SWAROVsky: Optimizing Resource Loading for Mobile Web Browsing

Xuanzhe Liu, *Member, IEEE,* Yun Ma, Xinyang Wang, Yunxin Liu *Senior Member, IEEE,* Tao Xie *Senior Member, IEEE,* and Gang Huang *Senior Member, IEEE*

Abstract—Imperfect Web resource loading prevents mobile Web browsing from providing satisfactory user experience. In this article, we design and implement the **SWAROVsky** system to address three main issues of current inefficient Web resource loading: (1) *on-demand and thus slow loading of sub-resources of webpages*; (2) *duplicated loading of resources with different URLs but the same content*; and (3) *redundant loading of the same resource due to improper cache configurations*. SWAROVsky employs a dual-proxy architecture that comprises a remote cloud-side proxy and a local proxy on mobile devices. The remote proxy proactively loads webpages from their original Web servers and maintains a resource loading graph for every single webpage. Based on the graph, the remote proxy is capable of deciding which resources are "really" needed for the webpage and their loading orders, and thus can synchronize these needed resources with the local proxy of a client efficiently and timely. The local proxy also runs an intelligent and light-weight algorithm to identify resources with different URLs but the same content, and thus can avoid duplicated downloading of the same content via network. Our system can be used with existing Web browsers and Web servers, and does not break the normal semantics of a webpage. Evaluations with 50 websites show that on average our system can reduce the page load time by **43.1%** and the network data transmission by **57.6%**, while imposing marginal system overhead.

Index Terms—Mobile Web, Resource Loading, Optimization.

1 Introduction

Web resource loading is a critical part of Web browsing. It is reported that 65% of page load time is spent on resource loading [1], and resource loading is the bottleneck of Web browsing [2]. Optimizing resource loading is particularly a key requirement on mobile devices. On one hand, mobile devices have limited computing capability, cellular data plan, and power supply of battery, and hence the resource loading should be traffic-saving and energy-efficient. On the other hand, the performance of mobile networks is significantly influenced by the characteristics of Web browsing traffic introduced by the resource loading [3].

Ideally, only the resources that are needed for rendering a webpage and not cached on the client should be fetched from the network. Although a lot of research efforts have been invested to optimize the loading of Web resources [4], [5], [6], our recent findings [7], [8] demonstrated that the current Web resource loading still suffers from the following three main issues of inefficiency.

• **On-demand but slow loading of sub-resources**. Modern webpages are complex and each webpage may consist of numerous Web resources, including HTML, CSS, JavaScript, images, etc. Those sub-resources are loaded only when they are needed. For example, an embedded image is fetched from the network only after the root HTML is parsed or a JavaScript function is executed. This on-demand process introduces multiple network Round-Trip Times (RTTs) and thus slows down resource loading.

- *Xuanzhe Liu, Yun Ma, Xinyang Wang, Gang Huang are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871. Email: {liuxuanzhe, mayun, wangxinyang, hg}@pku.edu.cn. Xuanzhe Liu and Gang Huang are also with Beida (Binghai) Information Research, Tianjin, 300450.*
- *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, USA. Email: taoxie@illinois.edu*

Manuscript received xx, 2016

• **Duplicated loading of resources with different URLs but the same content**. As shown in our previous study [7], it is common that different resources identified by different URLs could have the same content, due to the load balancing of Content Delivery Network (CDN) or intention of developers to force to re-load a resource. As a result, the same content is downloaded from the network for multiple times rather than being served from the local cache, because Web cache uses URLs to identify resources.

• **Redundant resource loading due to imperfect cache configurations**. Web cache is expiration-based where an expiration time for each Web resource is set by a Web server or a Web browser. If the current time is after the expiration time of a Web resource, a Web browser re-fetches the Web resource rather than loading it from the local cache. Our previous study [8] also shows that the expiration time of Web resources is rather conservative (i.e., too short). Consequently, many unchanged resources are wrongly expired in the local cache, leading to unnecessary network transmissions.

In this article, we aim to optimize Web resource loading by mitigating these three main issues. In particular, we design and implement a dual-proxy system **SWAROVsky** (**S**mart **W**eb **A**cceleration by **R**esource **O**ptimization o**V**er the **sky**). SWAROVsky consists of a remote proxy deployed on the cloud and a local proxy deployed locally on mobile devices. Different from traditional proxies that mainly perform simple request forwarding and cache lookup, our solution is proactive and intelligent. The remote proxy intelligently crawls and renders webpages from original Web servers. It stores all the downloaded resources in loading each webpage and builds a Resource Loading Graph (RLG) for the webpage. The resource loading graph of a webpage determines which resources are needed to load the webpage and in which order those resources should be loaded. As a result, when a client requests a webpage (via the local proxy of the client), the remote proxy is able to immediately send all the pre-fetched resources of the webpage to the client in

a batch and in the right order.

The local proxy on the mobile device records all the resources that the client has downloaded. When the client requests the same webpage for the next time, the local proxy tells the remote proxy what resources the client already has and thus the remote proxy sends only the missing resources to the client. The local proxy also runs an intelligent and lightweight algorithm to identify the resources that have different URLs but the same content, so that those resources are not downloaded for multiple times. The remote proxy periodically crawls data from Web servers to ensure that the client can always get the latest data, and unchanged resources are not retransmitted to the client even if their expiration time is not properly configured.

With the preceding techniques, SWAROVsky is able to minimize the data transmission in loading a webpage, and thus improves the page load time as well as optimizing the user experience. More specifically, this article makes the following main contributions:

- We present how to build a resource loading graph to capture the characteristics of the resource-loading process and derive the priority of resource transmission in rendering a webpage.
- We develop a resource synchronizer for a local proxy and its remote proxy to achieve accurate and timely resource downloading, so that those unnecessary redundant transfers can be eliminated.
- We design an efficient algorithm of resource matching to determine whether a resource identified via a new URL has the same content with a previously cached resource. Therefore, the same content is never repetitively downloaded.
- We implement the dual-proxy prototype system that requires no changes to existing Web browsers or Web servers. The evaluation results demonstrate that our system can significantly improve Web resource loading, i.e., reducing the average page load time by 43.1% and the average network data transmission by 57.6%, while imposing small system overhead.

Indeed, the idea of dual-proxy design in general is not entirely new. However, as studied in our previous work [7], existing systems have addressed only parts of the imperfect resource loading of mobile Web browsing. Our system comprehensively optimizes the resource loading process, and is unique compared to existing systems in terms of how to efficiently design the two proxies with the resource loading graph, resource synchronizer, and resource matching algorithm. More details on related work can be found in Section 11.

In the rest of this article, we take a motivating example to illustrate the three main issues of Web resource loading in Section 2. We define the design goals and describe the challenges in Section 3. We present the architecture overview of our system in Section 4. Then we describe the details of resource loading graph, resource synchronization, and resource matching in Sections 5-7, respectively. We present the implementation in Section 8, and the system evaluation in Section 9, and discuss issues in Section 10. After comparing with related work in Section 11, we draw conclusions in Section 12.

2 Motivating Example

We begin with a motivating example to illustrate the imperfect resource loading of current webpage rendering process. Figure 1(a) shows the root HTML of an example webpage identified via the URL of "http://foo.com/index.html". To load the webpage in a browser, the HTML document is first fetched from the network. The browser then parses the HTML document. During parsing, referenced resources are identified and fetched, such as "`a.css`" and "`b.js`". After the JavaScript code in the <script> tag is executed, 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). After all the resources are loaded, users can view the whole page on the screen.

