Empirically Assessing Opportunities for Prefetching and Caching in Mobile Apps

Yixue Zhao
University of Southern California
Los Angeles, California, USA

Paul Wat
University of Southern California
Los Angeles, California, USA

Marcelo Schmitt Laser
University of Southern California
Los Angeles, California, USA

Nenad Medvidović
University of Southern California
Los Angeles, California, USA

ABSTRACT

Network latency in mobile software has a large impact on user experience, with potentially severe economic consequences. Prefetching and caching have been shown effective in reducing the latencies in browser-based systems. However, those techniques cannot be directly applied to the emerging domain of mobile apps because of the differences in network interactions. Moreover, there is a lack of research on prefetching and caching techniques that may be suitable for the mobile app domain, and it is not clear whether such techniques can be effective or whether they are even feasible. This paper takes the first step toward answering these questions by conducting a comprehensive study to understand the characteristics of HTTP requests in over 1,000 popular Android apps. Our work focuses on the prefetchability of requests using static program analysis techniques and cacheability of resulting responses. We find that there is a substantial opportunity to leverage prefetching and caching in mobile apps, but that suitable techniques must take into account the nature of apps' network interactions and idiosyncrasies such as untrustworthy HTTP header information. Our observations provide guidelines for developers to utilize prefetching and caching schemes in app development, and motivate future research in this area.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

prefetching, caching, mobile apps, network latency, empirical study

ACM Reference Format:
Yixue Zhao, Paul Wat, Marcelo Schmitt Laser, and Nenad Medvidović. 2018. Empirically Assessing Opportunities for Prefetching and Caching in Mobile Apps. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238215

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238215

1 INTRODUCTION

There are over 5 billion mobile phone users and millions of mobile apps today [22]. The latency in mobile apps has been shown to have a large impact on user experience and potentially severe economic consequences [43]. The main cause of user-perceived latency is the network, since the majority of mobile apps fetch data from the Internet regularly [33].[1] Moreover, mobile devices rely on wireless networks, which can exhibit intermittent connectivity and low bandwidth [21].

Optimizing network performance has long been studied in distributed systems, and prefetching and caching techniques have been shown as a high-reward way to reduce network latency: they can bypass the performance bottleneck (network speed) and mask latency by returning a response to a request from a local cache immediately [21]. While prefetching makes use of built-in caching schemes, caching-only techniques are also widely employed (e.g., [31, 46]). In this work, we study the two related phenomena separately: *prefetchability* involves prefetching plus caching, while *cacheability* involves only caching.

The research on prefetching and caching techniques in the web browser domain has yielded a large body of work [12, 26, 28, 34, 36, 38, 40, 43]. However, the resulting techniques cannot be applied to mobile apps due to their different root causes of network latency. In the browser domain, the bottleneck for latency is resource loading since a large number of resources—usually files such as images—are needed within each HTTP request [42]. In the mobile app domain, each request only fetches a single response, and additional requests need to be issued explicitly to fetch further resources [23, 49]. Thus, prefetching and caching techniques in the browser domain target subresources within a single request [26, 34, 40, 43], while the research in the mobile app domain focuses on separate HTTP requests [46, 49].

Mobile users currently spend more than 80% of their time in mobile apps, rather than using mobile browsers [14]. Aside from a couple of exceptions, there has been a lack of research on prefetching and caching techniques that may be suitable for the mobile app domain. In fact, it is currently not clear whether such techniques can be effective or whether they are even feasible in practice. CacheKeeper [46] made an initial effort to study the redundant web traffic in mobile apps and proposed an OS-level caching service. However, the resulting service was only evaluated on 10 apps. Furthermore, CacheKeeper's performance highly depends on the flaws

[1] In this context, we define latency as the response time of an HTTP request.

in the web caching strategies employed in the original app, and its broader utility is unclear. Our previous work PALOMA [48, 49] used program analysis to identify HTTP requests that should be prefetched in mobile apps. We highlighted several program analysis challenges that can improve prefetching if addressed. However, PALOMA was evaluated on 32 apps. It is thus unclear to what extent PALOMA will be effective at a larger scale, and whether addressing the identified program analysis challenges is worthwhile.

The dearth and shortcomings of previous work motivated us to conduct a more extensive empirical study that aims to understand the characteristics of HTTP requests in mobile apps. In this paper, we report our results from the automated analysis of 1,687 most popular Android apps, spread across 33 app categories. Our work focuses on the prefetchability of requests (PALOMA's problem space) and cacheability of resulting responses (CacheKeeper's problem space). We found that a large number of HTTP requests used in real apps are prefetchable and the responses to those requests cacheable. This has the potential for significant reductions in user-perceived latency, which would, in turn, render the use of certain mobile apps even more attractive.

At the same time, our study highlighted the need to carefully consider which requests should be prefetched and which data cached, for two reasons. First, we empirically demonstrated the frequent lack of discipline with which developers use the relevant HTTP headers in mobile (specifically, Android) apps, making those headers misleading. Second, we showed that responses to certain HTTP requests that seem like good candidates for caching may yield incorrect app behaviors due to cache staleness.

Our study is the first to provide extensive empirical evidence regarding the opportunities for prefetching and caching in mobile apps. It is also the first to identify concrete shortcomings in the current app development practices that are guaranteed to hinder solutions that may otherwise seem easy and intuitive. As a result, the study has the potential to motivate significant future research in this area. In this paper, we have identified several promising research directions.

The remainder of the paper is organized as follows. Section 2 overviews the HTTP protocol and its use in the mobile app domain. Section 3 motivates and states our research questions. Section 4 describes our collection and processing of the subject apps. Section 5 discusses our findings and Section 6 describes the threats to their validity. A discussion of related work and conclusions round out the paper.

2 BACKGROUND

In this section, we overview aspects of the HTTP protocol that are relevant to prefetching and caching. We then illustrate with concrete examples of how developers perform network operations in mobile apps, with a particular focus on Android.

2.1 HTTP Protocol

Previous studies have shown that mobile apps spend between 34% and 85% of their time fetching data from the Internet [33]. The majority of apps run over HTTP [15], where requests are sent by clients and responses returned by servers.

An *HTTP request* consists of an HTTP method, the destination of the resource to fetch (i.e., the URL), and request headers and body, both of which are optional. The HTTP method—`GET`, `POST`, `DELETE`, etc.—needs to be specified by developers when sending a request. Optional request headers allow the client to pass additional information to the server [17], such as `Accept-Language: en-US`. The request body contains the resource to send to the server, but is only needed for "write" HTTP methods, such as `POST`.

HTTP 1.1 [18] defines eight methods. Some of them, such as `DELETE`, are not suitable for prefetching because they may change the server's state contrary to the user's intention. Only the `GET` and `HEAD` methods are considered "safe", in that they result in the retrieval of data and do not have any side-effects on the server [19, 20]. The `HEAD` method is similar to `GET`, except that its response does not contain a message body [19]. Thus, `GET` requests are of particular interest in our study.

