier of the resource; (2) *MD5*, which is the checksum of the resource content; (3) *size*, which is the length of the resource content; (4) *cache duration*, which is the configured expiration time. All these fields can be obtained from the HTTP response message. We define R_t as the set of concrete resources retrieved at time t.

The retrieved resources are the basis for package generation. The retrieving process should satisfy the following two requirements.

First, all the resources constituting a mobile Web app should be retrieved so that the package is accurately generated based on the complete set of resources. Since a lot of resources may be missed by only parsing the HTML document, we employ a real browser-runtime facility to actually launch and render the Web app. All the HTTP traffic is recorded in the progress of loading the Web app.

Second, all the updates of resources should be captured so that the package is maintained and updated timely to avoid inconsistencies. For simplicity, we currently use a fixed retrieving frequency to trigger resource retrieving. After a fixed time period, the Package Engine captures the latest status of the Web app and retrieves all the resources.

4.2 Normalizing Resources

In the second phase, the Package Engine identifies the resources that have different URLs but the same content at different visits to the Web app. We denote this kind of resources as "*change-in-name-only resources*" (*CINO* resources in short). We normalize CINO resources into one normalized resource, and keep the relationship of the normalized resource and one concrete resource in the Resource Mapping. The final Package Manifest consists of normalized resources so that CINO resources do not have to be re-downloaded for multiple times.

We observe that there are two frequent URL patterns of the CINO resources. One is the query strings generated by JavaScript, e.g. *Math.random()*, or by server scripts. In the motivating example, the URL of the image "d.jpg" has two different query strings "?892" and "?157" at two visits but the image does not change. In such a case, the URLs vary only in the query, i.e., the random value. The other pattern is the *Content Delivery Network (CDN) prefixes*. At different visits, resources could be retargeted at different CDN servers. In such a case, the paths of the URL are the same but the domain part of the URL could be changed according to the target CDN servers. Based on the two patterns, we assume that the URLs of CINO resources can be different in a certain part of the URL. Therefore, we can apply the *Longest-Substring* algorithm to find the base string of the URLs and use a regular expression to represent the changing part. For example, the image "d.jpg" in the motivating Web app has two different query strings "?892" and "?157" at two visits. So we can use the regular expression "$d.jpg\backslash?*$" to represent the normalized image resource.

We define a "*normalized resource*" structure to represent a unique normalized resource. A normalized resource is generated by aggregating CINO resources. It has all the fields of the "*concrete resource*" structure. The additional fields of "*normalized resource*" include (1) *expression*, which describes the URL pattern of CINO resources; (2) *predicted time*, which describes the estimated duration time that the resource remains unchanged; and (3) *status*, which records all the historic statuses of the resource. We use $status_t$ to denote the status at time t. Each status can be "*inexistent*", "*changed*", or "*unchanged*". Such historic status information is used to predict the update time. We define H_t as the set of normalized resources at time t. H_t is updated every time when concrete resources are retrieved.

Algorithm 4.1 describes the process of managing the set of normalized resources. The key functionality of the normalization is that we should enforce a one-to-one mapping between the normalized resources and the concrete resources of each visit, in order to prevent resources with different contents from being matched to one normalized resource. To this end, we compare the MD5 of the resource content to determine whether two resources are the same. Given the last set of normalized resources H_{t-1}, and the current set of concrete resources R_t, we assign H_{t-1} to H_t at first and initialize all resources' status of time t as "*inexistent*" (Lines 1-4). Then for each concrete resource r, we first check whether there is another resource q whose MD5 is the same with r. If exists, the regular expression is derived and q is annotated as "*unchanged*" (Lines 8-11). Otherwise, we check whether r's URL could match to any regular expressions of normalized resources in the set. If only one match is found, then the matched resource is updated as "*changed*" (Lines 12-15). If more than one match are found, then all the matched resources are removed and the new resource is added to H_t (Lines 16-19). Finally, we need to handle conflicts in the new set of normalized resources to ensure the one-to-one mapping (Line 21).

4.3 Predicting Update Time of Resources

In the third phase, we infer whether a resource is sufficiently stable by predicting the update time of the resource. We assume that the evolution history of a resource can reflect the trend of resource updates. For example, if a resource is updated every day in the history, it is likely to be updated in the next day. Therefore, we design an algorithm to predict the update time of resources based on their evolution histories, as shown in Algorithm 4.2.

By examining the evolution histories of some resources, we find that after a resource disappears at one visit, the possibility of its reappearance is rather small. So, every time when an "*inexistent*" status is captured for a resource, we immediately set its predicted time to 0 (Lines 1-3). For other

Input: Last set of normalized resources H_{t-1}, current set of concrete resources R_t
Output: Updated set of normalized resources H_t

```
1  INITIAL H_t ← H_{t-1};
2  foreach h ∈ H_t do
3  |  INITIAL h.status_t ← "inexistent";
4  end
5  foreach r ∈ R_t do
6  |  P ← FindSameURL(H_t, r);
7  |  q ← FindSameMD5(H_t, r);
8  |  if q ≠ null then
9  |  |  q.expression ←
   |  |  CalRegExpr(q.expression, r.URL);
10 |  |  q.status_t ← "unchanged";
11 |  end
12 |  else if P.size = 1 then
13 |  |  P.status_t ← "changed";
14 |  |  UpdateResource(P);
15 |  end
16 |  else
17 |  |  RemoveResource(P);
18 |  |  AddResource(r);
19 |  end
20 end
21 CheckMapping(R_t, H_t);
22 return H_t.
```

Algorithm 4.1: Normalize resources.

resources, we mainly capture the total times of the "*changed*" status and predict the next time when the resource is likely to change. In some cases, the resource can update in an unusual fashion, so we should not aggressively change the predicted time. For example, if a resource is updated once every day in the history and at one time it is updated one hour after the last update, we should use a modest way to reduce the predicted time. Here, we use the gradient descent algorithm [7] to predict the resource update time (Line 5). Furthermore, if no "*changed*" status is captured, we set the predicted time according to the number of the "*unchanged*" status other than infinite (Lines 6-8). Finally, we remove all the resources whose predicted time is 0 in order to limit the number of historic resources (Lines 10-12).