Figure 1(b) shows the cache configuration on the Web server for each resource of this webpage. The HTML resource "`index.html`" is not assigned an explicit expiration time so that the browser assigns a random expiration time to the HTML resource, usually less than one hour [8]. The CSS resource "`a.css`" and the JavaScript resource "`b.js`" are assigned an explicit expiration time of 86,400 seconds (one day) and 300 seconds, respectively. The image is assigned with an expiration time of one year. Suppose that a user revisits the webpage after one hour, and none of the resources has been updated yet since last time's visit. In such case, due to the expiration of HTML and JavaScript, the two resources have to be downloaded again from the network, while the CSS can be directly loaded from the local cache. For the image, when the JavaScript code executes 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.

Figure 1(c) shows the resource loading sequence of the two visits. 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_2 and t_3, respectively. During the loading of JavaScript, the HTML parsing pauses temporarily. Finally, the image starts to be loaded at t_4 and finishes at t_5. For the second visit, since HTML, JavaScript, and image 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 all the three main issues of inefficiency described in Section 1. "`a.css`" and "`b.js`" are loaded after "`index.html`" is parsed, 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.

As modern webpages become increasingly complex and contain more and more sub-resources, the on-demand sub-resource loading leads to a significantly long page load time. Attaching a random number to a resource is a common practice for load balancing or forcing to re-fetch the resource. Based on our previous measurement study [7], [8], more than 20% of redundant resource transfers are caused by resources with different URLs but the same content.

Ideally, the loading of the example webpage should be like Figure 1(d). 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.

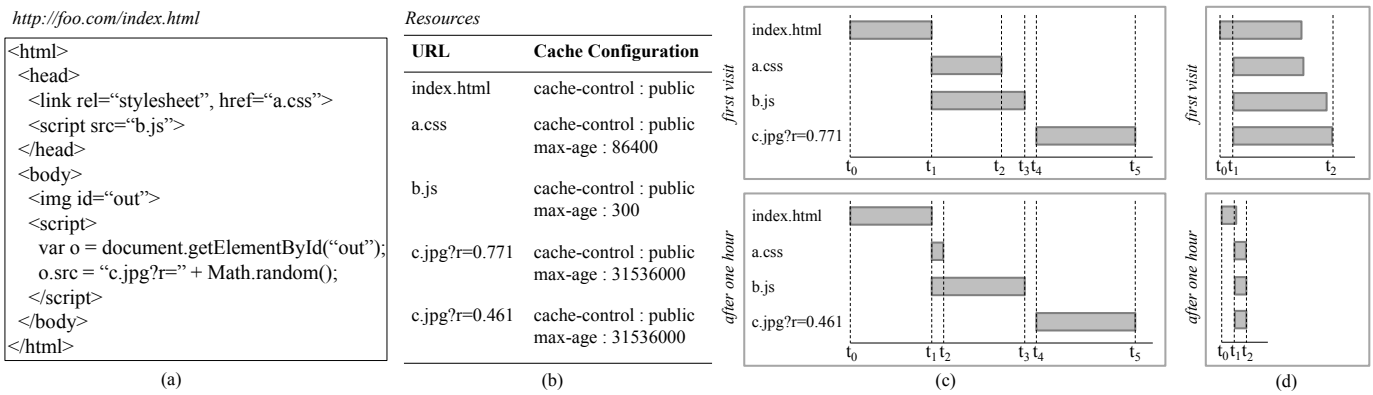


Fig. 1. Motivating example. (a) Root HTML of a webpage. (b) Cache configuration of every single resource. (c) Resource loading sequence of the current process. (d) Potential improvement of the process.

3 Design Goals and Challenges

Based on the findings of motivating example, we then define our system design goals and describe the requirements and challenges in addressing the three main issues of inefficiency.

3.1 Design Goals

To optimize the resource loading for mobile Web browsing, the following design goals should be realized.

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.

3.2 Challenges

It is challenging to address the main issues of inefficiency in loading Web resources, as mentioned in Section 1. To address the first issue (on-demand but slow loading of sub-resources), our system 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.

To address the second issue (duplicated loading of resources with different URLs but the same content), 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.

To address the third issue (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 System Overview

To achieve our design goals and tackle the challenges, the key idea of our system design is to leverage the 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 2 shows the overview of our 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. We describe the details of resource loading graph in Section 5.

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 6.

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

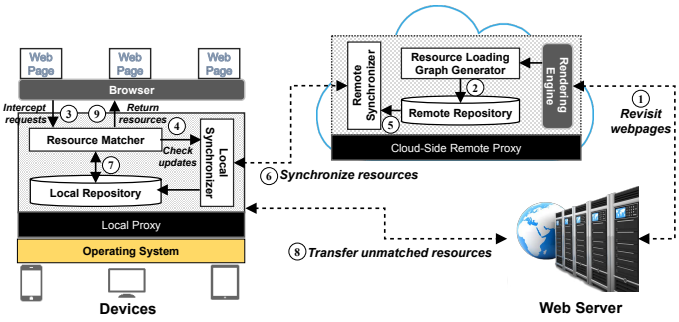


Fig. 2. Architecture overview of the SWAROVsky System.

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). We show the design of resource matching algorithm in Section 7.

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.

5 Resource Loading Graph

The resources to be loaded along with their loading order are crucial to improve the resource loading process. Given the resources to be loaded, our system determines whether to transfer the complete resource content or not, according to their status of update. Only the changed resources need to be downloaded. Given the loading order of resources, our system first synchronizes those resources that are likely to be loaded earlier than others. Since modern webpages become more and more complex and dynamic (e.g., using a lot of JavaScript code), their resource loading cannot be statically determined in an offline manner. We design a model of resource loading graph to capture resource loading information. The graph is generated dynamically by actually rendering webpages on the remote proxy.

5.1 Resource Loading Graph

The loading of a webpage essentially consists of the loading of every single resource. Except for the root HTML, every resource has one parent resource that has to be loaded beforehand. Hence, there are dependencies between resources. One straightforward technique is to structure the dependency according to the "`referrer`" field in the HTTP request message. This technique could be inaccurate because the resources loaded by JavaScript refer to the HTML other than the JavaScript. To obtain the accurate dependencies of resources, we build a resource loading graph based on the page's actual loading sequence.

After rendering a webpage, we can get a resource loading sequence where each resource has a start-loading time and an end-loading time. We define a directed acyclic graph (DAG) $G =< V, E >$, where V refers to the set of resources

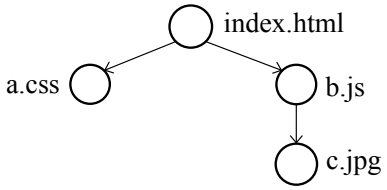


Fig. 3. Resource loading graph of the motivating example in Section 2.

and E refers to the dependencies. Each edge $< v_1, v_2 >$ is defined as follows: v_1's end-loading time must be earlier than v_2's start-loading time, and the difference between the two time stamps should be the minimum among all the resources. The root HTML has no parent vertex, and the HTML is the parent vertex for all those resources that cannot find a parent vertex based on the preceding rule.

Figure 3 shows the resource loading graph of the example described in Section 2. The `index.html` is the root vertex, and `a.css` and `b.js` are the two children vertexes of `index.html`. The image `c.jpg` is the children vertex of `b.js`.

5.2 Graph Generation

The resource loading graph can be dynamically established and incrementally maintained according to the order of loading sequence. When the request to a resource is started, the resource can be placed into the graph. So, it is possible to generate a partial graph in the progress of page loading. Such a partial graph is also generated and used when accessing a webpage on the mobile devices. The pseudo-code of dynamic graph generation is described in Algorithm 5.1.