An *HTTP response* consists of a status code, a status message, and response headers and body, both of which are optional. The status code and status message indicate whether the request was successful or not, and why. The response body contains the fetched resource from the server. Response headers contain additional information that is often used by developers to decide on their caching strategies. For example, the `Expires` header specifies when the response will become stale, while `Cache-Control` header contains the information pertaining to caching mechanisms such as `no-cache` and `max-age`. Interestingly, as observed in our study (see Section 5), those headers cannot always be trusted by developers, and sometimes they are missing altogether.

```
1 URL url = new URL("http://www.ase.com/post");
2 URLConnection conn = url.openConnection();
3 conn.setRequestMethod("POST");
4 conn.setRequestProperty("Accept-Language", "en-US");
5 OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream());
6 wr.write("post_data_to_send");
7 wr.flush();
8 InputStream responseStream = conn.getInputStream();
9 Map headerMap = conn.getHeaderFields();
```

Listing 1: Sending a POST request using the URLConnection library

```
1 OkHttpClient client = new OkHttpClient();
2 Request request = new Request.Builder()
3                   .url("http://www.ase.com/post")
4                   .addHeader("Accept-Language", "en-US")
5                   .post("post_data_to_send")
6                   .build();
7 Response response = client.newCall(request).execute();
8 Headers headers = response.headers();
```

Listing 2: Sending a POST request using the OkHttp library

2.2 HTTP Libraries Used in Mobile Apps

In Android apps, developers use off-the-shelf HTTP libraries to interact with servers. Listing 1 and Listing 2 demonstrate how developers send HTTP requests and receive responses using the two most popular HTTP libraries for Android: `URLConnection` and `OkHttp`.

When sending HTTP requests, developers need to specify the URL of the resource to be fetched (Listing 1: line 1, Listing 2: line 3), HTTP method (Listing 1: line 3, Listing 2: line 5), request headers

(line 4 in both Listings), and request body (Listing 1: line 6, Listing 2: line 5). Only the URL is mandatory and GET method will be used by default if the HTTP method is not specified (e.g., if line 3 in Listing 1 and line 5 in Listing 2 are removed). When receiving HTTP responses, developers can retrieve the response body (Listing 1: line 8, Listing 2: line 7) as well as the response headers (Listing 1: line 9, Listing 2: line 8) that may contain caching information.

3 RESEARCH QUESTIONS

The goal of this paper is to understand whether prefetching and caching can be applied to the mobile app domain effectively, in order to reduce user-perceived latency. We formulated nine research questions (RQs) to this end. These RQs target the *prefetchability* of HTTP requests, *cacheability* of HTTP responses, and *redundancies* among HTTP requests.

3.1 Prefetchability of HTTP Requests

Our objective is to assess the extent to which requests in mobile apps are *prefetchable*. Prefetchable requests are read-only requests that have no side-effects on the server state. As discussed above (Section 2.1), in the context of the HTTP protocol these are GET requests [19]. Furthermore, we study whether the prevalence of prefetchable requests varies across different app categories. Such variations may allow identifying app categories that are particularly suitable for prefetching.

We formulate three research questions to this end:

- **RQ$_1$** – What is the number of GET requests per app?
- **RQ$_2$** – What is the percentage of GET requests among all HTTP requests in mobile apps?
- **RQ$_3$** – How prevalent are GET requests across different app categories?

3.2 Cacheability of HTTP Responses

A prefetchable request may not be *cacheable* if the response to the request changes over time (e.g., in the case of weather data). In such cases, the cached response may be stale and serving it would lead to incorrect app behavior. To determine when a response becomes stale, or whether a request is cacheable at all, developers have to rely on the header information specified in the response, specifically, Expires and Cache-Control (recall Section 2.1). However, there are no standard rules for developers to follow when constructing a response, leaving open the possibility that header information may be unreliable or even missing.

To investigate this, we formulate four additional research questions:

- **RQ$_4$** – How prevalent are Expires headers?
- **RQ$_5$** – Are Expires headers trustworthy?
- **RQ$_6$** – How prevalent are Cache-Control headers?
- **RQ$_7$** – Are Cache-Control headers trustworthy?

3.3 Identifying Truly Redundant HTTP Requests

Caching is only effective when there exist redundant requests for the same resource. An HTTP request is *redundant* if a previous request specified the same HTTP method and URL, and yielded the same response; the later request is redundant because the original response could have been stored locally and reused. Previous work [46] suggests an opportunity for mobile app-based caching techniques, in that it identified the presence of redundant HTTP traffic and showed that implementations of web caching are inadequate for mobile apps. Our work goes beyond identifying redundant HTTP requests and tries to assess the intent behind them. A set of *ostensibly redundant* requests could be generated on purpose (e.g., to retrieve updated weather information), and thus may not be *truly redundant*. If a caching scheme fails to consider this, it will lead to cache staleness. We thus consider the actual responses to the candidate redundant requests, aiming to distinguish among them and provide better insights for future caching techniques in mobile apps.

With this in mind, we formulate the last two research questions:

- **RQ$_8$** – How prevalent are redundant HTTP requests?
- **RQ$_9$** – Are the identified ostensibly redundant requests truly redundant?

4 DATA COLLECTION

This section details (1) the workflow we used for data collection, (2) the criteria behind our selection of subject apps, (3) app instrumentation, (4) our collection of data via runtime testing, and (5) the reasons for eliminating certain apps from the subject set before conducting further analysis. All of the raw data regarding our subject apps and the corresponding code are publicly available [9].

4.1 Data Collection Workflow

Figure 1 illustrates the workflow we implemented for collecting the data needed to answer the nine research questions stated above. The initial subject apps were downloaded from the Google Play Store (Section 4.2). The apps were automatically instrumented based on the information extracted from HTTP library documentation and the decompiled code of several sample apps (Section 4.3). The instrumented apps were automatically tested using randomly generated inputs to produce logs that contain the information needed to answer RQ$_1$–RQ$_4$, RQ$_6$, and RQ$_8$ (Section 4.4). We manually examined the apps that could not be tested due to problems such as installation failures and runtime crashes, to identify the root causes of the problems (Section 4.5). Finally, we automatically sent GET requests to the subject apps at different time intervals, to answer RQ$_5$, RQ$_7$, and RQ$_9$ (Sections 5.2 and 5.3).

4.2 Initial Set of Subject Apps

We downloaded 1,687 top-ranked apps across 33 categories from the Google Play Store in the United States. 1,308 of the apps could be processed by Soot [37], a state-of-the-art tool for instrumenting Android apps, as further discussed in Section 4.3. The sizes of those 1,308 apps vary between 16 KB and 103.4 MB. The total number of HTTP requests per app varied between 0 and 1,243 in our tests, as described in Section 4.4.

