2019 IEEE International Conference on Web Services (ICWS)

A First Look at Instant Service Consumption with Quick Apps on Mobile Devices

Yi Liu[1], Enze Xu[1], Yun Ma[2], and *Xuanzhe Liu[1,3]

[1]Key Lab of High-Confidence Software Technology (Peking University), Ministry of Education, Beijing, China
[2]Tsinghua University, Beijing, China
[3]Peking University Information Technology Institute, Tianjin Binhai, China
Email: [1]{liuyi14, xez, liuxuanzhe}@pku.edu.cn, [2]yunma@tsinghua.edu.cn

***Abstract*—Mobile app ecosystem has gained giant success in providing services on mobile devices to facilitate almost all aspects in our daily life. However, the whole-package installation and dramatically increasing package size are now preventing users from trying more apps. To address the issue, many lightweight frameworks have emerged, enabling to provide the experience of instant service consumption where apps are of small size and no installation is needed to consuming services provided by the apps. In this paper, we conduct the first empirical study on instant service consumption on mobile devices. We focus on one of the most popular frameworks, quick apps, which are proposed and supported by nine mainstream mobile phone manufacturers in China. Quick apps are implemented with Web-based technologies, and run as native apps without the need of installation. We find that quick apps have much smaller size and only provide a limited set of services compared to their corresponding native apps. Then, we characterize the performance differences between quick apps and native apps in terms of launching time, data drain, and network connections, when the two kinds of apps provide the same services. Our observations reveal that quick apps perform better than native apps thanks to its much smaller size and less functionalities in a single page. Finally, we propose a machine learning based approach to helping developers construct the quick app from an existing native app.**

***Keywords*-*Instant Service Consumption; Quick App; Performance; Web Services;**

I. Introduction

With the prevalence of mobile devices like smartphones and tablets, mobile apps have seen widespread adoption and become the dominant consumers of Web services. A recent survey [20] reveals that both the leading app stores including Google Play and App Store have more than two million apps and contribute billions of downloads in 2017. Mobile apps are projected to generate 188.9 billion U.S. dollars in revenues in 2020.

Developers prefer to add new features to serve and attract users. However, increasing features (such as more SDKs, higher resolution images, more functionalities, etc.) result in larger app size. According to a report from Google Play [19], the average app size has quintupled since 2012. Although smart devices have more storage space than ever, user' high-quality photos, videos and other media files are occupying more and more space, which means the available space on devices is gradually tightening. Meanwhile, larger apps cost more time to download and even may result in sluggish responsiveness. A recent study [17] reveals that a full quarter of users delete an app simply because they need to free storage space on their devices. Meanwhile, the increasing size and whole-package installation negatively affect users' enthusiasm on trying new apps. Most apps fail to win the users' favor and lose the opportunity of success.

To address the issue, more and more service providers try to customize their apps with many newly-proposed instant-service frameworks to enhance the experience of accessing services. Google has announced instant apps [10] in 2016, which allow users try apps without installation. Instant apps are developed as native apps without additional skills, but have a variety of limitations. For example, they cannot use background services or notifications. Meanwhile, users have to download the full-size app once users need unsupported functionalities in the instant app. `WeChat` has been equipped with its mini program platform [21], which works in a similar way. Mini programs are essentially implemented as Web apps with some customizable features that are thus accessed via hyperlinks from the app, and may suffer from worse user experience than native apps. In 2018, Huawei, Xiaomi, OnePlus and other six top mobile phone manufacturers in China jointly launched a unified standard called **Quick Apps**. Quick apps are so small that users can directly start them within just a few seconds without installation. Quick apps have the pros of both Web apps (e.g. mini programs) and native apps, which can be easily developed with Web-based technologies, but run as native apps thanks to the underlying system-level support.

In order to investigate how service providers customize their apps to provide experience of instant service consumption, we conduct a comprehensive empirical study to characterize quick apps and analyze performance difference between quick apps and native apps. More specifically, this paper tries to answer the following three research questions:

- **RQ1: How do service providers leverage quick apps to provide instant service consumption?** The ecosystem of quick apps is so young, and users have no knowledge about them. We first collect quick apps as

*corresponding author: liuxuanzhe@pku.edu.cn many as possible to give users an intuitional knowledge of these apps.