Input: A set of resources $R = \{r_1, r_2, \cdots, r_n\}$
Output: Resource loading graph $G < V, E >$

```
1  INITIAL V as the resource of root HTML r_html
2  INITIAL E ← φ
3  foreach r_i ∈ R \ r_html do
4      span_min ← ∞
5      refer ← r_html
6      foreach v ∈ V do
7          span ← v.end − r_i.start
8          if span > 0 and span < span_min then
9              span_min ← span
10             referer ← v
11         end
12     end
13     add r_i into V
14     add < referer, r_i > into E
15 end
16 RETURN G < V, E >
```

Algorithm 5.1: Construction of Resource Loading Graph.

Initially, there is only one vertex in resource loading graph, i.e., the root HTML. For each subsequently requested resource, we traverse all the resources in the graph to find the resource whose end-loading time is the closest to the current resource's start-loading time. Then the parent vertex is set as the found resource. If no vertex is found, the parent vertex is set as the root HTML.

6 Resource Synchronization

When a webpage is being visited or revisited, the new or updated resources have to be downloaded from the server.

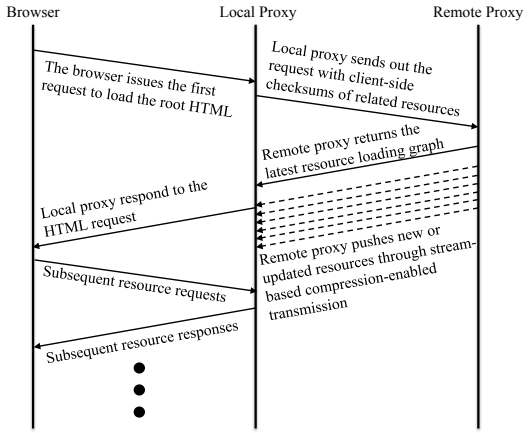


Fig. 4. High-level procedure of resource synchronization between the local and remote proxies.

According to the resource loading graph defined in the previous section, we can know all the resources to be loaded as well as their loading order. However, in practice, there should be an efficient way to download resources to the client side timely and accurately. In this section, we present the design of resource synchronization to achieve two goals: (1) sub-resources (i.e., resources except the root HTML) should be ready in the local environment before they are being requested; (2) the network traffic consumed in the synchronization should be minimized.

6.1 High-Level Procedure

Figure 4 illustrates the high-level procedure of resource synchronization between the local proxy and remote proxy. The resource synchronization starts when the browser issues the first request to load the root HTML of the target webpage. If the webpage has been previously visited, then the related resources have been downloaded in the previous visit. In this case, the local proxy sends the requests to the remote proxy with the checksums of all related resources in the local repository as well as the reference URL of the target webpage. If the webpage is visited for the first time, then only the reference URL is sent to the remote proxy.

When receiving the synchronization request, the remote proxy retrieves the corresponding resource loading graph. The remote proxy also compares the checksum of resources in the remote repository with those received from the client, and obtains the update status of each resource. Then the remote proxy returns the resource loading graph together with the resource's update status to the local proxy. Finally, all the resources whose checksum cannot be found in the local repository are pushed back to the local proxy according to the transmission priority computed based on the resource loading graph. In this way, we ensure that the unique resource content is never repetitively downloaded.

When the local proxy receives the resource loading graph, it responds to the browser requests. If a resource is not updated, it is directly returned to the browser. If a resource is entirely new or updated, it is returned to the browser after the resource has been successfully downloaded from the remote proxy. Our transmission priority aims to make resources ready on the client when they are being requested. In other cases, if a resource cannot be found in the resource loading graph, the resource matcher handles

this situation. We describe the resource matcher in the next section.

6.2 Priority Determination

For each path of the resource loading graph, a deeper resource vertex has a later loading time in the loading sequence. In other words, the loading priority of a deeper resource vertex should be lower. Therefore, we can choose any topological order of the resource loading graph as the transmission priority, and such order satisfies all dependencies.

However, the choice of topological order should be considered carefully. Since we use the stream-based transmission, we should aim to transmit a few resources at the same time to get the maximum utilization of the network bandwidth. Therefore, some topological orders could reduce the performance of page loading. With simple topological sequences, some "leaf node" (the vertex has zero out-degree) may have higher priority than the "middle node" (the vertex has one or more out-degree). That is, some crucial resources, which can trigger to load some other resources, may be blocked by common resources.

To keep the crucial resources that are always loaded before common resources, we design a customized topological sorting algorithm to compute a reasonable resource loading sequence. For a resource loading graph $G =< V, E >$, we build a reverse resource graph $G^* =< V^*, E^* >$, where $V* = V$, $E^* = \{< v_1, v_2 > \mid < v_2, v_1 >\in E\}$, i.e., all the edges in graph G are reversed in G^*. As the resource loading graph G is a directed acyclic graph (DAG), it is easy to prove that G^* is also a DAG. Therefore, we can obtain the topological order of G^*. With this order, we can keep that all the common resources are placed before the crucial resources. In addition, the order also satisfies the dependency order where the referred resources are placed before the parent resources. Such an order is a reversed transmission sequence, and the former resources should have a lower priority than those latter ones.

6.3 Resource Transmission

To make a resource's transmission faster with less consumption of data traffic, our system includes three optimizations, i.e., stream transmission, checksum-based redundancy removal, and data compression.

6.3.1 Stream transmission

We adopt the stream transmission technique, which can both reduce the number of TCP connections and deliver the resources to browsers as soon as possible. In fact, recently emerging protocols such as SPDY [9] and HTTP/2 [10] support the reuse of TCP connection and stream-based transmission. Both of the protocols also support priority for each stream. It is straightforward to adopt the mechanism provided by these latest protocols.

6.3.2 Checksum-based redundancy removal

To ensure that the cached resources are never redundantly downloaded if they are not changed, we adopt a technique for removing checksum-based redundancy. We calculate the checksum (MD5) of each resource fetched on the remote proxy. When the synchronization begins, the client proxy first sends checksums of all the related resources to the remote proxy. When transferring resources back to the client, the remote proxy checks whether the checksum has already been included in the checksums sent from the client. The synchronizer transfers only the resources whose checksum is missing.

6.3.3 Data compression

Regardless of the data traffic, some webpages could provide uncompressed data to users. Additionally, the HTTP headers are always uncompressed due to the limitation of the HTTP/1.1 protocol. We always keep all the data compressed in transmission, including our resource loading graph, the HTTP header, and the content. Compression can have some improvements in data traffic.

7 Resource Matching

After the graph is synchronized, the root HTML is returned to the browser at the mobile client side. Then, the subsequent requests are emitted by the browser to load resources. Those resources whose URL can be found in the resource loading graph are fetched either from the local cache for unchanged resources, or from the local proxy for new/changed resources.

However, as mentioned previously, between different visits of the same webpage, the URLs of some resources could change but their contents are essentially the same. Therefore, we need to determine whether a resource identified by a missing URL in the resource loading graph has the same content with other resources. With this knowledge, we can eliminate the unnecessary network transfers to repetitively download the same resource. To this end, we design a resource matching algorithm that relies on the resource loading graph and URL similarity.

7.1 Graph-based Positioning

When the local proxy responds to resource requests sent from the browser, we can generate a partial resource loading graph of the currently loading webpage on the browser. Since most of the resources are synchronized from the remote proxy, the client-side resource loading graph is likely to be similar to the one acquired from the remote proxy. We assume that the loading order of the same resource does not significantly change at different loads. Therefore, the same resource should have a similar position in the resource loading graph.

