Navigation-aware and Personalized Prefetching of Network Requests in Android Apps

Ivano Malavolta†, Francesco Nocera*, Patricia Lago†, Marina Mongiello*
†Vrije Universiteit Amsterdam, The Netherlands
*Polytechnic University of Bari, Italy
{i.malavolta | p.lago}@vu.nl, {francesco.nocera | marina.mongiello}@poliba.it

***Abstract*—Prefetching network requests has been advocated as a highly effective way of reducing network latency experienced by the user since it allows network responses to be generated immediately from a local cache.**

In this paper we discuss how user navigation patterns can be used for developing navigation-aware techniques for personalized prefetching of network requests of Android apps. The proposed idea opens for a new family of prefetching opportunities since it focusses at a higher level of abstraction with respect to state-of-the-art approaches for network requests prefetching. The proposed idea allows the development of approaches which adapt their prefetching behaviour according to the unique navigation patterns each user exhibits while interacting with a mobile app.

***Keywords*-Android, Network Prefetching, Personalization**

I. Introduction

The fact that mobile apps rely on wireless connectivity (*e.g.,* 3G, WiFi) is emerging as a recurrent challenge for mobile developers, who must take into account the unpredictability of the network underlying their apps [1], [2]. For developers, failing to properly consider network transfers may negatively affect the user experience, and in some cases hinder an effective usage of the app itself [3]. In turn, this can impact the app user ratings and reviews, which, unless properly addressed, can negatively impact the app's success.

Prefetching network requests has been advocated as an effective way of reducing the latency experienced by the users since it allows network responses to be generated *immediately* from a local cache [2]. However, despite their promising results, existing prefetching approaches for mobile apps (*e.g.,* [4]) can still be improved in many ways: (i) they *neglect the interaction patterns of each individual user* (*e.g.,* how she navigates within the app), by either prefetching resources that will not be used, or limiting their prefetching algorithms to a set of "safe" situations (hence limiting their potential), (ii) they *do not change as users' interaction patterns change*, and (iii) they rely on *approximated static analysis techniques* for identifying when and which network resources can be prefetched, thus potentially leading to unhandled control of data flow paths (*e.g.,* in case of reflection) or to the identification of paths that are infeasible at run-time (*i.e.,* false positives) [5].

With the aim of addressing the limitations above, in this paper we present our first investigation into a *navigation-aware technique for personalized prefetching of network requests of Android apps*. The proposed technique is called NAPPA and it is fully automated (with the possibility of custom behaviour provided by developers), transparent w.r.t. the back-end of the app (*i.e.,* it is independent from the data types provided by the back-end and it does not require any modifications in the business logic of the back-end), and adapts its prefetching behaviour according to the navigation patterns of the user.

NAPPA revolves around the concept of *Extended Navigation Graph* (ENG), where nodes represent Android activities in the app, edges represent navigations among activities, and each edge is annotated with information about the intent used for the navigation and the probability of being performed by the user. The intuition behind NAPPA is to build and keep updated the ENG of the app at run-time and to prefetch network requests according to the paths that will be most likely travelled by the user according to the current status of the ENG. As such, our prefetching mechanism is *personalized* since every app installation has its own ENG with different transitions and weights according to each user's unique navigation patterns.

Given the source code of an Android app, NAPPA acts in two phases: (i) at development time NAPPA automatically extracts all activities of the app and instruments it in order to continuously probe user navigation events and to inject the business logic for performing network prefetching, and (ii) at run-time NAPPA builds and keeps the ENG up to date according to user navigation events, prefetches network requests according to the current status of the ENG, and intercepts the app's network requests for serving prefetched resources, instead of using the network. We opted for building the ENG at run-time in order to ensure that its edges represent valid navigation transitions, as opposed to building it via static analysis, which may lead to incomplete ENGs due to well-known challenges such as the management of the implicit control/data flow among Android components, user-generated events, reflection, and multi-threading [5].

To the best of our knowledge, we are the first to propose an approach for prefetching network requests of Android apps that (i) works at a high level of abstraction (*i.e.,* the navigation of the user within the app), (ii) adapts to each individual user navigation patterns, and (iii) does not inherit the limitations of current static analysis techniques.

Paper outline. Sections II and Section III present background concepts and NAPPA, respectively. Sections IV and V discuss the next steps of this research and related work, respectively. Section VI closes the paper.

II. Background

A. User Navigation in Android Apps

According to the Android programming model [6], Android activities are the main building blocks of the app and represent single screens of the UI. Activities are in charge of (i) reacting to user events (*e.g.,* a touch on the screen), (ii) executing some specific functionality (possibly with the help of other app's components like services or content providers), and (iii) updating the user interface of the app for providing information to the user. An Android app comprises multiple activities to provide a cohesive user experience. When a user navigates within an app, the source activity starts a new instance of the target activity by creating an intent and passing it to the startActivity() method [7]. In this context, an Android intent is composed of two main parts [7]: (i) the *activity* to start and (ii) the *extras* representing the additional information required to perform the action, defined as a set of key-value pairs.