4.4 Generating Package

In general, the static resources of a mobile Web app should be encapsulated into the resource package so that the package can keep stable for a considerable time length. However, due to the dynamics of current mobile Web apps, there is no ever-clear boundary between static resources and dynamic resources. As a result, we use a revenue-based technique to generate resource packages. We regard that resources in the package can save data traffic for the users of the mobile Web app. The set of resources that could save the largest data traffic shall form the package. Given that different users may revisit the Web app at different time and with different frequencies, the saved data traffic can vary a lot. To measure the overall saved data traffic with the packaged resources, we assume a user distribution function σ to represent the percentage of users at different revisiting intervals. We define a metric, namely the *average saved data traffic*, to quantify

Input: Historic status $status_0, \ldots, status_t$ of a normalized resource $h \in H_t$, visiting interval vi
Output: Predicted update time of h

```
1  if h.status_t = "inexistence" then
2  |  h.predictedtime ← 0;
3  end
4  else
5  |  h.predictedtime ← GDM(status_0, ..., status_t);
6  |  if h.predictedtime = inf then
7  |  |  h.predictedtime ← |status.unchanged| * vi;
8  |  end
9  end
10 if h.predictedtime = 0 then
11 |  RemoveResource(h);
12 end
```

Algorithm 4.2: Predict update time of normalized resources.

how much data traffic all users of the Web app can save on average given the user distribution function.

Then, we present how to choose packaged resources based on the average saved data traffic. Let us assume that a subset of resources $M \subset H_t$ are selected. Suppose that T is the minimum predicted time in M, we then have an expectation that such a resource package M will be updated at time T. For each normalized resource in M, the browser does not need to request this resource before T. On the contrary, without the resource package, each resource should be requested from the server after its cache duration. Thus, for each resource, if T exceeds the cache duration, the traffic saving comes from the difference between the configured cache duration and our predicted time. If the predicted time is less than the cache duration, the resource can still be loaded from the cache according to our package mechanism, incurring no extra traffic.

Algorithm 4.3 shows how to select the best resource package. We first sort the normalized resources according to the predicted time in the ascending order (Line 1). Among all subsets of resources whose minimum predicted time is T, the benefit of a smaller set cannot exceed that of a bigger one. Thus, we do not need to enumerate all potential packages. We enumerate the potential T (Lines 2-10), and calculate the benefit of the largest set whose minimum predicted time is T (Lines 5-10). Finally, we choose the package that provides the maximum benefit (Lines 11-13).

To make the resource package sufficiently stable, we may not always use the resource package with the largest benefit. We first check whether the latest resource package is still valid where all the resources in the latest package has not been updated. If any resource is changed, we just use the package generated by Algorithm 4.3 to replace the invalid package. If the latest resource package is still valid, then we replace the latest package only when the benefit of the new package exceeds the current one by a given threshold.

5 The Wrapper

The functionality of the Wrapper is to equip mobile Web apps with the ability to use the resource packages atop the default resource-management mechanism of Web browsers. Figure 5 shows the workflow of the Wrapper. The Wrapper has an App-Specific Space, which is a dedicated local storage to store packaged resources for each Web app. Each Web

Input: Current set of normalized resources H_t, user distribution σ
Output: Resource package M

Sort H_t based on its predicted time in ascending order;
for $i \leftarrow 0$ *to* $|H_t|$ **do**
  $benefit(i) \leftarrow 0$;
  $T \leftarrow H_i.predictedtime$;
  for $j \leftarrow i$ *to* $|H_t|$ **do**
    if $H_j.cacheduration < T$ **then**
      $benefit(i) += \sigma(H_j.cacheduration, T) * H_j.size$;
    end
  end
end
Select i where $benefit(i)$ is the largest;
$M \leftarrow H_t(i, i+1, \ldots, |H_t|)$;
return M.

Algorithm 4.3: Select the packaged resources.

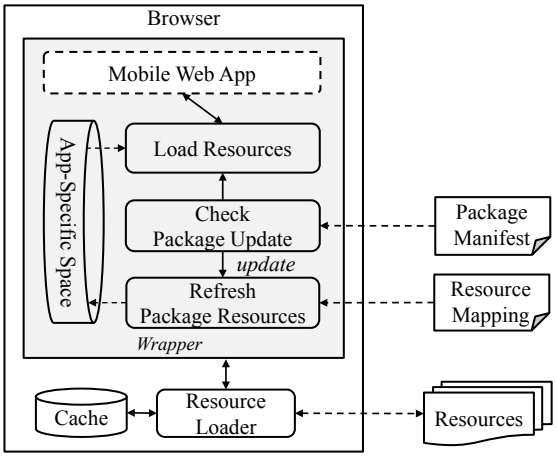

Fig. 5. The workflow of the Wrapper.

app has its own App-Specific Space that is not shared with other Web apps.

When an end-user visits the ReWAP-enabled mobile Web app, the Wrapper is first loaded from the server and runs in the browser. It has three phases to load the target mobile Web app.

Checking Package Update. After the Wrapper is loaded, it checks whether the corresponding resource package has been updated. The Wrapper communicates with the Package Engine to check whether the previously retrieved Package Manifest has been updated. If not updated, then there is no need to refresh the local resources and the Wrapper starts to load the target Web app. Otherwise, if the package has been updated, the Wrapper refreshes the packaged resources stored in the App-Specific Space. Only after the App-Specific Space has finished refreshing, can the target mobile Web app start loading in order to ensure all the resources are up-to-date.

Refreshing Package. When the resource package has to be updated according to the Package Manifest, the Wrapper refreshes the resources in the package based on the regular mechanism of Web resource management. Specifically, for each resource, the Wrapper first checks whether the resource's cache status is expired. If not expired, the resource does not need to be updated. If expired, the Wrapper sends a validation request to check with the original server whether the resource has been changed. Only when the resource is changed, does the Wrapper receive the whole resource. Otherwise, the server returns only a *Not Modified* response. For normalized resources, ReWAP checks the update status of the corresponding concrete resource according to the Resource Mapping.

Loading Resources. When the package update finishes, the Wrapper loads the target Web app. While loading, the Wrapper intercepts all the resource requests emitted by the Web app. For each resource request, the Wrapper checks the App-Specific Space by matching the URL with regular expressions specified in the Package Manifest. If found, then the resource is directly returned to the app from the App-Specific Space. Otherwise, the request is forwarded to the Resource Loader of browsers to retrieve the corresponding resource either from the cache or from the server.