Since the resource loading graph may change, we cannot simply use its parent or child vertex that can be unstable at different visits. Instead, we choose to use the sibling vertex, as a resource's siblings are likely the resources that are loaded simultaneously. The sibling vertexes of vertex v_t in the resource loading graph $G =< V, E >$ can be expressed as $S(G, v_t) = \{v_s \mid < v_r, v_s >, < v_r, v_t >\in E$ and $v_s \neq v_t$ and $v_r, v_s, v_t \in V\}$. Finding the siblings in the graph is realizable. Recall the generation of a resource loading graph: when the local proxy receives a resource's request from the browser, a vertex can be added into the resource loading graph. Therefore, the graph-based positioning can also be dynamically performed.

Next, we quantify the positioning. Assume the graph synchronized from the proxy as $G_1 =< V_1, E_1 >$, and the graph generated during the page loading on the client as $G_2 =< V_2, E_2 >$. The existing resource vertex is $v_e(v_e \in V_1)$. And the target resource vertex is $v_t(v_t \in V2)$. Therefore, the union of $S(G_1, v_e)$ and $S(G_2, v_t)$ is the potentially matched vertexes between the two resources. We quantify the matching level as below:

$$M = \frac{|S(G_1, v_e) \cap S(G_2, v_t)|}{2} \cdot \left(\frac{1}{|S(G_1, v_e)|} + \frac{1}{|S(G_2, v_t)|}\right)$$

The formula shows that the matching level always ranges from 0 to 1. When a resource has a zero matching level, we do not consider it as a potential matched resource.

TABLE 1
An example of splitting a URL into different parts.

URL:	api.dos.	aliexpress.com/	aliexpress/data/	doQuery.json?	widgetId=101&	locale=en_US&	limit=30
Segment:	Prefix	Host	Path	File	Arg1	Arg2	Arg3

7.2 URL Similarity

Then, we consider the similarity of URLs. We assume that resources with the same content usually have similar URLs. For each URL, we split it with a common separator (such as "/", "?", and "=") and get the words consisting of this URL. We match the words between two URLs. The more words can be matched, the higher chance the two resources can be the same.

Furthermore, we should distinguish different parts of the URL because different parts have apparently different importance levels for our resource matching algorithm. We divide a URL into five parts: prefix, host, path, file, and arguments. Table 1 shows an example of how a concrete URL is split into segments.

It is important to set the crucial coefficient for each segment. However, we cannot determine the coefficient statically because different resources may have different URL parts to determine the content. Some are determined by the arguments, e.g., id=1 and id=2 indicate two different resources. However, for some URLs parsed by JavaScript, the argument includes only the timestamp, which is little semantically meaningful for the content.

After repetitively revisiting the same webpage, we can collect a set of URLs that correspond to the same resource. First, we divide all the URLs with the preceding technique. Then we count the maximum number of the same string that appears in a segment. We count the number for every URL and segment. Then, the crucial coefficient of a segment in a URL can be defined as

$$\alpha_{seg} = \frac{n_{moststring}}{n_{URL} + 1}.$$

Here, $n_{moststring}$ represents the number of the most common string that appears in this segment, and n_{URL} is the number of URLs that point to the same resource. The coefficient should be always between 0 and 1. So we can get the crucial coefficient for each segment. For those segments that are stable for the same URL, the coefficient is higher. Otherwise, it is lower. Note that the crucial coefficient calculated on the proxy should be sent to the mobile client with the graph.

Finally, we can compute the similarity of URLs. For a matched URL, if the requesting URL has the same segment with it, the segment's matching degree is set as 1. Otherwise, the matching degree is $1 - \alpha$ (α is the segment's crucial coefficient). Then we multiply all the matching degrees together to get the final similarity, which can be represented as

$$Similarity = \prod_{i=1}^{segmentsinURL} (1 - x_i \cdot \alpha_i)$$

where α_i is the crucial coefficient for segment i, and x_i represents whether the segment is different, i.e., 0 denotes the same one and 1 denotes a different one.

7.3 Matching Algorithm

For each resource to be matched, we can calculate the positioning matching level and URL similarity with each existing resource in the resource loading graph. Both scores range from 0 to 1. We multiply these two scores and get the final resource matching score of each existing resource. We regard the resource with the highest score as the most probable same resource.

It is important to set a threshold to determine whether the target resource is the same to an existing resource. We conduct experiments to find the appropriate threshold, which we discuss in Section 9. We finally set the threshold as 0.52. When the score is higher than 0.52, we regard that the target resource is the same as an existing one. If it is lower than 0.52, the target resource is a brand-new resource, and the local proxy should request the resource from the original Web server.

7.4 Bypass Mechanism

Resource requests that cannot be matched to a resource in the resource loading graph are directly forwarded to the original Web server. However, developers may intentionally enforce some resources to have very short cache time or to be requested at every visit, e.g., collecting statistics of clients. Without breaking such semantics, even if the resources can be matched in the resource loading graph, they should still be requested from the original server. To this end, we add some bypass mechanisms. Resources whose cache policy is "no-cache" and those who have empty content are enabled by bypass mechanisms, since in practice these resources are mainly used to upload some specific-purpose data to servers.

8 Implementation and Deployment

According to the design presented in the preceding sections, we implement the SWAROVsky system and deploy it onto the current Web architecture. This section describes the details.

The local proxy is implemented as a standard Android system-wide service that listens to a specific network port. The Resource Synchronizer is implemented upon SPDY protocol to leverage its stream-based priority-enabled transmission mechanism. We use the SPDY library provided by Jetty [11] to implement the transportation layer. We do not use the recently released HTTP/2 protocol due to the lack of a stable library on Android, but it is easy to replace SPDY with HTTP/2 in our implementation in the future. The remote resource proxy is implemented as a Jetty container to enable SPDY. The Resource Loading Graph Generator on the remote proxy is implemented based on the Chromium Embedded Framework (CEF) [12] and Chrome remote debugging protocol [13]. CEF is a browser facility that has the same kernel as the latest Chromium browser but exposes APIs to control the browser. We use CEF to actually load webpages on the remote proxy and at the same time obtain the resource loading sequences via the Chrome remote debugging protocol. To keep up with the resource update, we revisit webpages every 30 minutes and update the related resource loading graph.

Since the remote proxy is designed to serve per user, it can be deployed as a stand-alone service on a personal cloud, which is emerging as a popular cloud computing

paradigm [14], [15]. In particular, according to a previous study on browsing behaviors [16], an individual user is quite likely to revisit on a fixed range of webpages. Hence, it is feasible to employ such deployment fashion in SWAROVsky. Indeed, there can exist other deployment alternatives and we discuss them further in Section 10.

9 Evaluations

We evaluate SWAROVsky from four aspects. First, we investigate the improvement of resource loading accomplished by our system compared to the original resource loading process. Second, we evaluate the performance of the key components in our system. Third, we examine the overhead of our system brought by the dual proxy. Finally, we collect user browsing data to evaluate how our system can benefit real users.

9.1 Evaluation Setup

All our experiments are performed on a Samsung N7100 smartphone (with CPU 1.6GHz and 2GB RAM), running the Android 4.4 operating system. We use the latest Firefox browser as the front-end because in Firefox: (1) configuring the network proxy is rather easy, and (2) leveraging the add-ons can help perform automatic experimentation. We install the local resource proxy on the smartphone and configure Firefox to use the proxy. We also develop a Firefox add-on to enable automatic opening of webpages given specific URLs.