B. Network Requests in Android Apps

The majority of network-connected Android apps request and receive data from their remote backends in a RESTful fashion via the HTTP protocol [8], [9]. Android developers can choose among a plethora of HTTP client libraries for their apps, such as *OkHttp*[1], *Volley*[2], *Retrofit*[3]. Android HTTP clients provide features such as connection pooling, concurrent requests, socket sharing, etc. In this study we focus on *OkHttp* since it is the HTTP client providing the official implementation of the *HttpUrlConnection* interface from Android 4.4 and it is at the basis of the most widely used networking libraries for Android, such as Volley, Picasso, and Retrofit [10].

C. The Prefetching Opportunity

If we consider activities as nodes and explicit intent launches as transitions among nodes, we can obtain the navigation graph of an Android app. If we enrich this graph with (i) information about the probabilities of triggering each transition, (ii) the URLs requested by every activity, and (iii) how intent extras can map to dynamic fragments of the requested URLs, then we obtain the Extended Navigation Graph (ENG). NAPPA exploits the ENG at run-time in order to (i) resolve as soon as possible the URLs to prefetch and (ii) prefetch only the network requests belonging to the activities that will be most *likely* visited by the user according to her own navigation patterns exhibited during previous usage sessions of the app. This reasoning leads to the navigation-aware and personalized prefetching of network requests for Android apps.

III. The Approach

We designed NAPPA as a two-phased technique: the first phase is performed only once at development time and the second one is carried out at run-time throughout the execution of the app.

[1]http://square.github.io/okhttp
[2]https://developer.android.com/training/volley
[3]https://square.github.io/retrofit/

A. NAPPA at development time

As shown in Figure 1, the only input required by NAPPA is **the original source code of the app** being developed. NAPPA does not impose any specific development style to the developer, provided that the app makes HTTP requests by passing through *OkHttp*.

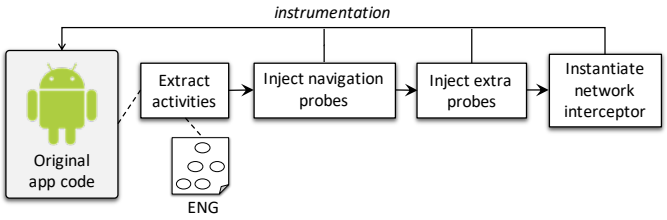

Fig. 1: Overview of NAPPA at development time

The first step of NAPPA is the **extraction of all activities** of the app in order to build the initial version of the ENG. This step is realized by (i) parsing the AndroidManifest.xml file of the app, (ii) producing a node in the ENG for each detected activity and (iii) labelling each node with the full path of the Java class implementing the activity. Since it is mandatory to declare all activities of an Android app in its manifest file[4], the produced ENG is complete with respect to the coverage of all activities of the app being analysed.

Then, NAPPA **injects a navigation probe** in the Java class corresponding to each extracted activity. The navigation probe notifies NAPPA when a navigation occurs from/to every activity of the app at run-time. This step is realized by injecting notification statements (*i.e.,* probes) in the body of the onStart and onStop methods of each previously extracted activity. Since those methods are called by the Android OS every time it needs to make the current activity visible or not visible to the user, NAPPA is able to correctly and timely detect any transition within the ENG of the app.

The next step consists in the **injection of extra probes**, which will be in charge of notifying NAPPA every time an intent's extra is set by the app. This step is realized by injecting notification statements immediately after a call to either the putExtra or putExtras methods in each activity (they are the only methods in the Intent class where its extra fields can be changed).

Finally, NAPPA **instantiates a network interceptor** in order to log all HTTP requests issued by the app and serve prefetched resources. At run-time the activities performed by the network interceptor are totally transparent to the developer.

Summarizing, at the end of the steps described above, NAPPA has (i) extracted the ENG of the app containing all activities of the app, (ii) instrumented the app with probes for notifying about navigation and intents' extra update events at run-time, and (iii) added a transparent network interceptor for logging all outgoing HTTP requests of the app at run-time and serving prefetched resources. All those steps are

[4]https://developer.android.com/guide/topics/manifest/activity-element

performed *automatically* without requiring any intervention by the developer.

B. NAPPA at run-time

Figure 2 shows the main components of NAPPA at run-time. All together they are responsible for (i) keeping the ENG always up-to-date according to the navigations and actions performed by the user, (ii) identifying which network resources can be prefetched, (iii) prefetching network resources, and (iv) making prefetched resources available to the app via a lightweight URL map. In the following the behaviour and main responsibilities of these components of NAPPA are presented.