Table 1 summarizes the information about the 1,308 subject apps. The table shows the maximum and average numbers of HTTP requests per app for each category; the minimum number of HTTP requests in every category is 0 and we thus omit it from the table. Finally, the right-most column shows the number of apps in each

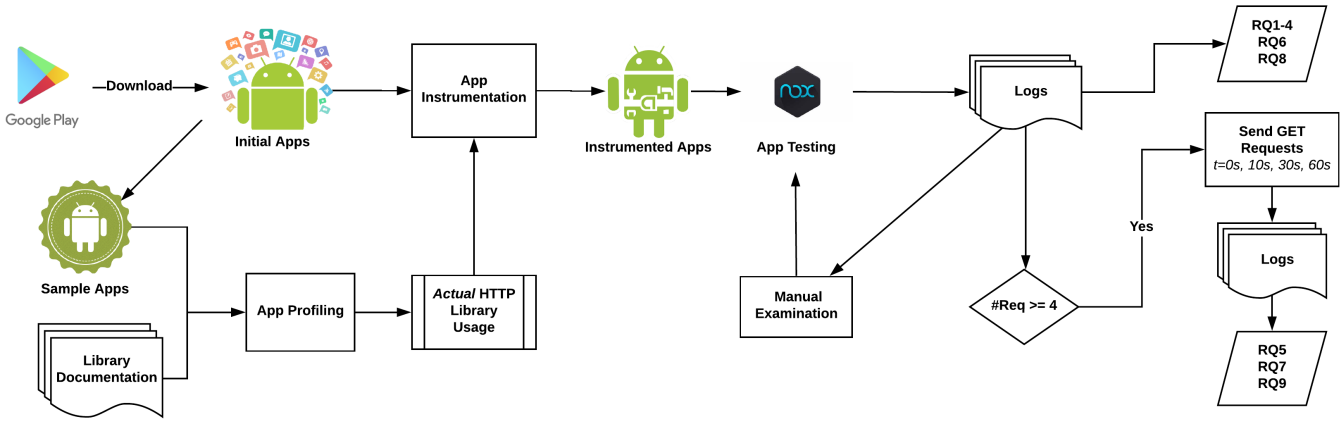

Figure 1: Our data collection workflow. The *App Profiling*, *App Instrumentation*, *App Testing*, and *Send GET Requests* components perform automated tasks.

category that sent at least four HTTP requests in our tests, as well as the percentage of such apps compared to the total number of apps in the given category. The reason behind highlighting this subset of the 1,308 subject apps will be explained in Section 4.5.

4.3 App Instrumentation

Each subject app went through an automated instrumentation process offline that used Soot [37] to insert code that captures information about HTTP requests and responses. This information is primarily located in the HTTP headers. Capturing such information in the browser domain is straightforward because HTTP requests and responses are managed in a unified way. On the other hand, mobile apps presented a challenge: we first had to identify how the HTTP requests and responses are handled in different HTTP libraries (recall Section 2.2); only then could we instrument the corresponding code to capture this information automatically.

It was thus necessary to determine what libraries most apps use to send HTTP requests. We first identified a set of popular HTTP libraries, including `URLConnection` [7], `OkHttp` [4], `Volley` [8], and `Retrofit` [5]. We then analyzed a sample of the subject apps' bytecodes and checked the package names against the libraries. For example, the presence of the string "java.net.URLConnection" generally indicates the use of the `URLConnection` library.

The data gathered from our analysis point to `URLConnection` and `OkHttp` as the most popular HTTP libraries used in the subject apps. This is unsurprising: `URLConnection` is the standard built-in library of the Android framework, and it has been augmented with `OkHttp` since Android v.4.4 (KitKat). We thus decided to focus on `URLConnection` and `OkHttp` in our study.

We then performed a more detailed analysis of how our subject apps use these two libraries. We recorded the runtimes of those methods that are imported from `URLConnection` and `OkHttp`, and narrowed our focus to methods that are most time-consuming. The rationale is that those are most likely to be the methods related to sending requests and receiving responses over the network.

In addition, we inspected the decompiled code of the subject apps, as well as the documentation and source code of the HTTP libraries used in the apps, to identify the actual usage of HTTP requests and responses. The reason for this additional inspection is that developers send requests and receive responses in various ways, even when using the same HTTP library. Listings 1 and 2 in Section 2 only demonstrate one common way of using each of the two HTTP libraries. While recommended in the libraries' documentation, there is no requirement or guarantee that developers will follow this guidance in their apps. Furthermore, the examples

Table 1: App information for each category among initial subjects

Category	#Apps	Max #Req	Avg #Req	#Apps (#Req≥4)
1. Art & Design	11	14	2.27	3 (27.27%)
2. Auto & Vehicles	29	6	1.07	4 (13.79%)
3. Beauty	11	1243	120.82	6 (54.55%)
4. Books & Reference	40	108	11.58	16 (40%)
5. Business	55	87	5.71	17 (30.91%)
6. Comics	55	319	20.84	19 (34.55%)
7. Communications	40	96	3.98	8 (20%)
8. Dating	16	334	29.94	6 (37.5%)
9. Education	55	62	4.98	17 (30.91%)
10. Entertainment	28	134	12.36	11 (39.29%)
11. Events	8	53	14.13	5 (62.5%)
12. Finance	61	150	15.97	27 (44.26%)
13. Food & Drink	28	188	16.43	13 (46.43%)
14. Games	37	59	12.59	25 (67.57%)
15. Health & Fitness	41	14	3.44	15 (36.59%)
16. House & Home	25	149	17.96	8 (32%)
17. Libraries & Demo	45	22	0.6	1 (2.22%)
18. Lifestyle	21	82	12.48	12 (57.14%)
19. Maps & Navigation	54	206	8.37	8 (14.81%)
20. Medical	59	63	2.8	10 (16.95%)
21. Music & Audio	43	44	5.47	14 (32.56%)
22. News & Magazines	49	802	37.71	26 (53.06%)
23. Parenting	24	28	2.54	5 (20.83%)
24. Personalization	31	288	29.61	11 (35.48%)
25. Photography	43	58	7.72	14 (32.56%)
26. Productivity	68	119	8.31	24 (35.29%)
27. Shopping	46	198	21.54	22 (47.83%)
28. Social	48	108	10.4	23 (47.92%)
29. Sports	43	146	19.42	18 (41.86%)
30. Tools	54	130	6.44	16 (29.63%)
31. Travel & Local	63	208	14.33	27 (42.86%)
32. Video Players & Editors	47	134	5.89	8 (17.02%)
33. Weather	30	123	14.7	12 (40%)
Total	**1308**	**1243**	**50.20**	**451 (34.48%)**

in the documentation are at the source code level, while our instrumentation using Soot [37] is at the bytecode level. This meant that we needed to understand the actual usage of those two HTTP libraries at the bytecode level. With the additional inspection, we were able to identify the actual methods used for sending requests and receiving responses in the apps, allowing us to instrument the code to capture the precise information needed for our study. For example, line 9 in Listing 1 defines `headerMap` that contains all of the header information; our instrumentation then inserts a method after line 9 to capture the headers relevant to our study, such as `Expires` header. It is important to note that the instrumented apps' primary functionality is left unchanged in this process.

4.4 App Testing

After the instrumentation, each app was subjected to random input testing through Android Debug Bridge (adb) [1]. We used the UI/Application exerciser tool Monkey [6] to generate random streams of user events, such as clicks, touches, and swipes. We used random events in this study for two reasons: (1) to avoid bias introduced by particular user behaviors and (2) to generate large volumes of runtime requests automatically, which would not be practical if we relied on a human user. This is further discussed in Section 6. The apps were run on the NoxPlayer Android emulator [3]. Each test consisted of 3,000 events under WiFi network settings. We also explored testing with 1,000, 5,000, and 10,000 events. We found that 3,000 was the smallest number of events that yielded a representative number of HTTP requests triggered at runtime across the subject apps; neither 5,000 nor 10,000 events resulted in a significant increase in HTTP requests, while 1,000 events proved to be too few to adequately exercise the relevant functionality in the apps.

