Rethinking Client-Driven Resource Management for Mobile Web:
Measurement, Deployment, and Runtime

Xuanzhe Liu¹, Yun Ma¹, Felix Xiaozhu Lin²
¹Key Lab of High-Confidence Software Technology (Peking University), MoE, PRC, Beijing, China, 100871
²Purdue University, West Lafayette, IN, 47906
Email: {xzl, mayun}@pku.edu.cn, xzl@purdue.edu

Abstract—After its birth since early 1990s, the Web has been the major factor that drives the success of the Internet. In the past decade, the access to the Web has undergone a tremendous evolution from PC to mobile devices, i.e., via smartphones, tablet computers, and wearable devices. From the software system perspective, it is a key challenge to make the future mobile Web more “user friendly”, i.e., smooth interactions, short page load time, reasonable data traffic volume, efficient energy drain, etc. This paper introduces our recent efforts that propose holistic resource management from the client-driven perspective, including the thorough measurement and root cause analysis, a new Web app deployment model, and a new intelligent middlebox-based runtime.

1. Introduction

When Sir Tim Berners-Lee invented the first Web in early 1990s, the Web, or the Browser-Server architecture, has already become the most successful and popular application on the Internet. Web applications, (abbreviated as Web apps in the remainder of this paper) spread in almost every domain. In the past decade, the growing advantages in communication and network technologies are changing the way people interact with Web apps, providing them with different types of devices for access at any time from anywhere, with any services, and contents customized to users’ preferences and usage environment. In particular, the announcement of Apple iPhone in 2007 opened a new era for the Web. The users no longer access the Web apps only from the Web browsers on desktop PCs. Instead, the mobile devices play as the main access channel to access the Web apps. In the past decade, it is observed that more and more cloud service providers release Web apps by which users can access the cloud services from mobile devices. With the advances of popular Web technologies such as HTML5, JavaScript, CSS, and powerful library support, the mobile Web apps are supposed to provide comparable functionalities and user experiences of desktop and native applications. Current Web technologies can realize quite complex features, covering online video/audio streaming, games, and even virtual reality (VR) and augmented reality (AR) applications. In addition, even millions of native applications are actually built by leveraging the RESTful Web APIs (Application Programming Interfaces) and the embedded browser kernels such as WebKit or WebView to provide functionalities and GUIs in a Web-like way. More and more applications are “born on the Web”.

In a sense, we live on, work by, communicate with the mobile Web, and enjoying more friendly experiences is our intuitive requirement. Indeed, the user experience is subjective and may be characterized with different metrics. From the software system perspective, besides the adaptive layout and touch-oriented control in user interface, we require the high quality of user experience (QoE) of the mobile Web in various other aspects. For example, we need the page load and rendering process of the mobile Web apps to be efficient under dynamic network connections and on different hardware models whose computation power can vary a lot. Fast page load time can also help reduce the lightening time of the device screen, which in turn improves the battery life. Meanwhile, under mobile computing environment, the data traffic volume is a non-trivial issue for end-users who usually pay for limited cellular data plan.

Optimizing the preceding quality-of-experience metrics of mobile Web has raised a lot of challenges and opportunities for system research. Compared to traditional approaches that are mainly based on the server-side resource management, the specific features of mobile Web computing, such as diverse mobile device hardware, the dynamics of network connections, and the various behaviors of users, require higher adaptive and personalized resource management, and thus make the client-driven an emerging direction [1].

In the past a few years, we have proposed a holistic client-driven approach, by rethinking the resource consumption measurement, the new application deployment models, and the new browser-server runtime. This vision paper will summarize these efforts that have been published on related venues (including WWW [2], IMC [3], TMC [4], [5], [6], TOIT [7]) and discuss issues in progress.

This paper is organized as follows. Section 2 presents the background of Web page load and the client-side measurement along with the results. We then describe two orthogonal solutions at the application-level and runtime-level, respectively: Section 3 describes the ReWAP approach that promises very tiny developer efforts to refactor current mobile Web apps, and Section 4 describes the middlebox-based runtime to optimize the resource loading. Section 5 discusses about other in-progress research efforts that are
Figure 1. General Web Page Load Processes

comprised in the total approach. Section 6 ends up the paper.