Note that the update check of resource packages and the refresh of resources are actually handled by the Resource Loader of the client-side browsers to manage the resource package with the regular Web mechanism.

6 IMPLEMENTATION

Implementing ReWAP requires that the mobile Web app has an app-specific cache space in order to manage the packaged resources. With the popularity and wide adoption of HTML5, modern Web browsers have provided some APIs for Web apps to control the storage space, such as Web Storage [8], Application Cache [9], and Service Worker [10]. Therefore, ReWAP implemented based on these APIs can run directly on the latest mobile browsers. In this way, we can realize the easy and fast deployment without introducing additional requirements to end-users.

Currently, we choose Application Cache (in short as AppCache) for our implementation. The main reason is that, apart from a dedicated cache space for every single Web app, AppCache also provides a mechanism to check the status for a bunch of resources rather than iteratively checking each resource. This feature can be leveraged to facilitate the implementation of the resource package. We discuss other implementation alternatives in Section 8.

In this section, we first present some background knowledge of AppCache and then describe the details of our implementation and deployment.

6.1 HTML5 Application Cache

The Application Cache (AppCache) is HTML5's feature that aims to allow Web apps to be reliably accessed when the browser is offline. To enable AppCache, developers provide a manifest file to specify what resources are needed for Web apps to work offline, and configure the manifest file to the"*manifest*" attribute of an HTML document's `<html>` tag. The manifest file mainly consists of two sections. Each section has a list of URLs specifying the behavior of the corresponding resources.

- **CACHE.** Resources listed in this section are explicitly cached after they are downloaded for the first time. Even when the browser is online, these resources are still loaded from the AppCache rather than being downloaded from the network. Note that HTML documents referring to manifest files are set in the CACHE section by default.

- **NETWORK.** Resources listed in this section can bypass the AppCache and be requested regularly by the browser. When the browser is offline, these resources cannot be loaded from the AppCache. A wildcard flag "*" can be used to make any resource that is not listed in the CACHE section bypass the AppCache mechanism.

When an HTML document enables AppCache, the loading process of the HTML is different from the regular procedure. If the HTML is loaded for the first time, the manifest file is first downloaded and then all the resources specified in the CACHE section are retrieved. When a resource is requested, the browser first checks whether the URL can be found in the AppCache. If found, the resource is returned directly by the AppCache. Otherwise, the resource is downloaded from the network. If the HTML is revisited after some time when the browser is online, the browser first checks whether the manifest file has been changed on the server by an HTTP validation request. If the manifest is not changed, the browser loads the Web page as usual and resources are retrieved a prior from the AppCache. If the manifest is changed, the browser pauses the HTML parsing, reloads all the resources specified in the CACHE section, and continues HTML parsing.

6.2 Implementation of the Package Engine

We implement the Package Engine in Java as a stand-alone component that can be deployed as a service at the server side. For the browser facility of the Package Engine, we use the Chromium Embedded Framework [11] to render the mobile Web apps and record the HTTP traffic. We currently use a fixed frequency that could be configured by developers to retrieve resource updates of the mobile Web apps. We also assume a heuristic user distribution that the percentage of user revisits equally ranges from 1 minutes to 1 day. The Package Manifest is implemented based on the manifest file of the AppCache. We put the URLs of concrete resources in the manifest and use the generated Resource Mapping file to help check whether a resource is in the package or not by matching with the Resource Mapping.

6.3 Implementation of the Wrapper

The Wrapper is implemented as an HTML page with the AppCache enabled together with a JavaScript library. The Wrapper is totally implemented by standard Web technologies so that it can run directly on modern mobile browsers.

The HTML page is configured to use the manifest file generated by the Package Engine. The App-Specific Space and Check Package Update can be provided by the AppCache. To realize loading the target Web app, the page registers a JavaScript callback function on the `onload` event. When loading the HTML page is finished, the callback function is executed to dynamically fetch the root HTML of the target Web app and modify the DOM tree to render the actual page. Therefore, we can ensure that the retrieved HTML file of the target Web app is up-to-date.

To intercept resource requests, we use JavaScript's reflection mechanism to register callback functions for all the cases of resource requests. When the requested URL matches a URL's regular expression in the Resource Mapping, we replace the requested URL with the corresponding concrete URL to make the resource loaded from the AppCache.

Since the AppCache can work only in the next load after refresh, we explicitly call the `swap()` function of the AppCache when the AppCache's *update* event is triggered in order to make the AppCache use the latest resources.

6.4 Deployment

Given the implementations based on the AppCache, developers can easily deploy ReWAP on their mobile Web architecture, requiring no extra cost to end users.

The whole ReWAP is deployed as a separate service on the Web server. Developers can configure the target mobile Web app to be integrated with ReWAP to launch a certain instance of ReWAP. When ReWAP is launched, the Package Engine is automatically started as a background process on the server, and the Wrapper specific to the configured app is created in a certain folder. While the Package Engine is running, the two configuration files, Package Manifest and Resource Mapping, are also generated in the same folder as the Wrapper. The developers need to only make the folder accessible by the standard HTTP protocol where the Wrapper, Package Manifest, and Resource Mapping are assigned dedicated URLs.

To make the Wrapper work for the target mobile Web app, the developers need to only configure the server to redirect the entrance of the Web app to the URL of the Wrapper. There is no modification of any line of code for the original Web app. When users visit the ReWAP-enabled app, the Wrapper is first loaded to the browser. Nevertheless, the users are unaware of the existence of ReWAP when they request the target Web apps.

7 Evaluations

We evaluate ReWAP from three main aspects. First, we investigate the overall performance of ReWAP by measuring how much data traffic can be saved for mobile Web apps with ReWAP compared to the original apps that are with regular cache mechanisms enabled. Second, we evaluate the performance of the Package Engine, such as the resource normalization and the prediction of update time. Third, we evaluate the overhead of the Wrapper incurred to the mobile Web apps.

7.1 Experiment Setup

The main goal of ReWAP is to reduce data traffic of mobile Web apps when they are revisited. To evaluate how much data traffic can be saved, it is essential to acquire the baseline of how much the data traffic is originally consumed. In addition, the resource package of a Web app is dynamically generated and updated at the same time when resources of the Web app get updated. As a result, we need actual resources of Web apps to evaluate the performance of ReWAP.