All tests were preceded by a fresh installation of the given subject app, and the app was removed from the emulator after each test's conclusion. This minimized the chances of errors caused by any interference between apps or by previously saved settings.

4.5 Final Set of Subject Apps

The objective of our study is to determine whether and when HTTP requests should be prefetched and their responses cached. In some cases, the number of HTTP requests triggered in our tests was very low, suggesting that prefetching and caching in such apps would not be beneficial. To determine the nature of "low network usage" apps and the underlying reasons behind the data we obtained, we manually inspected each app, starting with those that do not trigger any requests.

A total of 623 out of the 1,308 subject apps triggered no requests. We identified six recurring reasons behind this:

(1) The app's installation failed.
(2) The app crashed upon launching.
(3) The app's version was incompatible with the NoxPlayer Android emulator [3].
(4) The app was obfuscated so that the methods relevant to HTTP requests were not captured by our instrumentation.
(5) The app required external information before it could be used, such as a bank PIN (commonly required in the *Finance* category) or a vehicle license plate (commonly required in the *Auto & Vehicles* category).
(6) The app only contained static content and did not rely on the network.

Note that, while we could not automatically test the above apps, many of them may, in fact, trigger HTTP requests at runtime. The only exception are apps from the last category. The automated nature of our app testing prevented us from determining the exact numbers of apps that fell in each of the above six categories. A manual inspection of a random sample of the apps suggests that, with a 95% confidence level, no more than 50% of the 623 apps contained only static content.

An additional 234 of the 1,308 subject apps triggered 1-3 requests at runtime. We observed a common pattern among these apps. Namely, regardless of the type of app, those requests tended to be one or more of the following:

(1) Load an application-specific configuration file.
(2) Log in with Facebook using Facebook `GraphRequest`.
(3) Use monitoring services, such as `Crashlytics` or `Google Analytics`.

Further manual testing of these apps yielded no additional HTTP requests beyond the above three. This finding shows a common usage of popular third-party services in mobile app development, whose impact on app performance should also be taken into account in terms of overhead, data usage, and energy consumption.

We were unable to identify any patterns such as the above in apps that trigger any other number of requests. Thus, the below analysis of *prefetchability* and *cacheability* is based on 451 of our subject apps that trigger four or more requests at runtime, corresponding to the right-most column of Table 1.

5 RESULTS AND DISCUSSION

This section describes the results of our analysis, framed by the nine research questions from Section 3, and discusses the lessons learned from the results. Table 2 summarizes the information about the final set of 451 subject apps in each category that are analyzed in this section. Note that the app categories are numbered 1-33, to aid the depiction and understanding of the figures in the remainder of this section. Among the 451 apps, the number of HTTP requests ranged between 4 (the cut-off number for our analysis, as discussed above) and 1,243, with the average slightly above 35 requests per app.

5.1 Prefetchability of HTTP Requests

Recall from Section 3 that we try to answer three research questions regarding the prefetchability of HTTP requests. Specifically, we are interested in `GET` requests, which are the primary candidates for prefetching.

- **RQ$_1$** – What is the number of `GET` requests per app?
- **RQ$_2$** – What is the percentage of `GET` requests among all HTTP requests in mobile apps?
- **RQ$_3$** – How prevalent are `GET` requests across different app categories?

To answer the above questions, we instrumented and tested our subject apps using the procedure described in Section 4. We calculated the total number of `GET` requests observed during our testing,

and the percentage of GET requests among all HTTP requests triggered at runtime in each app. We subsequently grouped the results by app category. Figure 2 depicts the minimum, maximum, and average *numbers* of GET requests per app (RQ_1) across the different categories (RQ_3). Figure 3 depicts the minimum, maximum, and average *percentages* of GET requests as compared to all HTTP requests (RQ_2) in each app category (RQ_3).

Our data indicate that GET requests are pervasive across all 33 app categories. As shown in Figure 2, seven categories contained apps that sent 150 or more GET requests. On average, an app sent 28 GET requests, and those requests comprised 68% of all HTTP requests sent by the app. As shown in Figure 3, several categories—*Beauty* (94%), *Comics* (87%), *Entertainment* (88%), and *Events* (87%)—had very high percentages of GET requests. Only two categories—*Dating* (43%) and *Tools* (44%)—had slightly fewer than 50% of GET requests.

These results suggest that there is a significant opportunity to exploit prefetching among the 451 subject apps that sent 4 or more HTTP requests. It was surprising to see that 102 apps, spanning 29 of the 33 categories, sent only GET requests. Certain categories are potentially more suitable for prefetching than others. This is a by-product of the types of functionality that are typical in a given category. The nature of apps in "stable" domains, such as *Art & Design* or *Libraries & Demo*, is such that they may be able to operate with less remotely accessed data than apps in more "dynamic" domains such as *News & Magazines* or *Shopping*. This suggests that prefetching and caching techniques may benefit from leveraging knowledge regarding an app's domain.

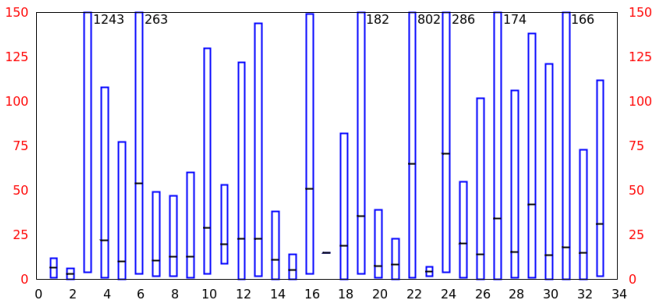

Figure 2: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *numbers* of GET requests in apps across the 33 app categories. Apps in 7 categories had maximums higher than 150 (numbers displayed beside the corresponding bars). Note that the average for app category 3 is also higher than 150, and thus not shown.

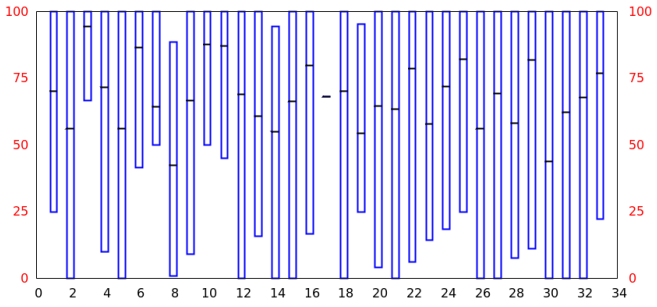

Figure 3: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *percentages* of GET requests in apps across the 33 app categories.