We deploy the remote resource proxy on a laptop computer, Thinkpad T420i with the Windows 7 OS. We connect the smartphone and the laptop computer under the same WiFi environment. We also hardwire the smartphone with the laptop computer by USB to enable monitoring the CPU and memory usage of the smartphone through ADB.

The dataset. We choose the webpages to be evaluated from the websites of Alexa top 500 ranking list [17]. We exclude websites based on the following rules. (1) For the websites that are banned in mainland China and cannot be accessed, such as Google, Facebook, and YouTube, we directly filter them out. (2) For the websites that have multiple domains, such as `amazon.com`, `amazon.co.jp`, and `amazon.co.uk`, we choose only the main domain, e.g., `amazon.com`. (3) For the websites transferred over HTTPS, we do not take them into account, because our current implementation has not realized the support of HTTPS. Finally, we have 190 websites remaining. As mentioned later, we revisit webpages every 30 minutes for evaluating the performance with various time intervals. In order to reduce the record workload, we choose 50 websites[1], covering not only feature-rich websites with numerous media resources but also simple ones with mostly textual contents. We evaluate the landing page of each website because it is the entrance and is usually revisited frequently by users.

9.2 Improvement of Resource Loading

We employ two metrics to measure the performance of resource loading. One is the page load time, which is the critical factor influencing the mobile-Web-browsing experiences. We use the time interval elapsing from the start-loading request to the triggering of the `onload()` event, which is widely used as the quantifier of page load time [18]. The other metric is the transferred data traffic that occurs at the edge of mobile devices; such a metric is a key factor especially when the devices are used under cellular networks. We examine how our system can improve the two metrics in the situations of cold start and warm load. Cold start is the case where a webpage is visited for the first time. Warm load is the case where the webpage is revisited after some time.

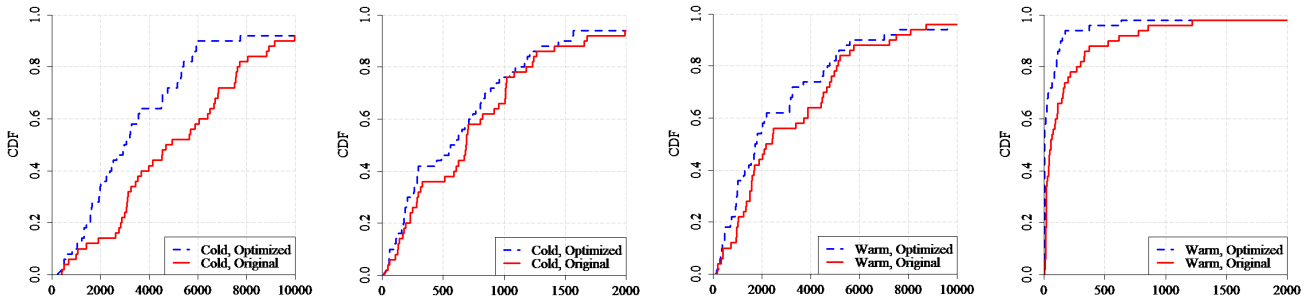
Improvement in a real scenario. For a real scenario, we enable the smartphones to directly connect to the Internet through high-bandwidth Wi-Fi. We load each webpage for 10 turns by enabling and disabling our system, respectively. Each turn consists of two loads. The second load is performed immediately when the first load finishes. We regard the first load as the cold start case and the second load as the warm load case.

Figure 5 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.

Improvement in a simulation test. One typical application scenario of our system is to improve browsing experiences when a webpage is revisited after some time. However, it is practically difficult to make this evaluation because we cannot wait for very long time without changing the status of the smartphone. Therefore, we adopt a record-and-replay approach to conducting simulation-based evaluation. We use our data collection tool [7] to record a three-week snapshots of the 50 selected webpages by revisiting them every 30 minutes. The first visit time represents the cold-start case. Each of the revisitings represents a warm-load case where the webpage is revisited after some time. We deploy a server to store these traces under the same WiFi of the smartphone and laptop computer, and configure resource loading from the server instead of the original servers. Based on a previous study on cellular network [19], we add 150ms latency for the replay server to simulate the long distance between the remote server and the mobile device.

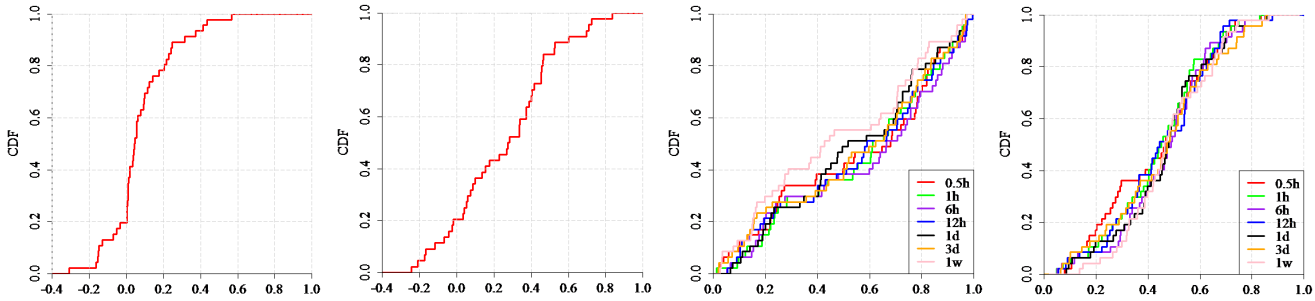
Figure 6 shows the improvement of page load time and data traffic for cold start and warm load. For the cold start, compared to the traditional loading, we can observe that both metrics can be reduced to some extent for about **80%** of the websites. The median improvements of page load time and data usage are **26.9%** and **4%**, respectively. Page load time can be reduced substantially because we save the connections to the original servers by just communicating with our proxy server. The reduction of data traffic is relatively small since all the resources have to be downloaded to the client. In some cases, there is a slight increase of page load time and data usage because SWAROVsky needs to transfer some additional information.

1. Please refer to http://www.mobisaas.org/projects/swarovsky for a complete list of the chosen websites as well as the evaluation results. We also provide a demo video to demonstrate our system.



(a) Page load time (ms) (b) Data traffic (KB) (c) Page load time (ms) (d) Data traffic (KB)

Fig. 5. Distribution of page load time and data traffic for cold start and warm load with or without our system in the real scenario.



(a) Reduction of data traffic (b) Reduction of page load time (c) Reduction of data traffic (d) Reduction of page load time

Fig. 6. Distribution of improvement of page load time and data traffic for cold start and warm load with our system.

For the warm load, we simulate the revisit interval ranging from 0.5 hour to 1 week to figure out the improvement. Figures 6(c) and 6(d) show the improvement of data traffic and page load time for warm load, respectively. We can observe that on average the page load time can be reduced by **43.1%** and data traffic can be reduced by **57.6%**. The reduction varies among different websites and different revisiting intervals. The improvement is more substantial for shorter revisiting intervals than for longer ones.

9.3 Evaluation of Each Component

Next we evaluate the performance of key components in our system.

9.3.1 Graph Maintenance

The graph generator on the remote proxy repetitively revisits webpages in order to capture the latest resource status of the webpage. The shorter the revisiting interval is, the more accurately the status of resources could be predicted. However, shorter revisiting interval increases the workload of the remote proxy, while longer interval may lead to visiting out-of-date information. We study the influences of revisit frequency on user experience. We choose five webpages and revisit them every 30 seconds, recording all the resources in each visit. We calculate the coverage ratio of matched resources under different intervals. The larger the coverage ratio is, the more latest information users can retrieve. Figure 7(a) shows the evolving of coverage ratio within 30 minutes. We match the resources both by its URL (solid lines) as well as by its actual content (dotted lines). The coverage ratio slightly decreases when the interval becomes longer. But the decrease is rather smooth. In the worst case of 30-minute interval, the coverage rate is still about **96%**. The ratio evolves differently for different webpages. If a webpage contains a lot of dynamic contents, the coverage ratio decreases accordingly. The differences between the coverage by URL and content implies the problem of URL-based resource identification. For example, the content coverage changes more smoothly than the URL coverage for Douban, Outbrain, and Yandex, indicating that some resources of these three webpages have URL change but no content updates.