2. What Matters in Web Page Load

2.1. General Workflow of Web Page Load

Usually, opening a web page requires three main processes, as shown in Figure 1: (1) **Network**: usually users need to initiate requests to a target URL and fetch corresponding resources from a remote server (either from local cache without network). (2) **Parsing**: when browser receives the response of first chunk of the root page, the HTML Parser will iteratively parse the page and download all objects embedded in that page, including HTML documents, Cascading Style Sheets (CSS), JavaScript, images, etc, and transform these objects to a Document Object Model (DOM) tree. The DOM tree maintains the intermediate representation and provides common interface for programs to manipulate of the page. (3) **Rendering**: the browser generates layouts and paints the page, by computing and updating the screen locations from the intermediate representation based on some rules, e.g., CSS may change DOM nodes for specific color, and generates the final graphical representation. Roughly, we can group Parsing and Rendering as **Computation** that are locally performed on device.

In practice, client-side cache is particularly helpful for resource loading on mobile devices. By saving the resources of visited webpages into the local storage facilities (memory or disk), these webpages may be served from the local storage rather than being re-fetched from the Web servers when the webpages are accessed again.

Indeed, it is significant to understand how mobile Web browsing could benefit from “client-side cache” and how well “client-side cache” takes effect in practice. To this end, several measurement studies have been conducted by analyzing user access traces gathered from Internet Service Providers (ISPs) or instrumented client devices [8], [9], [10], [11].

2.2. Proactive Client-Side Resource Measurement

Although some existing studies have already revealed the performance problems of client-side cache on mobile devices, they focus on only the imperfect client-side implementations. They do not comprehensively characterize the end-to-end Web cache performance involving not only Web clients, but also Web servers and user behaviors. More specifically, the existing measurement studies on mobile Web cache have various limitations.

To better understand the client-side cache efficiency, we have proposed a proactive approach to acquiring detailed history of resource update [4]. We choose two sets of websites for a comparative study to uncover both the state-of-the-art and ordinary status of mobile Web cache performance. One set (Top) comes from Alexa’s Top 100 list. The other set (Random) comes from 100 websites randomly chosen from Alexa’s Top 1,000,000 list. We periodically crawl their resource update history for one week, forming the basis of our analysis. We then designed a measurement model for analyzing cache performance. Based on the resource update history, we formally define the metrics to quantitatively measure cache performance. We then build a cache behavior taxonomy to correlate cache configurations to resource updates. Such a taxonomy maps each possible case to one of the four behavior patterns: Positive Hit (PH), Negative Hit (NH), Positive Miss (PM), and Negative Miss (NM). The derived findings are listed in Table 1.

Based on the findings, we have demystified the root causes for undesirable mobile Web cache performance. Three main root causes are identified:

- **Same Content**: the same resources that have different URLs when requested at different times.
- **Heuristic Expiration**: the caching policy is not explicitly defined by the server and thus it depends on browsers to infer an expiration time.
- **Conservative Expiration Time**: the expiration time is set to be too short.

2.3. Imperfect Client-Side Web Cache

The preceding factors can lead to redundant transfers of resources, which in turn significantly compromises the mobile Web efficiency. We illustrate redundant transfers via an example shown in Figure 2.

Figure 2(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
<table>
<thead>
<tr>
<th>No.</th>
<th>Findings from Measurement Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>More than half of the total resource bytes and about 70% of the resource objects can be cached after 24 hours. Although 80% of the cacheable bytes are actually cached for Top, the percentage of cached objects is under 50%. For Random, both cached bytes and objects occupy more than 50% of the cacheability.</td>
</tr>
<tr>
<td>2</td>
<td>The cacheability varies for different types of resources, and the actual performance is also different. The results between Top and Random are not the same.</td>
</tr>
<tr>
<td>3</td>
<td>The Redundant Transfer problem is worse when a website is revisited after a shorter period. Some popular websites even have 100% Negative Miss Ratio.</td>
</tr>
<tr>
<td>4</td>
<td>NH of Top and Random share a similar proportion. HTML accounts for the largest NH. NH of CSS for Top is larger than for Random, while JS is on the contrary. NH of images is not 0.</td>
</tr>
<tr>
<td>5</td>
<td>Same Content accounts for the main source of NM. More than 20% of NM resources are resulted from Same Content for half of Top and 25% of Random. Same Content is caused by the version management and Web hierarchy architecture.</td>
</tr>
<tr>
<td>6</td>
<td>6% and 33.7% of Top and Random resources are configured as Heuristic Expiration, but the average updating cycle is 5.9 hours and 107 hours for Top and Random, respectively.</td>
</tr>
<tr>
<td>7</td>
<td>40% and 24% of the resources for Top and Random are configured by less than one-day expiration time. But 89% and 86% of them have no updates for Top and Random, respectively.</td>
</tr>
</tbody>
</table>