Table 2: App information for each category among final subjects

Category	#Apps	Min. #Req	Max. #Req	Avg. #Req
1. Art & Design	3	4	14	8.33
2. Auto & Vehicles	4	4	6	4.75
3. Beauty	6	4	1243	220.33
4. Books & Reference	16	4	108	27.94
5. Business	17	4	87	17.24
6. Comics	19	4	319	59.58
7. Communications	8	4	96	19
8. Dating	6	5	334	78.83
9. Education	17	4	62	15.06
10. Entertainment	11	6	134	30.73
11. Events	5	11	53	22.2
12. Finance	27	5	150	35.59
13. Food & Drink	13	4	188	33.46
14. Games	25	4	59	18
15. Health & Fitness	15	4	14	8.13
16. House & Home	8	4	149	55.38
17. Libraries & Demo	1	22	22	22
18. Lifestyle	12	4	82	21
19. Maps & Navigation	8	8	206	54.88
20. Medical	10	4	63	14
21. Music & Audio	14	5	44	16.14
22. News & Magazines	26	4	802	70.88
23. Parenting	5	4	28	12
24. Personalization	11	6	288	82.73
25. Photography	14	4	58	23
26. Productivity	24	4	119	22.67
27. Shopping	22	4	198	44.14
28. Social	23	4	108	20.35
29. Sports	18	7	146	45.67
30. Tools	16	4	130	21.44
31. Travel & Local	27	4	208	32.11
32. Video Players & Editors	8	4	134	33.63
33. Weather	12	7	123	36.17
Total	**451**	**4**	**1243**	**35.28**

5.2 Cacheability of HTTP Responses

As discussed in Section 3, the cacheability of HTTP responses is a function of the presence of `Cache-Control` and `Expires` headers, and their trustworthiness. To that end, we try to answer the following four research questions.

- **RQ$_4$** – How prevalent are `Expires` headers?
- **RQ$_5$** – Are `Expires` headers trustworthy?
- **RQ$_6$** – How prevalent are `Cache-Control` headers?
- **RQ$_7$** – Are `Cache-Control` headers trustworthy?

To answer the above questions, we instrumented the subject apps to capture response headers (recall Section 4.3) and calculate the numbers of occurrences of the two relevant headers. To determine whether the header of a given request is trustworthy, we made each request 4 times: at initial time t, $t + 10$, $t + 30$, and $t + 60$ seconds. This allowed us to determine whether later responses reflect what is specified in the header of the original response.

For example, let us assume that the original request is sent at `time(t)` and that the response header contains `Expires: time(exp)`.

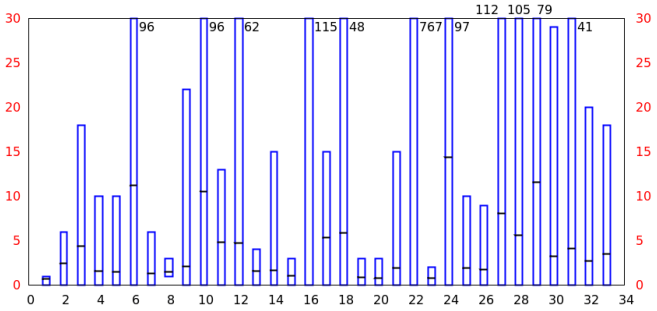

Figure 4: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *numbers* of `Expires` headers in each app category. Apps in 11 categories had maximums higher than 30 (numbers displayed beside or above the corresponding bars).

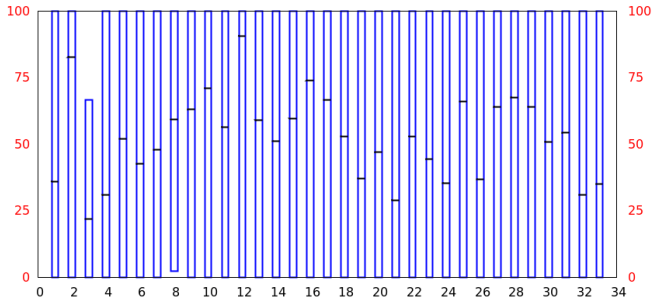

Figure 5: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *percentages* of the `Expires` headers for each app category.

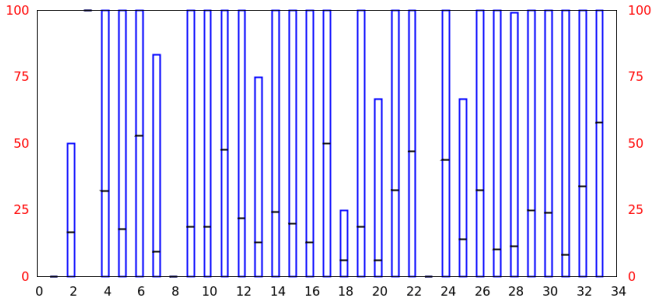

Figure 6: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) percentages of *trusted* `Expires` headers in each app category.

We will mark the header as untrustworthy if it falls into any of the following three cases, where `x` is the time period after the original request is sent:

(1) `time(exp) <= time(t)`
(2) `(time(t) < time(exp) ≤ time(t+x))`
    `∧ (response@(t) = response@(t+x))`
(3) `(time(exp) > time(t+x))`
    `∧ (response@(t) ≠ response@(t+x))`

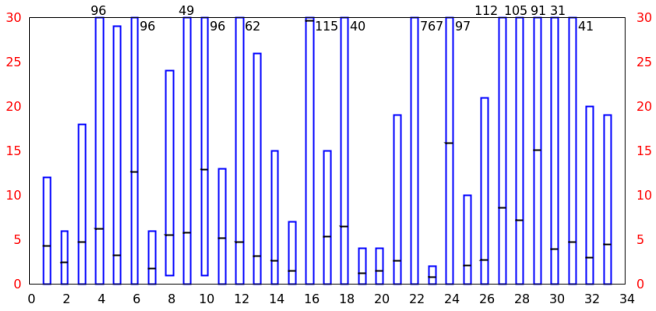

Figure 7: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *numbers* of `Cache-Control` headers in each app category. Apps in 14 categories had maximums higher than 30 (numbers displayed beside or above the corresponding bars).

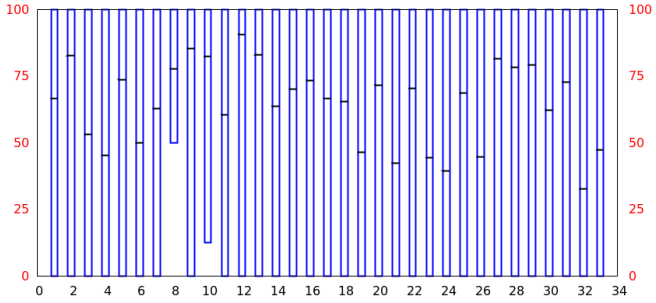

Figure 8: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *percentages* of `Cache-Control` headers in each app category.

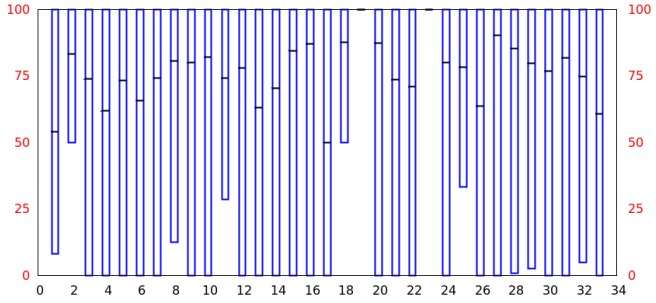

Figure 9: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) percentages of *trusted* `Cache-Control` headers in each app category.

In our case, `x` is any of 10*s*, 30*s*, or 60*s*. The first case indicates a scenario where the response expires before the request is even sent. The second case indicates a scenario where the response is supposed to have expired, but it has remained unchanged. Finally, the third case indicates a scenario where the response should have remained the same, but it changed.