9.3.2 Resource Synchronization

We then disable the stream-based priority-enabled resource transmission between the local proxy and remote proxy, and repeat the simulation test. All the resources are synchronized to the client only when the local proxy explicitly sends the request. We find that there is not obvious influence to the improvement of data usage. However, the page load time increases 39% on average compared to the current resource loading without our system. For the warm load, compared to our full-function system, the improvement of data usage does not vary much and the average improvement of page load time is reduced by **10.2%**. Therefore, we conclude that the stream-based and priority-enabled resource transmission is essential to the improvement of resource loading.

9.3.3 Resource Matching

To evaluate the performance of resource matching, we use the same data set with that described in Section 9.3.1: five websites are repetitively visited every 30 seconds for 100 visits. We use the records of 90 visits to train our model and calculate the parameter values. Then we use records of the last 10 visits as testing data to evaluate the performance. For each resource whose URL is missing in the repository, we calculate the matching score with all the other resources, and choose the one with the highest score as the matched

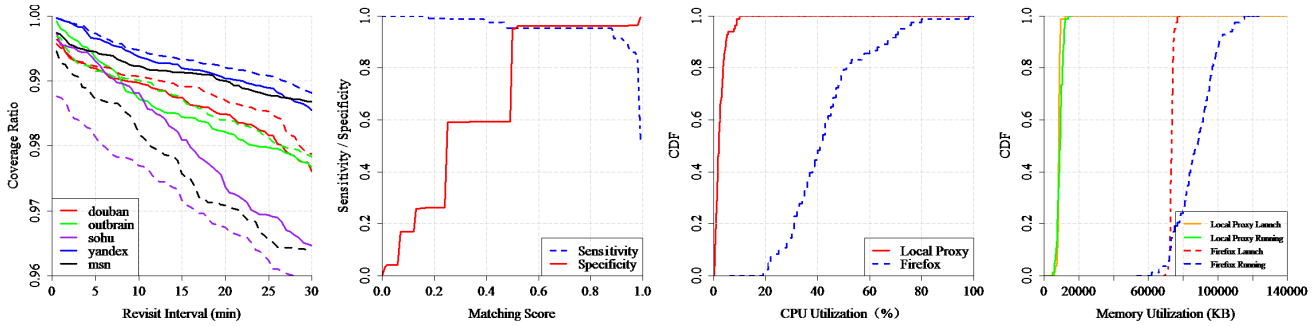


Fig. 7. Evaluation results. (a) Resource coverage when webpages are revisited every 30 seconds. (b) Specificity/Sensitivity Cut-off curve of resource matching. (c) Distribution of CPU utilization of local proxy and Firefox. (d) Distribution of memory utilization of local proxy and Firefox.

resource. If the contents of two resources are the same, the matching is correct. Otherwise, the matching fails. We draw the Specificity/Sensitivity Cut-off curve by choosing the threshold from 0 to 1 in Figure 7(b). The optimized cut-off value is empirically set as 0.52, where the matching precision can reach up to **95.8%**.

There are two kinds of the remaining **4.2%** mismatched resources: one is the "*false-accept*" resources where the two matched resources do not have the same content; the other is "*false-reject*" resources where the resources to be matched actually have the same content with a certain resource but our algorithm fails to identify them. More specifically, the "*false-reject*" resources account for about **2%** of all resources, while the "*false-accept*" resources account for **2.2%**. In practice, the "*false-reject*" resources have no side effect on the correctness of Web browsing, as the browser needs only to download them again from the server. In contrast, we need to carefully deal with the "*false-accept*" resources, as they can possibly lead to returning wrong resources and break the page semantics. However, by manually justifying these resources, we find that all of the "*false-accept*" resources originate from small GIF images related to advertisements. Therefore, although these "*false-accept*" resources may possibly affect the clicks of ads, our resource matching algorithm is able to promise the normal page semantics of resources including HTML, JavaScript, CSS, and most static media objects. Additionally, as we later discuss in Section 10, these "*false-accept*" resources can be avoided while preserving the effects of our approach.

Indeed, the cut-off value can have impact on the final precision of all matched resources as well as the "*false-reject*" and "*false-accept*". Our current empirically optimized value is set as 0.52 after several rounds of experiments over our current data set. In practice, such a value can be better tuned based on the actual user visits of specific webpages, to reduce the side-effect caused by "*false-accept*".

9.4 System Overhead

To investigate the overhead of our system, we study the CPU and memory usage of the local proxy and the remote proxy, respectively. The data is gathered when we perform the simulation tests.

Local proxy. Figures 7(c) and 7(d) show the distribution of CPU and memory utilization of local proxy and Firefox, respectively. For CPU overhead, the average utilization of the local proxy is just about 2.7% , which is only 1/10 of the utilization of Firefox. During the experiment, the CPU utilization of local proxy does not change as significantly as Firefox. The memory used by enabling the local proxy is also about 1/10 of the Firefox. The memory usage does not increase so much after loading some pages compared with Firefox. These results can demonstrate that our local proxy is lightweight without influencing the experience of browsers.

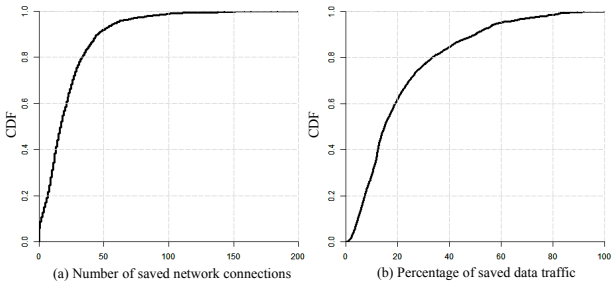


Fig. 8. Results of evaluation based on real users.

Remote proxy. The memory usage of resource synchronizer is about 37 MB when initially started. While synchronizing resources with the local proxy, the memory usage is around 70 MB, which is about twice of the cold start. The Resource Loading Graph Generator has more memory usage. When initially started, it has 100 MB memory. When repetitively loading 100 webpages one by one, the memory usage raises about 30%. The CPU utilization is about 10%. On the whole, the overhead of the remote proxy is considerable.

Network overhead. The resource synchronization requires transferring the information of resource loading graph from the remote proxy to the local proxy, incurring network overhead. We then measure the size of resource loading graph. For the 50 webpages that we use for simulation test, the average size is about 3KB and the maximum is below 8KB. So the overhead does not significantly impact the performance of our system.

9.5 Evaluations with User Study

We conduct a user study to evaluate how SWAROVsky performs in real scenarios. For SWAROVsky, we design a customized WebView-based browser, which can record the resource-loading information while browsing webpages. For each resource, we record all the headers from HTTP request and response messages, as well as the MD5 of the response body. The recorded logs are daily uploaded to a server when the smartphone is connected to WiFi. Then we

invite 10 college-student volunteers from Peking University to install the customized browser for Web browsing instead of using other browsers. Our customized browser does not introduce any learning curve or additional cost to users. To protect user privacy, the user identities (i.e., the serial number of devices) are anonymized by a hashtag. All the volunteers confirm the data-collection purpose and statements. The data collection lasts for one week and we finally get 1,883 records of webpage browsing.