- **RQ2: Are quick apps really quick enough to beat the native apps?** We conduct some experiments to collect the runtime information of both native apps and quick apps to check if quick apps really perform better. We find that quick apps indeed can save network traffic and launching time.
- **RQ3: How should developers transform an existing native app into a quick app?** Based on our analysis, the ecosystem of quick apps is a rising star. However, most developers have no experience to construct a quick app. It is useful to provide an approach to helping developers to select pages from the existing native app to construct a quick app.

We collect all 297 quick apps provided by Huawei App Store [2], of which 168 quick apps have corresponding native apps. In our experiment, we use two smartphones of the same model to run the native app and the quick app simultaneously in the same environment, and explore how they perform when visiting the same page. We use a proxy between the Web service providers and mobile devices to capture the HTTP and HTTPS traffic for further analysis. Meanwhile, we also record the loading time of both the native app and quick app when visiting the same page. In order to determine whether a page of a native app should be transformed into a quick-app version, we have manually labeled hundreds of pages, extracted 22 features for each selected page, and designed a machine learning based approach. Our approach can achieve 84% of precision, 92% of recall, and 88% of F-measure, which is promising for real usage.

The remainder of this paper is organized as follows. Section II introduces some basic knowledge about quick apps. Section III introduces the measurement methodology and how we conduct our experiments. Section IV presents the results and analysis of our study. Section V presents related work, and Section VI concludes this paper.

II. Background

In this section, we give some background knowledge about quick apps.

In early 2018, nine top Chinese smartphones makers, including Huawei, Xiaomi, OnePlus, etc., worked together to launch a unified standard called quick app [1], aiming to empower developers through standardization. Users can directly access these new apps without installation.

Figure 1 shows the architecture of the quick app platform. Developers can develop quick apps with front-end framework and Web-based technologies, which is easy and efficient. Developers can specify all pages in a manifest file, and construct each page with pre-defined UI components and APIs.

[2]Huawei App Store is the official app store like Google Play for Huawei devices

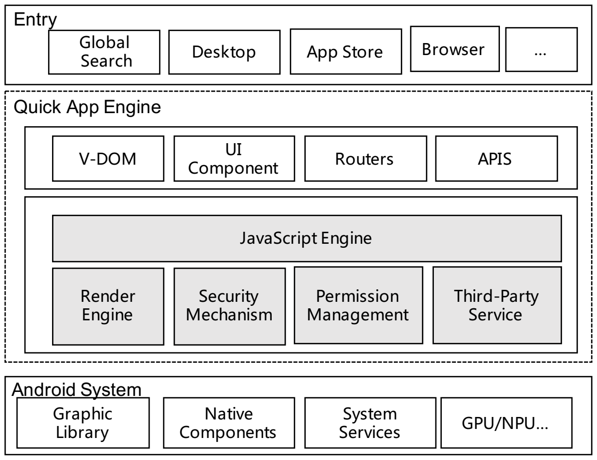

Figure 1: The Architecture of Quick Apps

The quick app engine applies virtual DOM technology [7] to improve the performance of UI updating. In fact, the render engine and JavaScript engine will map the virtual DOM of a running quick app to native components of the Android system. Such technology enables quick apps to perform much better than pure Web apps, and users even feel as they are interacting with native apps. Meanwhile, the quick app engine is rooted in system, which can realize deep integration with GPU/NPU to get better performance than other cross-platform frameworks of the application layer [8].

The quick app engine also encapsulates a lot of system services and third-party services so that quick apps can access most services to alleviate the gap between quick apps and native apps. It is worth to mention that each page of a quick app can be accessed via a deep link [23][16], which facilitates users to access services of a quick app and makes it possible to assemble various services with mashup on the mobile platform [16].

III. Measurement Methodology

The quick app ecosystem is a rising star to benefit both developers and end users. Developers can easily develop a “light” app to low the threshold for access. Meanwhile, users can try a new app without installation, and enjoy the competitive user experience as the native app. We collect the basic info and usage data of quick apps from the app store, and proactively check if quick apps can achieve the desired effect as they advocate.