We use the analogous algorithm to determine whether the `Cache-Control` header is trustworthy, based on the `max-age` field specified within the header.

Figure 4 shows the minimum, maximum, and average numbers of the `Expires` headers included in HTTP responses for each app category (RQ_4). Figure 5 shows the minimum, maximum, and average percentages of the `Expires` headers among all the response headers in each app category (RQ_4). Figure 6 shows the percentages of the trustworthy `Expires` headers among all the `Expires` headers (RQ_5). Figures 7, 8, and 9 show the analogous information for the `Cache-Control` header (RQ_6, RQ_7).

From the results, we can conclude that the `Expires` headers and `Cache-Control` headers are not always included in the responses, and they are not always trustworthy. The `Cache-Control` header tends to be used more reliably than the `Expires` header. Across the 33 app categories, 53% of the response headers contain `Expires` on average, while 65% contain `Cache-Control`. Only an average of 25% of the `Expires` headers are trustworthy, while 77% of the `Cache-Control` headers are trustworthy. While there are individual apps among our subjects where each of the two headers was used in a completely trustworthy manner (100%), there were an even greater number of apps where the opposite was true (0%).

These results strongly suggest that developers should not depend on the response headers to determine their caching schemes. Unfortunately, there are currently no reliable alternatives for the mobile app domain. However, this presents a research opportunity to investigate more intelligent approaches. One strategy that suggests itself based on our study would involve learning the correct information to include in the headers based on historical data. Such a technique could then automatically suggest app modifications, in order to fix the "buggy" headers.

5.3 Identifying Truly Redundant HTTP Requests

As discussed in Section 3, redundant HTTP requests are good candidates for prefetching and caching. However, certain HTTP requests are only *ostensibly redundant* in that they seem identical but actually yield different responses. Our final two research questions aim to shed light on this issue.

- **RQ$_8$** – How prevalent are redundant HTTP requests?
- **RQ$_9$** – Are the identified ostensibly redundant requests truly redundant?

In our analysis, we have specifically focused on `GET` requests, as discussed previously.

To answer the above questions, upon completion of testing a given app (by executing the 3,000 events as explained in Section 4.4), we identify the ostensibly redundant requests in each app. We then run a script that executes the app by sending each identified request four times: at initial time t, $t + 10$, $t + 30$, and $t + 60$ seconds. We check whether the responses change during this interval. This helps to identify HTTP requests that are truly redundant; the responses to those requests are thus suitable candidates for caching.

Figure 10 shows the minimum, maximum, and average percentages of the identified ostensibly redundant requests as compared to the total number of requests in each app category (RQ_8). Figure 11 shows the minimum, maximum, and average expiration times for the identified requests (RQ_9). A request's expiration time is the time at which its response is different from the response received for the initial request at time t. Finally, Figure 12 shows the minimum, maximum, and average percentages of the *truly* redundant requests (RQ_9).

As Figure 10 shows, redundant requests comprise a significant proportion of all HTTP requests across most of the app categories. In certain apps, nearly 100% of the requests are redundant, while the average across all apps is ≈20%. By themselves, these results would suggest considerable cacheability potential.

This is further bolstered by some of the results in Figure 11, which points to several apps in which the HTTP requests did not

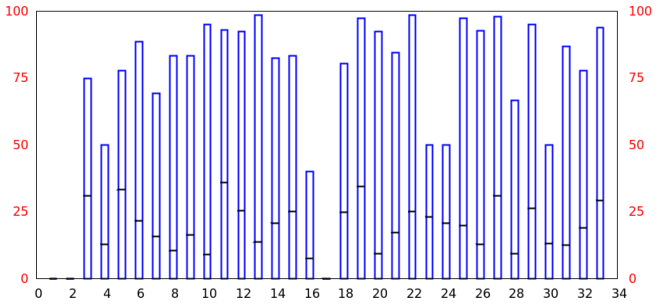

Figure 10: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) percentages of *ostensibly* redundant requests in each app category.

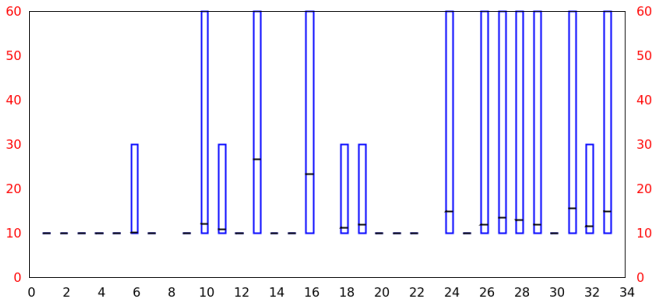

Figure 11: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) *expiration times* for the redundant requests in each app category.

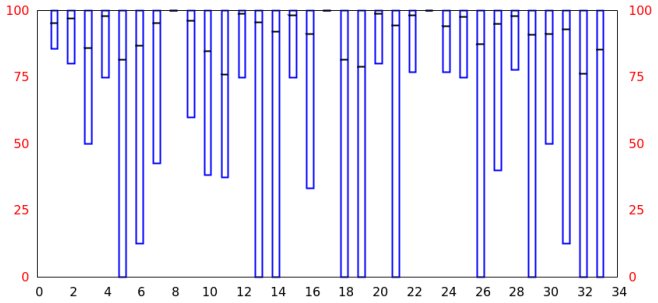

Figure 12: Minimum (bottom edges), maximum (top edges), and average (horizontal dashes) percentages of *truly* redundant requests in each app category.

expire even after the full 60s. However, this is somewhat deceptive: The average request expiration time was 12s across the 33 app categories; it was exactly 10s for several of the categories; and only two categories—*Food & Drink* and *House & Home*—had average expiration times over 20s. Since 10s was the shortest interval used in our study, these results suggest that most redundant requests expire within a relatively short time period. This should be taken into account when devising caching schemes for mobile apps.

Finally, Figure 12 shows that, on average, an overwhelming majority of ostensibly redundant requests are *truly redundant* across the 33 app categories. This means that the ostensibly redundant request did not expire at one or more of the 10s, 30s, and 60s checkpoints. In a number of individual apps, all ostensibly redundant requests are truly redundant (the maximum value of 100%), while their average for app categories is as high as 92%. This observation shows a large opportunity for caching redundant requests in mobile apps.

5.4 Implications

Our study provides evidence that prefetching and caching can be beneficial in a large number of mobile app scenarios. At the same time, we came across several apps in which prefetching and caching are unlikely to have significant, or any, benefits. At the least, these include the several hundred apps from our original set of subjects that only provide static content or make very few (1-3) HTTP requests. In fact, it is possible that the number of these apps surpasses the 451 apps that do rely on the network and that we included in our final set of subjects (recall the discussion in Section 4.5). This outcome was at least somewhat surprising, given the long history of research on data prefetching and caching in distributed systems, of which mobile apps are only a more recent example.

A deeper analysis helps to identify several reasons behind this. For example, in hindsight it may have been expected that apps from the *Libraries & Demo* or *Video Players & Editors* categories provide static content, such as PDF viewers, organizers, digital books, and video players. On the other hand, we did not expect to find almost as much static content in *Auto & Vehicles*. We already discussed in Section 4.5 that a number of apps from this category required login by suppling a license plate number. An additional, large number of apps also contained purely static content, such as instructions on how to perform car maintenance. This is reflected in our data: under 14% of the *Auto & Vehicles* apps made it into our final set of 451 subjects (recall Table 1).