### TABLE 1. SUMMARY OF FINDINGS

<table>
<thead>
<tr>
<th>No.</th>
<th>Findings from Measurement Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>More than half of the total resource bytes and about 70% of the resource objects can be cached after 24 hours. Although 80% of the cacheable bytes are actually cached for Top, the percentage of cached objects is under 50%. For Random, both cached bytes and objects occupy more than 50% of the cacheability.</td>
</tr>
<tr>
<td>2</td>
<td>The cacheability varies for different types of resources, and the actual performance is also different. The results between Top and Random are not the same.</td>
</tr>
<tr>
<td>3</td>
<td>The Redundant Transfer problem is worse when a website is revisited after a shorter period. Some popular websites even have 100% Negative Miss Ratio.</td>
</tr>
<tr>
<td>4</td>
<td>NH of Top and Random share a similar proportion. HTML accounts for the largest NH. NH of CSS for Top is larger than for Random, while JS is on the contrary. NH of images is not 0.</td>
</tr>
<tr>
<td>5</td>
<td>Same Content accounts for the main source of NM. More than 20% of NM resources are resulted from Same Content for half of Top and 25% of Random. Same Content is caused by the version management and Web hierarchy architecture.</td>
</tr>
<tr>
<td>6</td>
<td>6% and 33.7% of Top and Random resources are configured as Heuristic Expiration, but the average updating cycle is 5.9 hours and 107 hours for Top and Random, respectively.</td>
</tr>
<tr>
<td>7</td>
<td>40% and 24% of the resources for Top and Random are configured by less than one-day expiration time. But 89% and 86% of them have no updates for Top and Random, respectively.</td>
</tr>
</tbody>
</table>

![Figure 2. Motivating example. (a) Resource excerpts of a Web app; (b) Cache configuration; (c) Cache entries between two visits.](image)

![background image “bg.png” is identified. When parsing the HTML resource is finished and the onload event is triggered, JavaScript function f is executed and an image is loaded. Note that the URL of the target image is generated by the JavaScript code that attaches a random string of numbers to “c.jpg” (e.g., 0.771). Therefore, when the app is visited at the first time, 5 resources are actually retrieved. Figure 2(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 top table in Figure 2(c) shows the resources in the browser cache after the first visit.](image)

Suppose that a user revisits the webpage after one hour, and none of the resources has been updated yet since last time’s visit. The bottom table in Figure 2(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 data transfers (RDT) of resources falling in the following main categories 2, 13.

**RDT1: 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.

**RDT2: 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.

**RDT3: 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, when the JavaScript function f is executed again, a new random string of numbers is attached to “c.jpg” (e.g., 0.461). As
a result, the browser cannot find the resource in the local cache and has to download it again from the network. 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.

2.4. Optimizing Client-Side Resource Management

To address the imperfectness of current client-side resource management, we have proposed two solutions. The first one is at the application level, and slightly refactors existing mobile Web apps with very minimal developer efforts along with the current deployment model. The second one is at the runtime level, and does not instrument existing applications but redefines the resource loading process by employing the middlebox that can be deployed on edge and third-party cloud services. We describe each solution in Section 3 and Section 4, respectively. So far our solutions deal with different Web pages in a Web app separately. In the next two sections, the term “Web app” refers to a single Web page referenced by an HTML document.

3. Refactoring Mobile Web into “Native”

We first describe how to refactor existing mobile Web apps to reduce the impact of imperfect cache. The core idea is to “radically” make the resource management perform in a “native-like” fashion.

3.1. Root Cause Analysis of Web Cache

One key reason for the preceding redundant transfers is the inefficient resource management of Web apps. Figure 3(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 [14], 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.

3.2. Learning Lessons from Native Apps

In contrast, the resource management of native apps works in a different fashion and can be more efficient. Figure 3(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. The insight underlying our new approach is that redundant transfers originate from the principle of the resource-management mechanism adopted by Web apps.
3.3. Redesigning Resource Management

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 keeping 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.