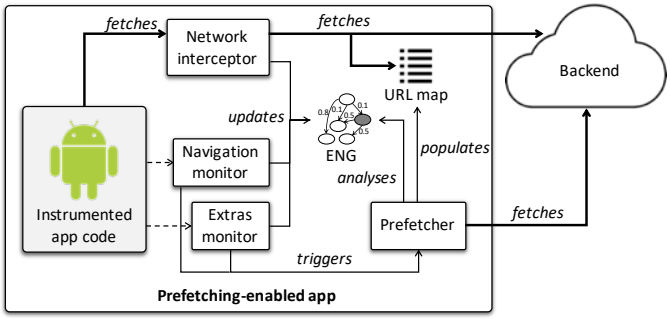

Fig. 2: Overview of NAPPA at run-time

The **Navigation monitor** is a passive component, which is triggered every time a navigation probe in the app raises a navigation event (*i.e.,* every time the user is moving between two screens of the app). The main responsibilities of the navigation monitor are: (i) to keep track of the current activity within the ENG, (ii) to add a transition in the ENG if a raised navigation event is involving a target activity that has never been visited before, (iii) to update the weight on the transitions of the ENG for every received navigation event, and (iv) to trigger the Prefetcher component at every navigation event.

The **Extras monitor** is also a passive component and it is triggered whenever a previously-injected extra probe raises an event related to the setting of one of the intent's extra fields. The main responsibilities of the extra monitor are: (i) to update the ENG with the newly set extra field, and (ii) to trigger the Prefetcher component for every received event.

At the core of NAPPA lies the **Prefetcher**. It is triggered by the two monitor components every time a user navigates within the app or an intent's extra is set; then, the Prefetcher analyses the current ENG and, based on the currently-prefetchable network resources, it fetches them from the backend of the app and stores them into the URL map. It is important to note that we designed NAPPA with *separation of concerns and maintainability* in mind: if a different prefetching algorithm will be needed in the future or the developer needs a custom prefetching algorithm (*e.g.,* by exploiting machine learning techniques for better predicting the user's navigation events), the developer can implement such a change simply by updating the Prefetcher component, without impacting the other components of NAPPA.

The **URL map** is a lightweight hashmap storing a set of key-value pairs. For each entry of the map, the key and value parts contain the URL and corresponding payload of the prefetched resource, respectively. The URL map is cleared every time the app goes in background or is killed in order to have a relatively low number of entries in the map and to keep only fresh data in the prefetched resources.

The **Network interceptor** has three main responsibilities: (i) to log all outgoing HTTP requests of the app at run-time, (ii) to update the ENG according to the logged HTTP requests, and (iii) to serve prefetched resources as soon as one of the outgoing requests made by the app matches an entry in the URL map. Logging HTTP requests is necessary in order to update the list of the performed network requests during the current activity in the ENG; this information will be used by the prefetching algorithm for identifying new mappings between some intent's extra field and dynamic URL fragments requested by the app. Technically, NAPPA exploits the interceptors mechanism of OkHttp, which allows us to add our network interceptor as last element of a potentially not-empty chain of already existing interceptors. This makes our prefetching mechanism totally *transparent to the developer* in terms of effort, and *unintrusive* with respect to the business logic of the app.

IV. What's next

We developed a first **prototype**[5] of the proposed technique in order to assess its feasibility. The prototype allows developers to automatically perform all the steps of the development-time phase (see Section III-A) via a dedicated IntelliJ plugin. The network interceptor component has been implemented as an Android library and it is based on *OkHttp*. The prototype also covers the run-time phase of the NAPPA, where a dedicated Android library realizes all the components discussed in Section III-B. Persistence is implemented by using the Room Persistence Library[6] and the SQLite Android native driver.

In the short term we are designing and conducting a series of **experiments** for thoroughly evaluating NAPPA. The first experiment aims at assessing the accuracy of NAPPA in identifying reachable activities within the dynamically-built ENG. We chose to evaluate the reachable activities in the ENG since the ENG is the core of the whole approach and failing to build an accurate ENG at run-time may potentially result in a high number of unused prefetched resources (false positives) or missed prefetching opportunities (false negatives). Preliminary results involving 10 3rd-party apps reveal that NAPPA is able to accurately detect navigation traces among activities w.r.t. the results of Gator [11], the state-of-the-art approach for building window transition graphs (*i.e.,* models of Android apps similar to our ENG). Moreover, in a second experiment we will assess the latency reduction Android apps can exhibit when adopting NAPPA. This experiment will be performed on a large scale and will involve the execution and

[5]https://github.com/S2-group/NAPPA
[6]https://developer.android.com/topic/libraries/architecture/room

measurement of real Android apps mined from the Google Play store. Currently, we are investigating on how to build a dataset of interaction traces of real users of Android apps, so to be able to realistically evaluate the latency reduction introduced by NAPPA. Finally, we are planning to perform a third experiment for assessing the overhead of NAPPA at run-time in terms of both performance and energy consumption.