Therefore, we use a simulation-based way to conduct our experiments. The idea is that we gather all the resources constituting a mobile Web app at several time points. Then we simulate an ideal browser cache with unlimited storage size to compute the actual data traffic when the app is revisited with different revisiting intervals. Meanwhile, based on the gathered resources, we are also able to simulate the process of ReWAP to generate and maintain the resource packages. In this way, we can not only compute the data traffic when the app is revisited with ReWAP, but also investigate the internal components of ReWAP including resource normalization and update time prediction.

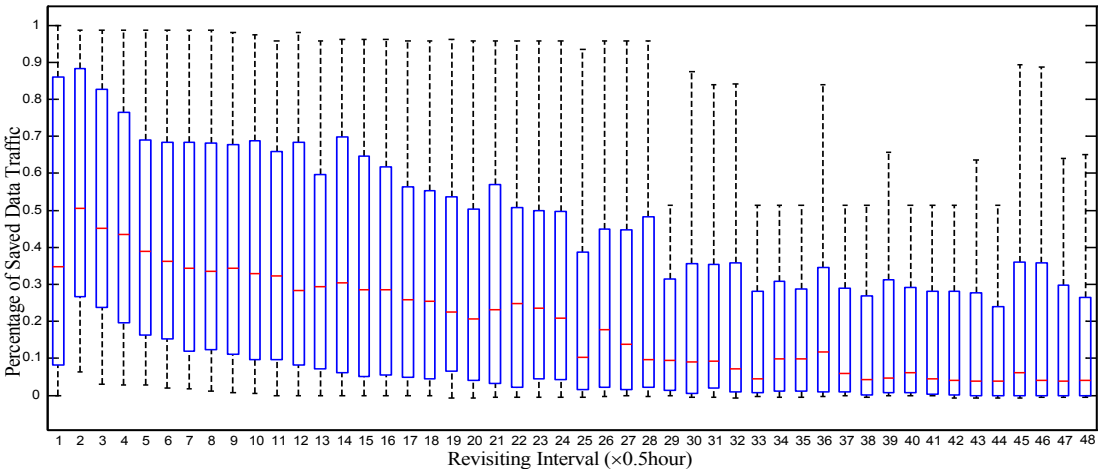

Fig. 6. Distribution of saved data traffic comparing the case with ReWAP and the case without ReWAP.

We randomly choose 50 mobile Web apps[2] from the Alexa top 500 rank list and use the homepage of each app for the evaluations. These mobile Web apps cover typical categories such as news, entertainment, shopping, and business. The size of pages ranges from 38KB to 4MB, representing typical pages of real usage. Then we use the resource-collection platform designed in our previous work [5] to record all the resources of each page every 30 minutes. The platform uses a Chrome browser emulated to retrieve the mobile version of each page by setting the user agent parameter. We assume that the 30-minute interval is sufficiently short to capture the changes of stable resources of Web apps. Our data collection is carried out for 15 days to reduce bias by using sufficiently many samples. Based on this data set, we use the first 5 days' records to train our algorithms of normalization (Algorithm 4.1) and prediction (Algorithm 4.2), and use the last 10 days' records to generate resource packages. We also save all the internal data produced in the progress of package generation to evaluate the Package Engine.

7.2 Overall Performance

The overall performance of ReWAP is measured by how much data traffic it can save for a mobile Web app, compared to the same mobile Web apps that run with regular browser cache mechanism. In fact, the overall performance depends on how a mobile Web app is revisited by its users. For example, users who revisit a Web app every 30 minutes may have more traffic reduction than those who revisit every day, because resources are more likely to change after one day so as to deflate the performance of ReWAP. Therefore, we examine different revisiting at every 30-minute interval, i.e., there are 48 revisiting intervals (0.5 hour, 1 hour, 1.5 hours, ..., 24 hours). For every single revisiting interval ri, we assume that all users revisit the app with the same interval ri, indicating that the user distribution function σ in Algorithm 4.3 is 100% at ri and is 0 at all the other revisiting intervals. Based on the assumption, we can generate resource packages and compare the differences of data traffic consumed by mobile Web apps with and without ReWAP. Note that the data traffic of ReWAP includes the cost of the update of resource packages.

Figure 6 shows the overall performance of ReWAP in terms of saved data traffic. We demonstrate the distribution for each revisiting interval. Each distribution consists of results from all the 50 chosen mobile Web apps. The median of saved data traffic varies from 8% to 51%, indicating that mobile Web apps with ReWAP can reduce averagely 8% to 51% of the data traffic compared to the original Web apps with only browser cache enabled. When the revisiting interval becomes larger, the saved data traffic decreases because the resource package has to be refreshed due to the update of resources in the package.

For each revisiting interval, the distribution of saved data traffic varies from a large range. In the best cases, the saved data traffic can reach almost 100% when revisiting intervals are shorter than 5 hours. In other words, almost all the resources that should be downloaded from the network by the original Web app can be directly loaded from the local storage by ReWAP. In contrast, the largest saved data traffic is only about 50% for most revisiting intervals that are longer than 14 hours. The large variance of saved data traffic is due to the original cache performance of the mobile Web app. If resources of a Web app are configured with proper cache policies, the original data traffic is almost optimized so that there is little room for ReWAP to improve.

7.3 Performance of the Package Engine

The performance of ReWAP is determined by the resource packages generated by the Package Engine. The more captured resources, the more accurate predicted update time and the more stable resource packages, can lead to saving more data traffic. We evaluate the performance of the Package Engine by the intermediate data gathered during the generation of resource packages as illustrated in Section 7.1.