Another issue was presented by apps that used network communication that was either not based on HTTP or extensively used HTTP methods other than `GET`. For example, a number of apps in the *Communications* category provide instant messaging capabilities (including VoIP), while others actually implement browsers. *Maps & Navigation* provide GPS applications that differ significantly from typical HTTP services. Yet another example are *Finance* apps. Even though 44% of these apps made it into our final set of subjects, a lot of them are banking apps that predominantly perform push-type operations, making them ill-suited for prefetching.

Even within the 451 final subject apps, there are clearly some for which the benefits of prefetching and caching may be marginal. *Games* presented an interesting case. Over $\frac{2}{3}$ of the apps in this category made it into our final set of subjects since they used sufficiently large numbers of HTTP requests. These apps also exhibited very high `Cache-Control` trustworthiness. On the other hand, as expected, their requests tended to expire very quickly and to have little redundancy. Therefore, while a game app may be identified as a candidate for prefetching, the resulting cached data would become stale very quickly. In turn, this would possibly lead to incorrect app behavior or, just as bad, constant thrashing of the prefetching facilities that would cripple the app's performance.

These issues can be further illustrated with a somewhat crude analysis of an average app from our subject set. The average app sent 28 `GET` requests (recall Section 5.1) as a result of the 3,000 automatically generated UI events. 20% of those requests were truly redundant (recall Section 5.3). That means that up to 6 `GET` requests were prefetchable. Our previous work PALOMA [49] measured the processing of a single HTTP request to take slightly over 800ms under network conditions similar to ours. This would mean that an average app among our subjects would save only 4s by caching and reusing the results of the original request, assuming that the cache does not become stale.

While we must be cognizant of apps, such as those above, that are not especially amenable to prefetching and caching, several scenarios in our study paint a much more favorable picture. Consider the app from category 3 (*Beauty*) that issued 1,243 `GET` requests (recall Figure 2), all of which are truly redundant (corresponding to the maximum value for app category 3 in Figure 12). Even if we assume that the result of each redundant request can only be reused once before it expires (recall from Figure 11 that the expiration time for app category 3 is 10s), that still yields 621 requests for which the results can be reused from the local cache. Assuming once again the same execution conditions as PALOMA's, this would result in massive execution-time savings, totaling 497s or 8.5 minutes.

In summary, there is a notable opportunity for prefetching and caching in the mobile app domain. At the same time, the resulting techniques must take into account the characteristics of different app categories and different HTTP requests. Otherwise, the employed techniques may yield undesired outcomes, such as cache staleness, non-trivial performance overhead, and incorrect app behaviors.

6 THREATS TO VALIDITY

Our study is based on top-ranked, free Android apps. Therefore, our results may not hold for paid apps or lower-ranked apps. However, over 90% of the Android apps in the Google Play Store are free [2]. Furthermore, top-ranked apps are used most widely. This suggests that our results should have broad applicability.

We excluded from our numerical analysis the apps that trigger fewer than four HTTP requests at runtime. However, part of the objective of our study was to explore this problem space. Specifically, we identified the reasons behind the apps' low numbers of requests (recall Section 4.5). Furthermore, we acknowledged explicitly that the exclusion of these apps from the final set of subjects limits the applicability of our findings (recall Section 5.4).

Our study is based on apps that use the HTTP protocol and two HTTP libraries (`URLConnection` and `OkHttp`). Our findings

are unlikely to be directly applicable to other protocols for network communication, and they may not carry over to other HTTP libraries. However, most mobile apps, and in particular Android apps, rely on HTTP [15]. Furthermore, our focus is on the fundamental characteristics of HTTP requests and responses, and those characteristics do not change across different HTTP libraries. Including other libraries would naturally result in the inclusion of greater numbers of subject apps. However, given the popularity of the HTTP libraries we selected, our results should be widely representative among Android apps.

In our process for answering RQ_5, RQ_7, and RQ_9, we sent out sets of four requests, at times t, $t + 10$, $t + 30$, and $t + 60$ seconds (recall Sections 5.2 and 5.3). As shown in Figure 11, redundant requests tend to expire at $t + 10$ or soon thereafter. This indicates that $t + 60$ is a sufficiently long period to identify truly redundant requests in most cases. Furthermore, mobile users tend to use an app for relatively short periods, so that prefetching and caching far in advance is not necessary and is likely to yield cache staleness. While choosing different time intervals would likely not lead to different results, finer-grained intervals may give us tighter bounds on request expiration times.

Finally, our app usage information was obtained via automated generation of UI events, as opposed to logging real user events. This may result in numbers and sequences of HTTP requests that are not representative of actual app use. However, the purpose of our study was to analyze all possible HTTP requests that could be potentially triggered at runtime, and 3,000 random events were shown to be able to generate representative HTTP requests, as discussed in Section 4.4. Given the nature of the study and the large number of apps we aimed to analyze, it would have been unreasonable to attempt to find actual users for each app, while our results would potentially suffer from user-specific biases and idiosyncrasies in engaging the app. On the other hand, mimicking actual users with humans who are unfamiliar with the apps in question, which would have been a more likely alternative, would have suffered from the same potential problem as our automated testing. Furthermore, all of our research questions focus on individual HTTP requests rather than their sequences. Thus, real user traces would not lead to different results compared to random orders of runtime events. Finally, neither actual nor novice human users would have been able to repeatedly and reliably generate large numbers of events (3,000 per app execution in the main portion of our study, and up to 10,000 per execution in the preliminary analysis).

7 RELATED WORK

Web prefetching and caching are entrenched techniques to reduce network latency since the Internet was born and have attracted a large body of work in browser domain, including measurement studies to understand web performance and identify performance bottlenecks [16, 24, 30–32, 42], literature reviews and quantitative studies to compare fundamental prefetching and caching algorithms [12, 39, 43], leveraging prefetching and caching techniques at different levels, such as studying user browsing behaviors [25, 36], providing API support for developers [26], restructuring page load process [28, 40], providing server or infrastructure support [10, 13, 34, 35, 47].

The recent surge of mobile devices has attracted researchers to study prefetching and caching techniques in the context of mobile browsers and mobile apps. With the foundation of the traditional research in browser domain, mobile browser performance soon became a crowded research area [24, 27, 35, 40, 41, 43], but the research on mobile apps is still in its infancy. This is unfortunate because mobile users currently spend more than 80% of their time in mobile apps rather than mobile browsers [14]. In mobile app domain, Cachekeeper [46] studied the redundant HTTP traffic and proposed an OS-level caching service for HTTP requests on smartphones. PALOMA [49] used program analysis to address "what" and "when" to prefetch certain HTTP requests in mobile apps. However, those techniques were only evaluated on a small number of apps and the performance depends on the flaws of web caching schemes employed in the original apps, thus it is not clear to what extent those techniques will be effective in practice. Those shortcomings of existing approaches motivated us to conduct an in-depth study that aims to understand the characteristics of HTTP requests in order to guide future research in mobile apps. Other existing works that focus on mobile app performance are complementary to our focus, such as pre-launching mobile apps [29, 44, 45], balancing Quality-of-Service (QoS) to suggest "how much" to prefetch [11, 21], identifying performance bottlenecks [33].