A. Data Set

We choose Huawei App Store to get the published quick apps, since it holds the most number of quick apps and provides detailed description of each quick app, such as download number, ratings, etc. Huawei App Store has an entry to visit all quick apps, and can also get quick apps

Table I: Collected Quick Apps

Category	Quick App #	Ratio
Tools	64	21.55%
News & Reading	35	11.78%
Lifestyle	30	10.10%
Shopping	29	9.76%
Music & Audio	25	8.42%
Education	16	5.39%
Travel & Local	15	5.05%
Finance	13	4.38%
Game	13	4.38%
Maps & Navigation	12	4.04%
Sports & Health	12	4.04%
Child	11	3.70%
Business	7	2.36%
Communication	5	1.68%
Food	4	1.35%
Auto & Vehicles	3	1.01%
Personalization & Theme	2	0.67%
Photography	1	0.34%

by searching with the keyword "quick app". We traverse all quick apps from these two entries, and get 297 quick apps in total [3].

We then classify these quick apps into the 18 app categories in Huawei App Store based on tags of quick apps and our domain knowledge. Table I shows the number of quick apps in each category. These quick apps cover all the 18 app categories in Huawei App Store, including tools, news & reading, shopping, etc. In other words, users can satisfy their most daily requirements with quick apps instead of native apps. The category of tools contains the most number of quick apps. Among these 297 quick apps, 168 (56.57%) have corresponding native apps. Then, we crawl all related information of these quick apps and native apps, including their description, ratings, installation packages, download counts, etc.

For those apps that have both a native app and the corresponding quick app, we manually select some pages with same functionalities from both versions. We launch these pages to record loading time, snapshots of pages, and runtime page structure (XML format like HTML file of a Web page, which describes each component and its detail info, including type, position, size, etc.). We also record the network traces including the number of connections, response time, traffic data drain, etc., when visiting pages.

B. The Measurement Testbed

In this part, we introduce the infrastructural platforms used during data collection, and how we conduct our experiments.

We use Nexus 6 (3GB RAM, 32GB ROM, Quad-core 2.7 GHz) smartphone running Android 7.1.1 as our test devices.

[3]We collected these quick apps on December 5, 2018

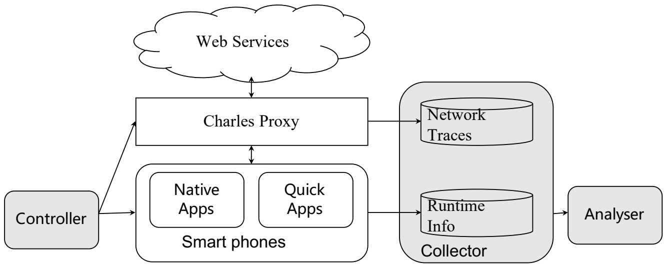

Figure 2: The work flow of data collection and analysis

We deploy native apps and quick apps on them, respectively, and open the corresponding pages under the same experimental condition. The Android devices are rooted so that we could prevent other apps from connecting network so as to reduce noise. Both smartphones connect to the same Wi-Fi to keep the same network condition. We install the runtime support environment of quick apps on one smartphone to run quick apps, and install the native app on the other one. Meanwhile, we set a proxy between the providers of Web services and smartphones with Charles [4] to intercept all HTTP and HTTPS network traces for further analysis of how quick apps and native apps consume Web services.

Figure 2 shows the work flow of how we conduct our experiments. There are three key components to collect and analyze data:

- **Controller.** Controller is responsive for automating the data collection processes. We use android debug bridge (ADB) to automate operations on native apps and quick apps. Meanwhile, controller will send commands to start/stop recording of the network requests with Charles proxy when apps are running.
- **Collector.** Collector is responsive for collecting runtime info and network traces when running quick apps and native apps. The runtime info includes loading time when visiting a page, a snapshot of current page, and page structure. The page structure is exported as an XML file like a HTML file of Web pages, in which each node describes the component type (e.g. TextView or ImageView), size, position, etc. Meanwhile, the collector will send a command to the Charles proxy to save these recorded network traces once the current test finished.
- **Analyzer.** The analyzer is responsive for filtering and analyzing collected data. First, we filter out those abnormal data. For example, some quick apps throw exceptions and exit. Then, the analyzer will conduct some analysis to compare the differences between native apps and quick apps. We will detailedly discuss these results later.