The Package Engine maintains a list of historic resources that are the candidates to be packaged. The list is refreshed every time when new resources are retrieved. Our current design uses a fixed frequency to retrieve resources. It is desirable if the resource list at a certain time t covers more resources at the time later than t so that more resources have the chances to be loaded from the resource package. Here we investigate the resource coverage of the Package Engine. Given a historic resource list H_t at the time t, the resource

2. Please visit https://sites.google.com/site/rewapsys/ for more details.

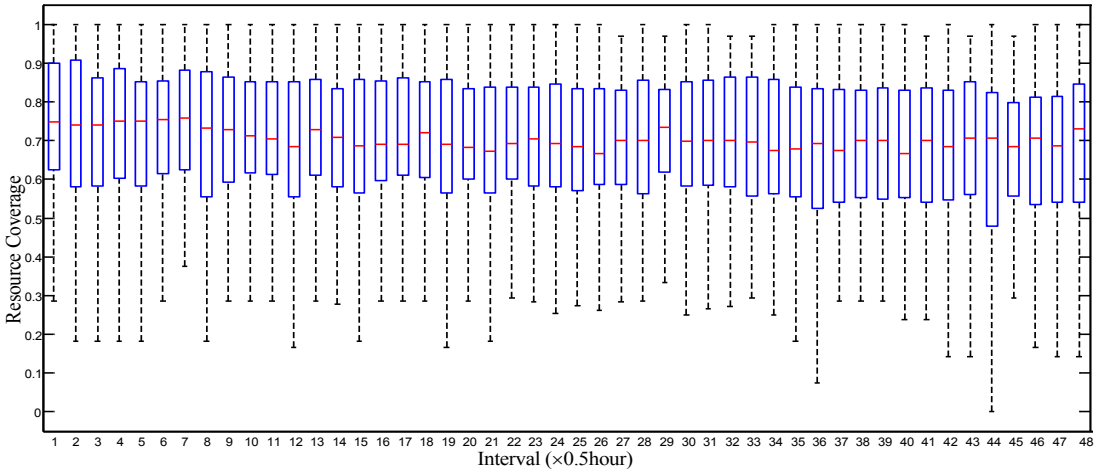

Fig. 7. Distribution of the resource coverage among different intervals.

coverage after an interval i is defined as the number of common resources between H_t and H_{t+i} divided by the number of resources in H_{t+i}. Figure 7 shows the distribution of resource coverage after different intervals ranging from 0.5 hour to one day. We can observe that the median coverage rate is around 70% and it is very stable for different intervals, indicating that about 70% of resources can be covered by the resource list among one day. This result mainly accounts for the contribution of our normalization technique that resources with different URLs but the same content can be normalized to one resource. Therefore, more resources can be covered for longer durations.

The Package Engine uses a predicted update time to judge whether a resource is stable. Since the update time is an important factor in calculating the benefit of resource packages in Algorithm 4.2, the prediction needs to be precise enough. The predicted update time is dynamically adjusted every time when new statuses of resources are retrieved, so we investigate the precision of prediction for different intervals. Figure 8 shows the distribution of the precision of predicted update time. We can observe that the precision decreases as the interval increases. For all the intervals, the median predicting precision is above **85%**. For intervals that are less than 5 hours, the median precision can reach **100%**. Overall, such accuracy can be satisfactory to most apps and can demonstrate the effectiveness of ReWAP. The high prediction precision makes ReWAP better distinguish stable resources so that the resource package could be stable enough to avoid being updated frequently.

Indeed, the more stable the resource packages are, the less refreshing the Wrapper performs. Less refreshing reduces the data traffic to update the resources in the package. We calculate the time duration between every two updates of resource packages. Figure 9 shows the distribution of the duration over the number of package refreshes. We can observe that the median duration of a resource package is 5 hours, indicating that resource packages should be updated every 5 hours in the medium cases. Therefore, the performance of ReWAP is better for revisiting intervals less than 5 hours for the experiment in Section 7.2.

7.4 Overhead of the Wrapper

When a ReWAP-enabled mobile Web app runs on the browsers, the Wrapper can possibly introduce overhead compared to the original app. The overhead lies in two main aspects. One is the manifest file that specifies the resource package. Downloading and refreshing the manifest file need extra data traffic introduced by ReWAP. The other is the computation logics of resource mapping and the Application Cache itself. The computation may affect the client side performance in terms of page load time, CPU and memory usage.

We investigate the size of all the manifest files generated during our experiment. Figure 10 shows the distribution of manifest files' size over the number of manifest files. We can see that the median size is only 5 KB and the largest is not more than 20 KB. Therefore, the overhead of manifest file is quite marginal.

To evaluate the overhead of computation logics and their influences on the page load time, we generate some test pages whose number of resources ranges from 20 to 100, and each resource is 100 KB. We assume that all these resources are put into the resource package and 10% of the packaged resources are normalized resources that are identified by URL regular expressions. Then we visit each page twice in the browser on a smartphone (Samsung Galaxy S4 with Android 5.0 OS) with and without ReWAP. The browser cache is cleared before the first visit to simulate the cold start of the page, and the second visit is triggered just after the first one finishes to simulate the warm load of the page. We record the CPU usage and memory usage as well as the page load time during loading each page with and without ReWAP. We find that the average CPU usage is increased by 15% for pages with ReWAP while the memory usage is of no significant differences.

Figure 11 shows the page load time for different pages in four cases: cold start without ReWAP, cold start with ReWAP, warm load without ReWAP, and warm load with ReWAP. We can observe that as the number of resources increases, the page load time increases in all cases, and the gap between cold start and warm load becomes larger no matter whether the app is equipped with or without ReWAP. For cold start, the page load time of pages with ReWAP is a little longer than that of pages without ReWAP. However, for warm load, the page load time of pages with ReWAP is much shorter than that of pages without ReWAP. This observation implies that ReWAP can also reduce the

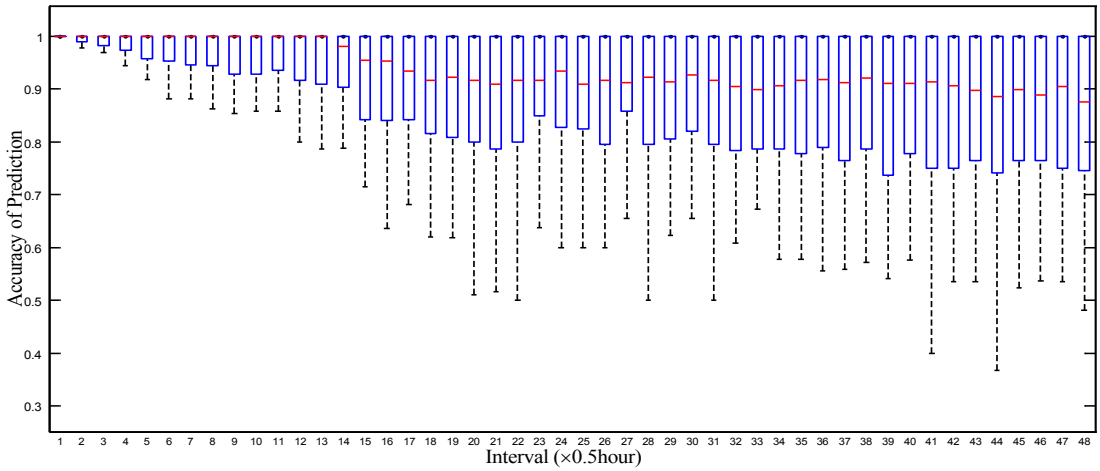

Fig. 8. Distribution of the accuracy of predicted update time among different intervals.