3.4. The ReWAP System Design

To this end, we have presented 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 browser performs as usual without additional modifications.

Figure 4 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.

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 Figure 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.

3.5. Experimental Evaluation Summary

We have conducted 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. We demonstrate the distribution for each revisiting interval in Figure 5. 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.

ReWAP relies on the Package Engine that predicts the resource update and pushes the changes to client. 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 \( H_i \) at the time \( t \), the resource coverage after an interval \( i \) is defined as the number of common resources between \( H_i \) and \( H_{i+1} \) divided by the number of resources in \( H_{i+1} \). Figure 6 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.

More details can be referred in our work [5].

3.6. Challenges and Discussions

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

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.

Shared Resources Exploration. The ReWAP 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 [1]. 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.

4. Leveraging MiddleBox at Runtime

Compared to the ReWAP that require developers to make some manual efforts including refactoring their existing legacy applications and deploying some modules on their web servers, we propose a runtime support to optimize the resource loading without any modification. Note that such a runtime is orthogonal to the ReWAP design that needs refactoring efforts.
4.1. Potential Improvements at Runtime

We begin with analyzing the resource loading of the motivating example in Section 2.4.

Figure 7(a) shows the resource loading sequence of the two visits, i.e., when the web app is visited for the first time and revisited after one hour. Each bar represents the time needed for loading the corresponding resource. For the first visit, the HTML resource starts to be loaded at \( t_0 \). Then at \( t_1 \), CSS and JavaScript resources start to be loaded and their loading finishes at \( t_3 \) and \( t_4 \), respectively. While the CSS is being loaded, the background image “bg.png” starts to be loaded at \( t_2 \). During the loading of JavaScript, the HTML parsing pauses temporarily. After the HTML parsing finishes and the JavaScript function \( f \) is executed, the image “c.jpg” starts to be loaded at \( t_5 \) and finishes at \( t_6 \). For the second visit, since HTML, JavaScript, and images resources have to be downloaded again from the network, the total resource loading time cannot be reduced, resulting in the same page load time.

The preceding webpage loading suffers from some inefficiencies. “a.css” and “b.js” are loaded after “index.html” is parsed, “bg.png” is loaded after the CSS is evaluated, and “c.jpg” is loaded after the JavaScript code is executed. For the second visit, “c.jpg” is fetched again from the server because it is identified by a random but different number. Neither “index.html” nor “b.js” is changed but both are also re-fetched from the server in the second visit due to cache expiration.

Ideally, the loading of the example webpage should be like Figure 7(b). For the first visit, if we could decide the needed resources before they are requested, we could download them altogether. For the second visit, if we could know that “c.jpg?r=0.461” and “c.jpg?r=0.771” essentially refer to the same resource, and neither “index.html” nor “b.js” is changed, we do not need to re-download any resources at all. Consequently, the page load time can be significantly reduced for both visits.

4.2. Principled Design for Runtime Optimization

Based on the findings of motivating example, we then aim to redefine the resource loading for mobile Web browsing, along with the following design goals.

Minimal page load time. The page load time is a key metric of Web browsing. We aim to accelerate the process of loading Web resources to reduce the page load time and improve the user experience.

Minimal network traffic. Smartphone users usually have limited data plan of cellular networks. We aim to reduce unnecessary data transmission and save data plan for users in the case of cellular networks.

No modifications of browsers or servers. For ease of deployment, we should require no modifications to existing Web browsers and Web servers, so that our solution can be seamlessly and easily deployed into current Web architecture.

Minimal system overhead. The computing capabilities (e.g., CPU, RAM, and GPU) of mobile devices are limited. We should provide a sufficiently lightweight solution to prevent the costs from surpassing the benefits.

4.3. Page Preloading on MiddleBox

To realize the preceding design goals, there are three major challenges to address in our runtime system with respect to the imperfectness mentioned in Section 2.

First, we should push resources to a client as soon as possible, even before they are actually requested. Doing so requires knowing the resources of a webpage as well as their loading order in advance. Since modern webpages become increasingly complex and dynamic, the resources of a webpage cannot be statically determined in advance before the webpage is actually rendered. When static determination is conducted, some of the resources may be missed and the order of resource pushing may be different from the order of resource loading. It is also challenging to avoid pushing unused resources. We should push only those resources that are really required.