In the long term we are planning to design and develop different **variations of the prefetching algorithm**, each of them following different strategies. For example, we will formulate prefetching as a variation of state reachability analysis with probabilities and exploit model checking at run-time for its resolution; we will map the ENG to different variations of Markov chains with probabilities; we will formulate prefetching as a graph-based combinatorial optimization problem and solve it analytically, etc. We will design and conduct large-scale experiments on the accuracy of each of the above mentioned variations involving real apps from the Google Play store and (possibly) reusing already existing benchmarks [4]. Also, the experiments will target the different advantages and disadvantages brought by each variation, *e.g.,* its performance at run-time, the resources it will consume, and its impact on the overall quality of the mobile app in terms of performance, user experience, and energy consumption.

V. Related Work

To the best of our knowledge, Bouquet [12] and PALOMA [4] are the first approaches proposed in the literature to prefetch network requests of Android Apps. Bouquet applies program analysis techniques to bundle HTTP requests in order to reduce energy consumption in mobile apps. The approach detects Sequential HTTP Requests Sessions (SHRS), in which the generation of the first request implies that the following requests will also be made, and then bundles the requests together to save energy. This can be considered a form of prefetching. This work, however, does not address inter-callback analysis and the SHRS are always in the same callback. Therefore, the prefetching only happens a few statements ahead (within milliseconds most of the time) and has no tangible effect on app execution time.

PALOMA is the first technique to apply program analysis to address *what* and *when* to prefetch certain HTTP requests in mobile apps in order to reduce user-perceived latency. With respect to PALOMA, our technique acts at development time and at run-time in order to address *when* and *what* network resources can be prefetched considering users' navigation within the app, how users interaction patterns change potentially avoiding to unhandled control/data-flow paths. PALOMA does not consider neither the user navigation through the app nor the history of past requests.

VI. Conclusions

In this paper we presented a new technique for navigation-aware and personalized prefetching of network requests in Android apps. The proposed technique works at a higher level of abstraction with respect to state-of-the-art approaches (*e.g.,* callback-based prefetching) and focusses on the so-called navigation graph of the app.

It is important to note that the same principles of NAPPA can be applied to other mobile platforms (*e.g.,* iOS), provided that it is possible to (i) build accurate ENGs at run-time and (ii) intercept and prefetch network requests independently from the business logic of the apps.

Finally, focussing on the navigation graph opens for a new family of prefetching opportunities. Firstly, the navigation graph can act as internal model for run-time prefetching algorithms, which now can look ahead *several steps* into the future network resources which will be requested by the app. Secondly, it allows us and other researchers to develop prefetching algorithms which take into account the unique and user-specific navigation patterns exhibited by each user, potentially reaching better results in terms of hit rate w.r.t. the one-size-fits-all prefetching approaches existing today.

Acknowledgment

The authors acknowledge Raffaele Roberto Laricchia and Francesco Bevilacqua - students of Polytechnic University of Bari - for the implementation of the NAPPA prototype.

References

[1] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 15–24. IEEE, 2013.

[2] Yixue Zhao, Paul Wat, Marcelo Schmitt Laser, and Nenad Medvidović. Empirically assessing opportunities for prefetching and caching in mobile apps. 2018.

[3] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, volume 12, pages 107–120, 2012.

[4] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *Internat. Conf. on Software Engineering*, 2018.

[5] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[6] *Android Application Fundamentals*, 2018. Available at https://developer.android.com/guide/components/fundamentals.

[7] *Android Intents and Intent Filters*, 2018. Available at https://developer.android.com/guide/components/intents-filters.

[8] *Android connectivity*, 2018. Available at https://developer.android.com/training/basics/network-ops/connecting.

[9] Yun Ma, Xuanzhe Liu, Yi Liu, Yunxin Liu, and Gang Huang. A tale of two fashions: An empirical study on the performance of native apps and web apps on android. *IEEE Transactions on Mobile Computing*, 17(5):990–1003, 2018.

[10] Leanid Vovk. How to choose an Android HTTP Library, 2018. Available at https://appdevelopermagazine.com/5265/2017/6/5/how-to-choose-an-android-http-library.

[11] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. *Automated Software Engineering*, Jun 2018.

[12] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th international conference on software engineering*, pages 249–260. ACM, 2016.