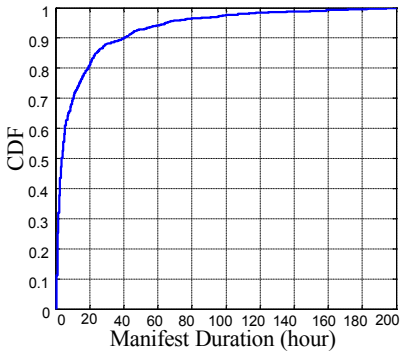

Fig. 9. Distribution of the durations of resource package.

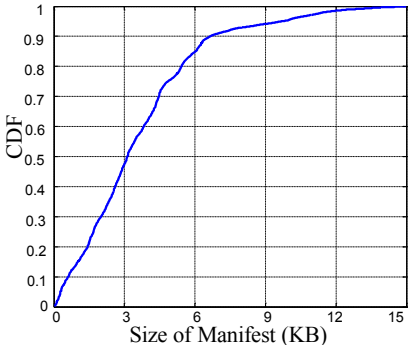

Fig. 10. Distribution of the size of Package Manifest.

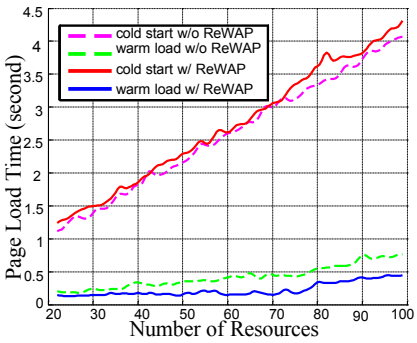

Fig. 11. Influences of ReWAP on page load time for cold start and warm load.

page load time when mobile Web apps are revisited. In the best cases, the page load time is reduced by more than **50%**.

In summary, we can conclude that the overhead of the Wrapper is very minor. ReWAP can significantly reduce the page load time when mobile Web apps are revisited. Therefore, the results demonstrate that ReWAP is practical to be adopted and applied.

7.5 Threats to Validity

The preceding simulated experiments demonstrate the effectiveness of ReWAP. Some issues should be pointed out as they may impact the generalization of these results.

• **Data set.** Our evaluations investigate only homepages from 50 mobile Web apps. Homepages usually have more static resources that can potentially be encapsulated into packages. Therefore, ReWAP may perform better on these homepages. We plan to gather more dynamic pages for further evaluations, such as personal pages in Facebook and pages related to Location-Based Services (LBS) in Yelp.

• **Retrieving interval.** In the experiment setup, we collect the data of resource updates by retrieving resources every 30 minutes. We assume that the 30-minute interval is sufficiently short to capture the changes of stable resources. However, there may be some resources that could change within 30 minutes. As a result, in order to ensure that all the resources are up-to-date, we evaluate the saved data traffic only for users whose revisiting interval is a multiple of 30 minutes. For other revisiting intervals, the saved data traffic shall fall between the numbers corresponding to the two nearest multiples of 30 minutes.

• **User revisiting distribution.** Since ReWAP generates a unique resource package for all the users of a mobile Web app, we use a user distribution function to calculate the benefit brought by different resource packages and choose the one with the largest benefit in our algorithm of generating resource packages (Algorithm 4.3). Certainly, different distributions could lead to different performances of ReWAP. In our evaluation, we use a 100% distribution function for each revisiting interval to be investigated so that the traffic reduction is an upper bound.

• **Size of browser cache.** Our comparison experiment assumes an ideal cache with unlimited storage size, meaning that resources can be always kept in the cache. However, in a real case, the size of cache is limited on mobile devices and cached resources are often removed out of the cache before their configured expiration time. Since ReWAP maintains an app-specific space for each mobile Web app, resources that may be removed out of the browser cache in the original app are also likely to be loaded from the app-specific space. Therefore, more data traffic can be saved by ReWAP in such real case.

8 Discussion

Before ReWAP is deployed in real-world practice, some potential issues need to be discussed and addressed.

8.1 Up-to-date Resource Package

The resource packages maintained by ReWAP should always be up-to-date and keep consistency with the latest version of the Web apps. For simplicity, our current implementation assumes that the Package Engine updates its maintained resources by periodically retrieving the changes of Web apps with a fixed time interval, e.g., every 30 minutes. In practice, if the Web apps happen to change between two "*retrieving time points*" of the Package Engine and the client-side requests arrive just during this interval, the Package Engine cannot provide the latest resource package. Since the Package Engine can be deployed at the same server of the Web apps, one feasible solution is to provide a notification service to inform the Package Engine whenever the Web apps are changed. In this way, resource packages maintained by the Package Engine can be always up-to-date and accurately accessed by clients.

The Package Engine normalizes and generates a regular expression to represent the "*Change-In-Name-Only*" resources. All resources whose URLs can match to the same regular expression are regarded to be identical. A Resource Mapping file is maintained between each regular expression and the corresponding concrete resource. As a result, there may be mismatches caused by the out-of-date Resource Mapping. However, as mentioned previously, the Package Engine can retrieve every change of Web apps with a notification service. In addition, the Resource Mapping is updated at the same time when the Package Engine retrieves the latest resources of the Web app. In this way, the accuracy of resource matching can be preserved.

Resources are selected into packages based on their predicted update time. One can argue that prediction may not be always accurate. However, according to Algorithm 4.3, the accuracy of prediction influences only the stableness of resource packages and further affects how much data traffic could be actually saved. The inaccurate prediction inherently cannot lead to using stable resources or incurring more data traffic. Even for the worst case, the data traffic of ReWAP-enabled Web apps can never exceed that of the original Web apps along with the size of the Package Manifest, because we use the original cache policy of resources as a baseline to select resources into packages.