Second, we should prevent duplicated resource loading. Loading Web resources needs to determine whether the resources with different URLs have the same content. When a browser sends a request to load a resource identified by URL \( u_1 \), the system needs to know whether a previously downloaded resource identified by URL \( u_2 \) has the same content with the resource identified by \( u_1 \). If these two resources are matched, the browser could just load resource \( u_2 \) to avoid redundantly downloading \( u_1 \) from the server. The challenge here is that our system should provide an efficient matching mechanism to find the target resource with minimal cost, as the matching is performed on a mobile device and the potential searching space can be potentially large. The matching should also be accurate enough to avoid loading wrong resources that may compromise the functionalities of the webpage.
Third, we should judge redundant resource loading due to imperfect cache configurations. Loading Web resources has to determine whether a cached resource has been already changed or not at the server since the last time of caching. If not changed, the browser can just load the cached resource locally instead of downloading it again. Therefore, no matter what expiration time the Web developer has configured for a resource, all the requests to the resource except the first one should never trigger downloading of the resource unless it has been changed. Since the change of resources could happen at any time and there is no mechanism to inform such changes, it is challenging for browsers to get the information of resource updates.

In summary, loading Web resources has to acquire some knowledge in advance in order to accelerate the loading process for mobile devices. Such knowledge includes identification of unique resources, resource update information, as well as resource loading order. To acquire this knowledge, our system has to preload the target webpage in advance because only by rendering the webpage can our system obtain the correct and complete resource information. However, the preloading itself consumes network traffic and introduces extra latency. As a result, our system should not perform the preloading process on mobile devices. Our system should be designed to balance different design goals to optimize the resource loading process.

4.4. The SWAROVsky System Design

To achieve our design goals and tackle the challenges, we have designed and implemented the SWAROVsky (Smart Web Acceleration by Resource Optimization oVer the sky) [6]. The key idea of our system design is to leverage the middlebox-based dual-proxy architecture, i.e., deploying a remote proxy to acquire knowledge of webpages and a local proxy that cooperates with the remote proxy to feed resources to browsers. The dual-proxy architecture requires no modification of existing Web browsers such as Chrome and Firefox, but can be seamlessly deployed with them. The users just need to configure their mobile-side browsers to use our local proxy. All the requests can be handled by the local proxy. The local proxy handles all the requests issued by browsers. The dual-proxy architecture also minimizes the system overhead by offloading much of the computation from the mobile devices to the remote servers. The local proxy performs only the basic and necessary computations while the remote proxy performs computation-intensive jobs. By carefully designing the communication between the two proxies, we can minimize the consumption of network traffic and reduce the workload of the local proxy.

Figure 8 shows the overview of the SWAROVsky system’s architecture. The users configure a list of webpages to be optimized by SWAROVsky. The remote proxy keeps revisiting the webpages and recording all the resource information during the loading procedure based on a browser rendering engine (1). We design a resource loading graph to capture all the information characterizing the loading of a webpage, including resource URLs, content, and loading dependencies. The Resource Loading Graph Generator is responsible for generating and updating the resource loading graph in the remote repository (2). The Resource Loading Graph Generator is the key component to determine how to synchronize resources between the two proxies and to identify whether the resources with different URLs have the same content.

When users open a webpage, all the resource requests are intercepted by the local resource proxy and handled by the Resource Matcher (3). If a request is the first one to retrieve the root HTML, the Resource Synchronizer starts to synchronize the resources from the remote resource proxy and fetch the resources to the local resource proxy (4). The synchronizer creates a profile of local resources related to the target webpage. The profile contains only the MD5 of the resources. Then the request to the root HTML and the created profile are sent to the remote proxy. The remote proxy obtains the corresponding resource loading graph, updates the resource status, and then indicates whether the resource has been changed by comparing the MD5 included in the profile against the existing profile in the remote repository (5). The graph is returned to the client first for synchronization. Only those changed resources are transferred back to the local repository in a stream-based, compression-enabled manner (6). The transmission order is determined based on the resource loading graph. We present the detailed design of resource synchronizer in Section ??.