8 CONCLUSION

In this paper, we presented the results of an extensive empirical study aimed at understanding the characteristics of HTTP requests and responses in mobile apps. We formulated nine research questions with the focus on the *prefetchability* of HTTP requests and *cacheability* of HTTP responses. Our overarching objective is to fill in the gap between the well-studied browser domain and comparatively less-explored mobile app domain, by motivating and providing guidelines for future research in this area.

Our results suggest that prefetching and caching can be useful across a wide range of mobile apps and scenarios, but they are not universally applicable and their benefits will vary. Certain app categories are more amenable for prefetching and caching. However, there is a non-trivial amount of variation even among different apps within a single cateogry. While our analysis reported in this paper does not provide definitive answers to questions of *what*, *when*, and *how much* to prefetch/cache, it provides a process, tools, and data that form a foundation for answering those questions much more precisely than has been possible thus far.

ACKNOWLEDGMENT

The authors thank William G.J. Halfond, Jiaping Gui, and the rest of their research group at the University of Southern California for providing us with the APKs for our subject apps. This work is supported by the U.S. National Science Foundation under grants no. CCF-1618231 and CCF-1717963, U.S. Office of Naval Research under grant no. N00014-17-1-2896, and by Huawei Technologies Co., Ltd.

REFERENCES

[1] 2018. Android Debug Bridge. https://developer.android.com/studio/command-line/adb

[2] 2018. Distribution of free and paid Android apps in the Google Play Store. https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/

[3] 2018. NoxPlayer. https://www.bignox.com/

[4] 2018. OkHttp Documentation. http://square.github.io/okhttp/

[5] 2018. Retrofit Documentation. http://square.github.io/retrofit/

[6] 2018. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html

[7] 2018. URLConnection Class Documentation. https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html

[8] 2018. Volley overview. https://developer.android.com/training/volley/

[9] 2018. The website of the raw data and the code of our analysis. https://github.com/felicitia/PALOMA-Analysis/tree/empirical

[10] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web.. In *NSDI*, Vol. 15. 367–380.

[11] Paul Baumann and Silvia Santini. 2017. Every Byte Counts: Selective Prefetching for Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 2, Article 6 (June 2017), 29 pages. https://doi.org/10.1145/3090052

[12] Christos Bouras, Agisilaos Konidaris, and Dionysios Kostoulas. 2004. Predictive Prefetching on the Web and Its Potential Impact in the Wide Area. *World Wide Web* 7, 2 (June 2004), 143–179. https://doi.org/10.1023/B:WWWJ.0000017208.87570.7a

[13] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices.. In *NSDI*, Vol. 1. 2–3.

[14] Dave Chaffey. 2018. Mobile Marketing Statistics compilation. https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

[15] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*. IEEE, 809–817.

[16] Jeffrey Erman, Alexandre Gerber, Mohammad Hajiaghayi, Dan Pei, Subhabrata Sen, and Oliver Spatscheck. 2011. To cache or not to cache: The 3G case. *IEEE Internet Computing* 15, 2 (2011), 27–34.

[17] Roy Fielding. 1999. RFC 2616, part of Hypertext Transfer Protocol – HTTP/1.1. https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

[18] Roy Fielding. 1999. RFC 2616, part of Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616#section-5.1.1

[19] Roy Fielding. 1999. RFC 2616, part of Hypertext Transfer Protocol – HTTP/1.1. https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[20] Roy Fielding. 1999. RFC 2616, part of Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616#section-9

[21] Brett D Higgins, Jason Flinn, Thomas J Giuli, Brian Noble, Christopher Peplin, and David Watson. 2012. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 155–168.

[22] SIMON KEMP. 2018. Digital in 2018. https://wearesocial.com/blog/2018/01/global-digital-report-2018

[23] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of http requests for mobile applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 249–260.

[24] X. Liu, Y. Ma, Y. Liu, T. Xie, and G. Huang. 2016. Demystifying the Imperfect Client-Side Cache Performance of Mobile Web Browsing. *IEEE Transactions on Mobile Computing* 15, 9 (Sept 2016), 2206–2220. https://doi.org/10.1109/TMC.2015.2489202

[25] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. 2012. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2150976.2150978

[26] James W Mickens, Jeremy Elson, Jon Howell, and Jay R Lorch. 2010. Crom: Faster Web Browsing Using Speculative Execution.. In *NSDI*, Vol. 10. 9–9.

[27] Javad Nejati and Aruna Balasubramanian. 2016. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1305–1315.

[28] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking.. In *NSDI*. 123–136.

[29] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM, 275–284.

[30] Feng Qian, Junxian Huang, Jeffrey Erman, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2013. How to reduce smartphone traffic volume by 30%?. In *International Conference on Passive and Active Network Measurement*. Springer, 42–52.

[31] Feng Qian, Kee Shen Quah, Junxian Huang, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 127–140.

[32] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. 2014. Characterizing resource usage for mobile web browsing. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 218–231.

[33] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild.. In *OSDI*, Vol. 12. 107–120.

[34] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. [n. d.]. Push or request: An investigation of http/2 server push for improving mobile performance. In *Proceedings of the 26th International Conference on World Wide Web*. 459–468.

[35] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 390–403. https://doi.org/10.1145/3098822.3098851

[36] N. Swaminathan and S. V. Raghavan. 2000. Intelligent prefetch in WWW using client behavior characterization. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*. 13–19. https://doi.org/10.1109/MASCOT.2000.876424

[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[38] Haoyu Wang, Junjun Kong, Yao Guo, and Xiangqun Chen. 2013. Mobile web browser optimizations in the cloud era: A survey. In *2013 IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*. IEEE, 527–536.

[39] Jia Wang. 1999. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Comput. Commun. Rev.* 29, 5 (Oct. 1999), 36–46. https://doi.org/10.1145/505696.505701

[40] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian.. In *NSDI*. 109–122.

[41] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. How effective is mobile browser cache?. In *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students, for the students*. ACM, 17–20.

[42] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why are web browsers slow on smartphones?. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 91–96.

[43] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How far can client-only solutions go for mobile browser speed?. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 31–40.

[44] Ye Xu, Mu Lin, Hong Lu, Giuseppe Cardone, Nicholas Lane, Zhenyu Chen, Andrew Campbell, and Tanzeem Choudhury. 2013. Preference, context and communities: a multi-faceted approach to predicting smartphone app usage patterns. In *Proceedings of the 2013 International Symposium on Wearable Computers*. ACM, 69–76.

[45] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 113–126.

[46] Yifan Zhang, Chiu Tan, and Li Qun. 2013. CacheKeeper: a system-wide web caching service for smartphones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM, 265–274.

[47] Bo Zhao, Byung Chul Tak, and Guohong Cao. 2014. *Mobile web browsing using the cloud*. Springer.

[48] Y. Zhao. 2017. Toward Client-Centric Approaches for Latency Minimization in Mobile Applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 203–204. https://doi.org/10.1109/MOBILESoft.2017.34

[49] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. 2018. Leveraging Program Analysis to Reduce User-Perceived Latency in Mobile Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*.