8.2 Shared Resources Exploration

Our approach currently treats each Web page independently, i.e., each Web page has a dedicated resource package. However, a mobile Web app can consist of many pages. In practice, pages belonging to the same Web app may share a lot of same resources [12]. It is then possible to further improve the performance by packaging resources from all the Web pages of a Web app together. We plan to support such mechanisms in our future work.

8.3 Alternative Implementation of the Wrapper

ReWAP currently relies on the HTML5 AppCache to implement the Wrapper. Other APIs that can provide app-specific space, such as Web Storage [8] and Service Worker [10], are alternatives to AppCache to control the browser-side cache space. We can also analyze the feasibility as well as advantages/disadvantages of these two options. Web storage is a key-value database that can store data for Web apps. The value of Web storage should be string. As a result, if we implement ReWAP based on Web storage, we can place only string-based resources into packages, where binary resources such as images cannot be benefited by ReWAP. Service worker provides a generic entry point for event-driven background processing in the Web Platform. It provides a hook API to intercept resource requests and APIs to access the cache space. If we choose service worker to implement the Wrapper, the request interception can be naturally supported since service worker can launch a stand-alone process to handle the task. However, we should implement the logics to maintain the dedicated cache space, to check updates of resource packages, and to refresh packaged resources if the package is updated. In fact, such logics are now conveniently supported by AppCache. However, in any case of AppCache, Web storage, and service worker, the Wapper's core functionalities of processing normalized resources are always the same. We plan to explore service worker in our future work and compare the developers' cost and actual performance with those of the current ReWAP, in order to decide the more efficient solutions.

8.4 Energy Efficiency

Since ReWAP reduces redundant transfers, ReWAP-enabled mobile Web apps are expected to consume less energy than the original ones. We plan to measure the energy consumption of mobile Web apps with and without ReWAP in our future work. Here we theoretically analyze ReWAP's potential influence on energy consumption. The energy consumption of a mobile Web app has a relationship with the network data traffic as well as the page load time. Less data traffic and shorter page load time could save energy consumption. In Section 7.4, we compare the page load time of 80 synthetic test pages with and without ReWAP in the cases of cold start and warm load. The page load time of ReWAP-enabled pages increases by a small margin in the case of cold start and decreases up to 50% in the case of warm load. Therefore, ReWAP may introduce a little extra energy consumption when the ReWAP-enabled apps are loaded for the first time, but can save energy when they are revisited.

9 Related Work

It is well known that the user experiences of mobile Web apps are far from satisfactory in terms of the page load time, data traffic drain, and energy consumption. Redundant transfers of resources are the most dominant issue leading to such inefficiency. Our work focuses on reducing redundant transfers to improve the user experiences of mobile Web apps. We then discuss related work.

• **Measurement studies on resource loadings of mobile Web apps.** Wang *et al*. [13] advocated that resource loading contributes most to the browser delay. Wang *et al*. [3] designed a lightweight in-browser profiler, called WProf, and studied the dependencies of activities when browsers load a webpage. Nejati *et al*. [14] extended WProf to WProf-M and studied the differences of page loading process between mobile and non-mobile browsers. Li *et al*. [15] designed WebProphet to capture dependencies among Web resources and to automate the prediction of user-perceived Web performance. The poor performance of mobile Web cache is a key issue leading to redundant transfers. Qian *et al*. [2] measured the performance of mobile Web cache in terms of the cache implementation and revealed that about 20% of the total Web traffic examined is redundant due to imperfect cache implementations. In their later work [16], they studied the caching efficiency for the most popular

500 websites and found that caching is poorly utilized for many mobile sites. Wang *et al*. [12] found that cache has very limited effectiveness: 60% of the requested resources either are expired or are not in the cache. Our previous work [4], [5] adopted a proactive approach to measuring the performance of mobile Web cache and found that more than 50% of resource requests are redundant on average for 55 popular mobile websites. In particular, we found some underlying factors leading to redundant transfers of mobile Web apps, i.e., Same Content, Heuristic Expiration, and Conservative Expiration Time. These measurement studies motivate us to reduce redundant transfers for mobile Web apps.

• **Techniques to reduce redundant transfers.** Wang *et al*. [17] investigated how Web browsing can benefit from micro-cache that separately caches layout, code, and data at a fine granularity. They studied how and when these resources are updated, and found that the layout and code that block subsequent object loads are highly cacheable. Our resource packaging can be viewed as to realize similar features proposed by micro-cache in the resource granularity. Most of the stable layout and code resources are put into the package to always be loaded from the local environment. Zhang *et al*. [18] implemented a system-wide service called CacheKeeper, to effectively reduce overhead caused by poor Web caching of mobile apps. CacheKeeper can also work for browsers but it relies on the support of operating systems. Our implementation of ReWAP utilizes the standard HTML5. HTML5 has been supported by all the commodity mobile Web browsers. So we can achieve fast and easy deployment with no cost to end-users.

• **General techniques to improve the performance of mobile Web.** Some previous work focuses on improving the compute-intensive operations for mobile Web browsers, such as style formatting [19], layout calculation [20], [21], and JavaScript execution [22], [23]. However, Wang *et al*. [13] argued that the key to improve the performance of mobile Web is to speed up resource loading. Various solutions have been proposed to optimize resource loading. These solutions include new network protocols such as SPDY [24] and HTTP2 [25], browser optimization such as prefetching [26] and speculative loading [12], and proxy-based systems such as Flywheel [27] and KLOTSKI [28]. These previous solutions are orthogonal to reducing redundant transfers.

• **Dynamics and user revisits of Web pages.** Douglis *et al*. [29] performed a live study on the influences of resource changes and user revisits on the Web caching in early 1997. Fetterly *et al*. [30] measured the degree of Web page changes and investigated the factors correlated with change intensity. Adar *et al*. [31] studied the Web revisiting behaviors from a live data set. They identified four revisiting patterns for different kinds of Web pages. In their later work [32], they studied the relationship between the dynamics of Web pages and user revisiting patterns. Although all these previous efforts focus on the Web for desktop computers, their findings can be partly leveraged by the mobile Web. Our work depends on the dynamics and user revisits of Web pages to maintain the resource package.