After the resource loading graph is received by the local proxy, the Resource Matcher begins to respond to the resource requests. We do not require all the resources to be successfully downloaded before processing the requests. For each request, if the URL can be found exactly in the resource loading graph, the corresponding response is constructed to encapsulate the resource content, either from the local repository if not changed, or from synchronization if changed (7). If the URL cannot be found, the Resource Matcher tries to determine whether another resource in the graph has the same content with the requested one. If matched, the response is constructed according to the matched resource. Otherwise, the Resource Matcher forwards the network request to the original Web server (8). These unmatched resources are usually user-related or location-based resources that different users in different locations may request different resources. Such a bypass mechanism ensures that our system never breaks down the page semantics. When the response is ready, it is then sent back to the browser (9).

Note that SWAROVsky aims to optimize the loading process of every single resource consisting of a webpage. In practice, there are some dynamic webpages whose resources may update more frequently, as well as personalized webpages that contain many user-specific resources. Since a webpage consists of many resources, not all resources in dynamic webpages change every time, and personalized webpages still have resources that are not user-specific. Therefore, SWAROVsky still has potential benefits for these two kinds of webpages.
4.5. Experimental Evaluation Summary

Figure 9 demonstrates the distribution of the page load time and data traffic for cold start and warm load with/without our system. It is observed that in each situation (the subfigure), the metric of our system (blue line) is smaller than the metric without our system (red line). Such an observation indicates that in the real scenario, SWAROVsky can substantially reduce the page load time and data traffic. On average, the page load time is reduced by 28.9% for cold start, while dropping to 9% for warm load because most of the resources can be served from the cache in the original browser without SWAROVsky so that the original page load time is short without SWAROVsky. The data traffic is reduced by 17.1% and 66.4% for cold start and warm load, respectively. The data traffic for warm load is reduced substantially because most of the original traffic for warm load is redundant because the page is revisited immediately after the cold start.

More details can be referred in our work [6].

4.6. Challenges and Discussions

We discuss some challenging issues that our current design and implementation have to address in further practice. Security and Privacy. Given that SWAROVsky is deployed as a personal cloud service to optimize the web resource loading for end users, it could support HTTPS as a trusted middlebox, where the communications and transmissions are assumed to be secure. In such a case, the connection between the browser and the local proxy, and the connection between the remote proxy and the original server, can be built upon two separated HTTPS connections, and our system intermediates the communication and optimization. For the bypass mechanisms, the local proxy initializes a new HTTPS connection to the original Web server. In addition, the connection protocol between the local proxy and the remote proxy can be built atop SPDY/HTTP2. In practice, supporting HTTPS-based middlebox is still rather challenging and various issues need to be addressed, including certificate management of multi-entities, explicit control and visibility to endpoints, etc. [15], [16]. Hence, supporting HTTPS in SWAROVsky is out of the scope of this paper, and we plan to take into account in the future.

Page Semantics. One critical requirement for designing the proxy is to preserve the consistent semantics of webpages. The content, layout, and function of webpages must be consistent with the case where the page is directly served from the original Web servers. Our system introduces a resource matching mechanism to avoid redundant transfers. Resources with different URLs may be matched to the same one, resulting in the possibility of breaking page semantics. Experimental results show that 2.8% resources are mismatched with previously cached resources. After careful examination, we find that these resources are all small images, most of which are related to dynamically generated advertisements. In contrast, the major resources including HTML, JavaScript, CSS, and most static media objects can be consistently matched by our algorithm. Therefore, these mismatched resources do not break the correct layout and function, and indeed benefit end-users in reducing data traffic. However, developers may require these resources for special purpose, such as advertisement and statistics. To alleviate the impact of mismatched resources in page semantics, one solution is to identify and enforce the resources to be directly fetched from original Web servers.

Scalability. In our current design, we do not take into account the storage management of the local and remote proxies. Resources are never cleaned out of the storage unless they are explicitly updated. However, in practice, we should efficiently manage those resources. The classic LRU algorithm could correlate the resource request history with the resource update history. For those resources that are hardly requested or always updated, we can remove them from the storage. Our current design targets at improving the resource loading process of a single webpage and enabling users to configure which webpages to be optimized. In practice, we could take inter-page relationship to proactively generate or update resource loading graphs for subsequent pages. For example, the remote proxy can refresh resource loading graphs of all the pages referred by the hyperlinks in the currently visited webpage, ensuring users to retrieve the latest resources when moving to the next webpage. In addition, it is possible to learn the browsing behaviors of users to identify the webpages to be optimized. We leave such user-based optimization as future work.