We then compute how much data traffic can be saved when the users set up SWAROVsky on their mobile devices. For each record, we simulate the process of SWAROVsky and calculate the total data traffic and number of network connections, including the connection between the local proxy and remote proxy, as well as the requests forwarded to the original servers. Then we compare the recorded data of real usage with the simulated data by SWAROVsky, and compute the saved network connections and the saved data traffic.

Figure 8(a) shows the distribution of the number of saved network connections for all users' visits to all webpages. The median is 20, indicating that 20 network connections can be saved. About 5% of visits to webpages have no connection savings because some webpages have very small number of resources and many of them are bypassed from SWAROVsky. In best cases, about 1% of visits to webpages save more than 100 connections. Figure 8(b) depicts the distribution of the percentage of saved data traffic for all users' visits to all webpages. The median is 18%, indicating that 18% of the transferred traffic could be saved by SWAROVsky compared to directly visiting the original browser. In summary, we can confirm that SWAROVsky benefits users in real-world scenarios.

10 Discussion

The preceding section demonstrates the effectiveness of our system. In this section, we discuss some issues that our current design and implementation have to address further.

10.1 Impact by HTTPS

The increased personalization of Internet services and rising concern over users' privacy on the Internet have led to a number of webpages accessing over HTTPS. HTTPS requires the end-to-end encryption that all functionalities must reside at the endpoints. Indeed, middleboxes such as SWAROVsky inherently cannot realize such functionalities in the context of HTTPS.

However, as middlboxes are so useful and desirable, people have been proposing solutions to enable HTTPS middlebox. One solution is the trusted middleboxes with special keys that allow them to terminate and split TLS connections [20]. The middleboxes are fully or partially trusted by endpoints so that the trusted middleboxes can function on the network. The other solution is to split traffic into HTTP and HTTPS [21], respectively. Public contents can be transferred via HTTP and thus benefit from the existing middleboxes, while private contents are still transferred via HTTPS to preserve security.

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. [20], [22]. Hence, supporting HTTPS in SWAROVsky is out of the scope of this article, and we do not evaluate the performance improvement under HTTPS.

10.2 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 these resources to be directly fetched from the original Web servers.

Our current implementation uses CEF to generate a resource loading graph and obtain resource content. However, the same webpage may require different resources on different browsers or devices. For example, high-quality images are loaded on devices with large-size high-resolution screens, while low-quality images are loaded on small-screen devices. Furthermore, due to the differences of Web engines, resource loading can vary a lot. As a possible solution, we can configure the size and resolution of CEF to simulate different devices. We plan to study the differences across browsers in future work.

10.3 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.

10.4 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 [23], [24]. 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 [25], [26], [27]. 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.

11 Related Work

Measuring and improving the user experience of mobile Web browsing have gained much attention in recent years. In this section, we highlight related work on resource loading process of mobile Web browsing and compare them with our approach.

11.1 Measurement of Resource Loading

Wang *et al*. [2] advocated that resource loading contributes most to the browser delay. Wang *et al*. [28] designed a lightweight in-browser profiler, called WProf, and studied the dependencies of activities when browsers load a webpage. Nejati *et al*. [29] extended WProf to WProf-M and studied the differences of page loading process between mobile and non-mobile browsers. Li *et al*. [30] designed WebProphet to capture dependencies among Web resources and to automate the prediction of user-perceived Web performance. Our system uses a resource loading graph, which is similar to WProf and WebProphet, but we focus only on the network loading sequence. We do not explicitly consider the computation dependencies that could potentially influence the loading order. Qian *et al*. [31] investigated the Web caching on mobile devices and found that about 20% of the total Web traffic under study is redundant due to imperfect cache implementations. In their subsequent work [32], they revealed poor resource utilizations of mobile Web browsing. In our previous work [7], [8], we measured and analyzed the poor performance of mobile Web caching resulted from imperfect cache configurations. Our work in this article is motivated by these previous measurement results that give us the direction to optimize the resource loading process.

11.2 Improvement of Resource Loading

New protocols and specifications. The shortcomings of HTTP/1.1 [33] are widely known. SPDY [9], [18] and HTTP/2 [10] have been proposed to mitigate the shortcomings by providing stream transmission, response priority, and server push. HTML 5 provides two server hint mechanisms [34], [35] to allow browsers to know what resources to be loaded so that the browser can prefetch them. These protocols and specifications provide mechanisms to improve resource loading but they rely on Web developers to adopt and configure these protocols and specifications in their own way.

Client-side improvement. Koukoumidis *et al*. [26] proposed PocketSearch to prefetch slowly-changed resources on mobile devices nightly when the devices are charging. In their subsequent work [16], they designed PocketWeb to prefetch dynamic resources in a timely fashion before user requests arrive. Zhang *et al*. [36] designed CacheKeeper, a system-wide service to effectively reduce overhead caused by poor Web caching for Android apps. The local resource proxy in our system plays a similar role as the CacheKeeper. Wang *et al*. [37] examined how Web browsing can benefit from a micro-cache that separately caches layout, code, and data at a fine granularity. Wang *et al*. [1] examined three client-only solutions to accelerate page load time: caching, prefetching, and speculative loading. They argued that infrastructure support to improve resource loading is necessary. Our system performs prefetch on the remote proxy rather than on the client device and we leverage the remote proxy to help maintain the local cache.

Proxy-based improvement. Proxy-based solutions are widely adopted in current Web architecture. Table 2 summarizes some state-of-the-art proxy-based solutions for mobile Web browsing. Some latest commodity browsers such as Amazon Silk Browser [38] and OperaMini [39] leverage a proxy architecture to offload some processes of page rendering to the cloud. Sivakumar *et al*. [6] proposed PARCEL to push Web resources in one or several bundles to a mobile device. PARCEL has to maintain the resource status on all its clients in order to reduce unnecessary transfers. Wang *et al*. [40], [41] designed EEP for energy-efficient mobile Web browsing. EEP follows the dual-proxy architecture. The target webpages is first rendered on the proxy and the resources are transferred to the client in a bundle. Sehati *et al*. [42] designed WebPro, leveraging a proxy-based approach to reduce latency. The proxy generates the profiles for webpages indicating the resource list. Rather than rendering the webpage when requested, WebPro uses the resource profile to acquire latest resources.

Butkiewicz *et al*. [4] designed KLOTSKI to prioritize resources that are most relevant to a user's preferences. KLOTSKI consists of two parts. The back-end captures invariant characteristics of a webpage by repetitively loading the webpage. The front-end determines transmission priority of resources based on user-specified preferences. Compared to KLOTSKI, our system not only realizes pushing resources before they are requested, but also focuses on eliminating redundant resource requests. Therefore, we could achieve reduction on both page load time and data usage. Agababov *et al*. [5] presented Google's data compression proxy Flywheel to reduce the network traffic of mobile Web browsing. Flywheel adopts a number of techniques to reduce transferred data, including image transcoding, minification of JavaScript and CSS, and gzip compression. The SWAROVsky system also uses gzip compression to reduce the size of transferred resources. But we do more work on not transferring unnecessary resources rather than only compressing them. Han *et al*. [43] designed MetaPush, a server push framework to reduce page load time without increasing network traffic. MetaPush leverages a meta file to make browsers aware of what resources to be loaded in the current page as well as in the potential subsequent pages. The client can determine what to prefetch according to the meta file. The framework is similar to our resource

TABLE 2
Comparison between SWAROVsky and related systems.