10 Concluding Remarks and Future Work

In this article, we have presented the ReWAP approach, by radically making the resource management of mobile Web apps perform in a similar fashion to native apps. The key rationale of ReWAP is to provide more efficient and "application-aware" control of resource management rather than relying on only the current mechanisms such as Web cache, to avoid the caused unnecessary redundant resource transfers. To realize the efficient resource packaging, ReWAP includes a normalization technique to identify the same resources but with different URLs, a learning-based technique to accurately predict the updates of resources, and an algorithm to minimize the refresh frequency of resource packages as well as reducing the overhead. We have evaluated ReWAP based on long-term (15-day) traces of existing mobile Web apps and the results demonstrate our approach's effectiveness and efficiency.

Given that ReWAP can be easily deployed into existing Web apps with very few manual efforts, our ongoing work is to encapsulate ReWAP as an independent module into currently popular Web servers such as Apache, Nginx, and Node.js. When ReWAP is deployed and the actual access logs are obtained, we can derive the distribution of a user's visit frequency for a given Web app, and thus optimize the prediction algorithm by tuning the parameters and designing online learning kernels. Apart from the data traffic and page load time, other performance issues, e.g., the energy drain, need to be addressed as well in future work.

Acknowledgment

This work was supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202, the Natural Science Foundation of China (Grant No. 61370020, 61421091, 61528201). Tao Xie's work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1434582, CCF-1434596, CNS-1513939, CNS-1564274.

References

[1] N. Serrano, J. Hernantes, and G. Gallardo, "Mobile web apps," *IEEE Software*, vol. 30, no. 5, pp. 22–27, 2013.
[2] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Web caching on smartphones: ideal vs. reality," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys 2012*, 2012, pp. 127–140.
[3] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with WProf," in *Proceedings of USENIX Conference on Networked Systems Design and Implementation, NSDI 2013*, 2013, pp. 473–485.
[4] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie, "Measurement and analysis of mobile web cache performance," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015*, 2015, pp. 691–701.
[5] X. Liu, Y. Ma, Y. Liu, T. Xie, and G. Huang, "Demystifying the imperfect client-side cache performance of mobile web browsing," *IEEE Transactions on Mobile Computing*, vol. 15, no. 9, pp. 2206–2220, 2016.
[6] "RFC 2616." [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.txt
[7] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Proceedings of the 21st International Conference on Machine Learning, ICML 2004*, 2004, pp. 919–926.
[8] "Web Storage." [Online]. Available: https://www.w3.org/TR/webstorage/
[9] "Application Cache." [Online]. Available: https://www.w3.org/TR/2011/WD-html5-20110525/offline.html
[10] "Service Worker." [Online]. Available: https://www.w3.org/TR/service-workers/
[11] "Chromium embedded framework." [Online]. Available: https://bitbucket.org/chromiumembedded/cef/
[12] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, "How far can client-only solutions go for mobile browser speed?" in *Proceedings of the 21st International Conference on World Wide Web, WWW 2012*, 2012, pp. 31–40.
[13] ——, "Why are web browsers slow on smartphones?" in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile 2011*, 2011, pp. 91–96.

[14] J. Nejati and A. Balasubramanian, "An in-depth study of mobile browser performance," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016*, 2016.
[15] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "WebProphet: Automating performance prediction for web services," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI 2010*, 2010, pp. 143–158.
[16] F. Qian, S. Sen, and O. Spatscheck, "Characterizing resource usage for mobile web browsing," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2014*, 2014, pp. 218–231.
[17] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "How much can we micro-cache web pages?" in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014*, 2014, pp. 249–256.
[18] Y. Zhang, C. Tan, and L. Qun, "Cachekeeper: A system-wide web caching service for smartphones," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2013*, 2013, pp. 265–274.
[19] H. Wang, M. Liu, Y. Guo, and X. Chen, "Similarity-based web browser optimization," in *Proceedings of the 23rd International World Wide Web Conference, WWW 2014*, 2014, pp. 575–584.
[20] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*, 2010, pp. 491–500.
[21] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering refactoring opportunities in cascading style sheets," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, 2014, pp. 496–506.
[22] 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, MobiSys 2010*, 2010, pp. 165–178.
[23] L. Gong, M. Pradel, and K. Sen, "JITProf: pinpointing jit-unfriendly Javascript code," in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015*, 2015, pp. 357–368.
[24] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is SPDY?" in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, 2014, pp. 387–399.
[25] "HTTP/2." [Online]. Available: https://http2.github.io/
[26] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "PocketWeb: instant web browsing for mobile devices," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, APSLOS 2012*, 2012, pp. 1–12.
[27] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin, "Flywheel: Google's data compression proxy for the mobile web," in *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, 2015, pp. 367–380.
[28] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, 2015, pp. 439–453.
[29] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. C. Mogul, "Rate of change and other metrics: a live study of the world wide web," in *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems, USITS 1997*, 1997.
[30] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener, "A large-scale study of the evolution of web pages," in *Proceedings of the 12th International World Wide Web Conference, WWW 2003*, 2003, pp. 669–678.
[31] E. Adar, J. Teevan, and S. T. Dumais, "Large scale analysis of web revisitation patterns," in *Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008*, 2008, pp. 1197–1206.
[32] ——, "Resonance on the web: web dynamics and revisitation patterns," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009*, 2009, pp. 1381–1390.

Xuanzhe Liu is an Associate Professor in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests are in the area of services computing, mobile computing, web-based systems, and big data analytic. He is a member of the IEEE.

Yun Ma is a Ph.D student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include services computing and web engineering.

Shuailiang Dong is an undergraduate student from Peking University majoring in computer science and technology. His research interests include mobile network and software engineering.

Yunxin Liu is a Lead Researcher in Microsoft Research. His research interests include mobile systems and networking.

Tao Xie is an Associate Professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at Urbana-Champaign, USA. His research interests are software testing, program analysis, software analytics, software security, and educational software engineering. He is a senior member of the IEEE.

Gang Huang is a Full Professor in Institute of Software, Peking University. His research interests are in the area of middleware of cloud computing and mobile computing. He is a member of the IEEE. He is the corresponding author of this article.