Deployment Alternatives. Currently, the remote proxy is designed to deploy on a personal cloud to serve a specific user. There are two deployment alternatives to satisfy different scenarios of mobile Web browsing. On one hand, the remote proxy can be deployed by cellular network operators or smartphone venders that are always striving to improve user experiences on mobile devices [17]. In this scenario, the local resource proxy can be integrated into the smartphone to benefit all the users who have access to the proxy-deployed cellular network. On the other hand, the remote resource proxy can be deployed in the local WiFi network environment, leveraging the idea of cloudlet [18], [19]. For example, the remote proxy can be integrated into a wireless router. In this scenario, the remote proxy can benefit all the users who have access to the local wireless network.
5. Some Ongoing Efforts

The preceding sections introduce the related efforts focusing on only data cache management at application-level and runtime-level. Our goal is to provide a holistic approach to optimizing client-side resource management for mobile Web, and some ongoing efforts are worth mentioning.

Client-side computation resource management. Indeed, this paper focuses on only the imperfect client-side cache management. When we design the SWAROVSky system, we find that prefetching and rendering a web page in advance can help reduce the data traffic. However, when interacting with the Web applications, the local proxy at the client-side suffers from the ever-increasing complexity of new Web applications, especially recent emerging Web-based virtual reality (VR), augmented reality (AR), and other in-situ analytics AI applications, and requires a lot of local computation resources such as CPU, GPU, and memory. In this way, we aim to further optimize computation resource management. An intuitive solution is to offload the computation-intensive logic onto the remote proxy of SWAROVSky. However, due to the dynamics of JavaScript and local resource constraints, such a computation offloading raises more challenges compared to offloading Java-based native apps [20], [21]. We recently implemented the i-Jacob system [7] to dynamically refactor JavaScript code and have some preliminary results. However, some technical issues such as dealing with closure and arbitrary variables (such as eval, still require in-depth research effort.

Prioritizing personalized Web prefetching. SWAROVSky relies on the predictive power according to user behaviors. Although the Web access behaviors can vary among different users, it is argued that some access patterns are still predictive [19]. In our previous efforts that were conducted over millions of Android users [3], [22], we have found users holding specific device models have similarities in Web browsing behavior. Both ReWAP and SWAROVSky can easily obtain the device information (e.g., by User-Agent), the priority of to-be-accessed Web pages can be predicted, and enable more efficient prefetching onto the remote proxy.

High-Level programming language support. ReWAP provides developers facilities to refactor their Web applications with some manual efforts. Intuitively, it should be better that the developers can informatively define some constraints and contextual adaptation rules in applications. However, we notice that today’s Web programming languages mainly express structure, style, and functionality of an application. End-user concerned QoE information mentioned in this paper, however, is largely unaccounted for [23]. Without such information, the underlying Web runtime may make uninformed decisions. We recently design MUIT [24], a domain-specific language that let Web developers specify user QoE expectations as JavaScript annotations. The philosophy behind MUIT is that developers provide minimal yet vital QoE information to guide the underlying runtime optimizations.

Optimizing resource management based on machine learning techniques. Resource management of Web browsing actually comprises of complex tasks spanning from the operating system and networking levels to the browser and application levels. Various of factors at different levels interfere with each other to influence the user experience of Web browsing. Traditional efforts including ours follow a top-down approach where the issues are first identified and solutions are then proposed. The recent progress of machine learning techniques provides a new way of optimizing resource management. By collecting information of resource management at different levels from large scale of Web browsing traces, it is possible to extract features and construct models of learning optimal resource-management decisions, e.g., resource loading order, sub-resources, CPU configuration [25] etc. Based on the learnt model, we could track back to the root causes and improve the design.

6. Conclusion

Users spend most of their time on mobile Web, but complain the imperfect quality of experience in terms of page load, data traffic, energy, and so on. We have proposed a holistic approach that rethinks the current client-side resource management. We demystify the imperfect client-side data cache mechanism, and propose two orthogonal solutions at the application-level (ReWAP) and runtime-level (SWAROVSky), respectively. Collaborated with industry vendors including Kika [26] and Chelaile [27], the real-world deployment and evaluation of our approach at scale, are now in progress.
References