	Optimization level	Optimization process	Timing to perform the optimization	Network transmission	Deployment	Application
SWAROVsky	Resource object	Preloading resources	Preprocesss before webpage visits	Private protocol above SPDY/HTTP2	Personal cloud service	Work for all browsers
Silk	Page	Run complex computation	On the webpage visits	Private protocol above HTTPS	Split browser	Only work for the specific browser
OperaMini	Page	Pre-rendering and bundling	On the webpage visits	Private protocol above HTTPS	Split browser	Only work for the specific browser
Flywheel	Resource object	Compress	On the webpage visits	HTTPS	Public service	Only work for the specific browser
PARCEL	Page	Bundle resources	Preprocesss before webpage visits	Private protocol over TCP	Public service	Work for all browsers
EEP	Page	Pre-rendering and bundling	On the webpage visits	Private protocol over TCP	Personal cloud service	Work for all browsers
WebPro	Resource object	Profiling resources	Preprocesss before webpage visits	Private protocol over TCP	Public service	Work for all browsers
KLOTSKI	Resource object	Derive the transmission priority	Preprocesss before webpage visits	SPDY	Public service	No modification for browsers
MetaPush	Resource object	Retrieve resources to enable push	Preprocesss before webpage visits	HTTP2	Public service	Modify the browser kernel
Shandian	Page	Offload computations	On the webpage visits	Private protocol over TCP	Split browser	Modify the browser kernel

synchronizer, but we also capture the dependencies of resources and actively push resources back to the client. Wang *et al*. [44] designed Shandian, a split-browser architecture that restructures the page load process to speed up page loads. For each page to be visited, Shandian first loads the page on the proxy server, synchronizes the state of the page load process to the client when the onload event is fired, recovers the state on the client, and resumes the remaining process. Shandian requires to modify the browser kernel to realize the state synchronization, but our system can work seamlessly with latest commodity browsers by just deploying a separated client-side proxy.

12 Conclusion

In this article, we have presented SWAROVsky, a dual-proxy system to optimize resource loading for mobile Web browsing. Our system includes novel designs, including a resource loading graph, a resource synchronizer, and a resource matching algorithm, to optimize the resource loading without requiring modification of browsers and with quite marginal system overhead.

Some further improvements are ongoing. First, we are designing a storage management algorithm based on learning user access traces to remove unnecessary resources stored in the local proxy. Second, we are studying the shared resources of different webpages to build global resource loading graphs in order to further reduce redundant transfers when users navigate among webpages. Third, we plan to expand the user study and learn the user behaviors to further improve the applicability and efficiency of SWAROVsky.

Acknowledgment

This work was supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A203, the Natural Science Foundation of China (Grant No. 61370020, 61421091, 61528201, 61529201), and the Microsoft-PKU Joint Research Program. 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. The first two authors, Xuanzhe Liu and Yun Ma, contributed equally to this work.

References

[1] 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.

[2] ——, "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.

[3] V. Sevani and B. Raman, "HTTPDissect: Detailed performance analysis of HTTP web browsing traffic in TDMA mesh networks," *IEEE Transactions on Mobile Computing*, vol. 15, no. 4, pp. 853–867, 2016.

[4] 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.

[5] 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.

[6] A. Sivakumar, S. P. Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen, "Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014*, 2014, pp. 325–336.

[7] 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.

[8] 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.

[9] "Spdy protocol - draft 3," 2015. [Online]. Available: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3

[10] "HTTP/2," 2015. [Online]. Available: https://http2.github.io/

[11] "Jetty," 2014. [Online]. Available: http://www.eclipse.org/jetty/

[12] "Chromium embedded framework," 2015. [Online]. Available: https://bitbucket.org/chromiumembedded/cef/

[13] "Chrome remote debugging protocol," 2015. [Online]. Available: https://developer.chrome.com/devtools/docs/debugger-protocol

[14] Y. Tian, B. Song, and E. N. Huh, "Towards the development of personal cloud computing for mobile thin-clients," in *Proceedings of International Conference on Information Science and Applications, ICISA 2011*, 2011, pp. 1–5.

[15] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: understanding personal cloud storage services," in *ACM Conference on Internet Measurement Conference, IMC 2012*, 2012, pp. 481–494.

[16] 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.

[17] "Alexa top sites," 2016. [Online]. Available: http://www.alexa.com/topsites

[18] 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.

[19] "3G/4G wireless network latency: Comparing verizon, AT&T, Sprint and T-Mobile in February 2014," 2014.

[20] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. R. Rodríguez, and P. Steenkiste, "Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, 2015, pp. 199–212.

[21] Z. Zhou and T. Benson, "Towards a safe playground for HTTPS and middle boxes with QoS2," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox 2015*, 2015, pp. 7–12.

[22] T. Fossati, V. K. Gurbani, and V. Kolesnikov, "Love all, trust few: on trusting intermediaries in HTTP," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2015*, 2015, pp. 1–6.

[23] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," *Acm Sigcomm Computer Communication Review*, vol. 41, no. 4, pp. 374–385, 2011.

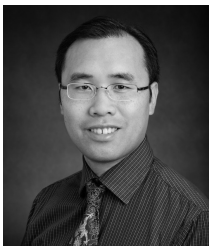
[24] G. Huang, H. Cai, M. Swiech, Y. Zhang, X. Liu, and P. Dinda, "DelayDroid: an instrumented approach to reducing tail-time energy of android apps," *Science China Information Sciences*, vol. 60, no. 1, 2017.
[25] M. Satyanarayanan, P. Bahl, R. Cceres, and N. Davies, "The case for VM-Based Cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.
[26] E. Koukoumidis, D. Lymberopoulos, K. Strauss, J. Liu, and D. Burger, "Pocket cloudlets," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011*, 2011, pp. 171–184.
[27] Y. J. Xing, Y. Zhi, C. Chi, and Y. F. Dai, "Beehive: low-cost content subscription service using cloudlets," *Science China Information Sciences*, vol. 56, no. 7, pp. 1–16, 2013.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] "Rfc 2616," 2014. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.txt
[34] "Server hint (prefetch)," 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Link_prefetching_FAQ
[35] "Server hint (subresource)," 2015. [Online]. Available: https://www.chromium.org/spdy/link-headers-and-server-hint/link-rel-subresource
[36] 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.
[37] 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.
[38] "Amazon silk browser," 2014. [Online]. Available: http://aws.amazon.com/cn/documentation/silk/
[39] "Opera mini browser," 2014. [Online]. Available: http://www.opera.com/mobile/mini/iphone
[40] L. Wang, B. Yu, and J. Manner, "Proxies for energy-efficient web access revisited," in *Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking*, 2011, pp. 55–58.
[41] L. Wang and J. Manner, "Energy-efficient mobile web in a bundle," *Computer Networks*, vol. 57, no. 17, pp. 3581 – 3600, 2013.
[42] A. Sehati and M. Ghaderi, "Webpro: A proxy-based approach for low latency web browsing on mobile devices," in *Proceedings of the 23th International Workshop on Quality of Service, IWQoS 2015*, 2015.
[43] B. Han, S. Hao, and F. Qian, "Metapush: Cellular-friendly server push for http/2," in *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, 2015, pp. 57–62.
[44] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with Shandian," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016*, 2016, pp. 109–122.

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 analytics. 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.

Xinyang Wang received his Bachelor degree in Computer Science from Peking University in 2015. He is currently pursing his Masters degree in Computer Science at the University of Washington. His field of interest includes mobile network, wearable computing, and software engineering.

Yunxin Liu is a Lead Researcher in Microsoft Research. His research interests are 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 serves as the correspondence of this article.