C. Definition

In order to better describe our study, we define some terms as following:

- **Component**. A user interface UI of a native app or a quick app consists of several components, and each component has a set of attributes, formally, $UI = \{comp_1, comp_2, ..., comp_n\}, comp_i = \{att_{i,1}, att_{i,2}, ..., att_{i,k}\}$. These attributes include component type, size, position, content, status, etc.
- **Feature**. "Text content" usually describes or implies functional information of a component [5]. We use texts appeared in a page to represent its features. We regard the text content of a component as a feature as if it satisfies: 1) text labels are immutable; 2) the length of text is no more than 4 (developers always use short phrase to represent a functionality [14]).
- **Page**. We take the set of components in an Android activity as a page at runtime. Actually, each page defined in a quick app will be hosted and displayed in a pre-defined activity by quick app engine at runtime. We can export the page as an XML file, in which each element corresponds to a component. The rendering result of a page can be saved as a snapshot (a png format image).
- **Trace**. A trace is a sequence of HTTP/HTTPS requests and responses of Web services when we visit a page with a native app or a quick app.
- **Number of connections**. The number of connections is the number of pairs of requests and respective responses in a trace. It can reflect how many services a page integrate.
- **Traffic volume**. The traffic volume means the total network traffic of a trace, including the bytes sent and bytes received.
- **Launching Time**. The launching time is the time spent on launching a native app or a quick app. It reflects the responsiveness of an app.

IV. Data Analysis

In this section, we analyze the collected data to answer three research questions, respectively.

A. The Quick Apps Ecosystem

In this part, we give a statistical analysis of collected data to give users a straightforward perception of quick apps.

Figure 3 shows the distribution of package size of quick apps and native apps with a box plot. In order to better exhibit, sizes are in logarithmic scale. We can find that quick apps have a much smaller size than native apps. In the median case, the size of a quick app is only 274.38 KB, but the size of a native app is 27,235 KB. The size of a native app is roughly 100 times larger than the size of a native app in the media case. Meanwhile, the size of 80% of quick apps we collected is no more than 500 KB, and the largest size of collected quick apps is only 1,736 KB. To make a comparison with Web apps, a recent study of *httparchive.org* [3] reports that the median size of mobile

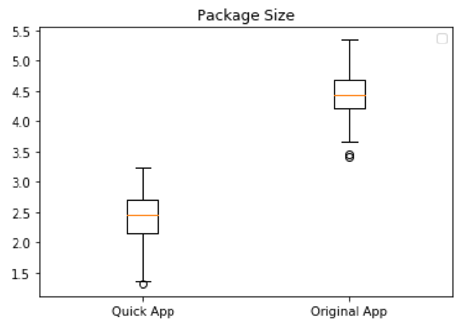

Figure 3: Package Size of Quick Apps and Native Apps

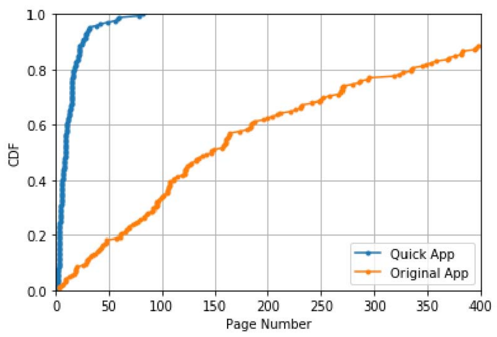

Figure 4: # Page of Quick Apps and Native Apps

Web pages in 2018 was 1,285 KB (4.68 times larger than a quick app). In other words, a quick app is much smaller than the mobile Web page and native app, and users can quickly access a quick app for the first time.

Figure 4 shows the cumulative distribution of page number of both quick apps and native apps. We extract the pages from the manifest files in installation packages, which contain predefined pages in both quick apps (named 'Page') and native apps (named 'Activity'). We can find that quick apps contain much less pages than native apps. In the median case, the quick app contains only 9 pages, while the native app contains 147 pages. The quick app has 82 pages at most, while the native app has 1,056 pages at most. Less pages can reduce the package size effectively so that users can access the quick app as quickly as possible, and occupy less local storage space. However, users have to download the full-size app if they desired features are not included in the small set of pages of a quick app. Although developers can add more features in the quick app with plenty of development efforts, it will increase the download size and affect the instant experience.

Figure 5 shows the cumulative distribution of ratings of both quick apps and native apps. In Huawei App Store, users can give a rating for an app from 1 to 5. In the median case, the score of the quick app is 2.5, but the score of the native app is 3.5. We manually scan the comments of quick apps, and find that users mostly complain the compatibility and unresponsiveness of quick apps. Figure 6 shows the cumulative distribution of growth rate of quick apps within

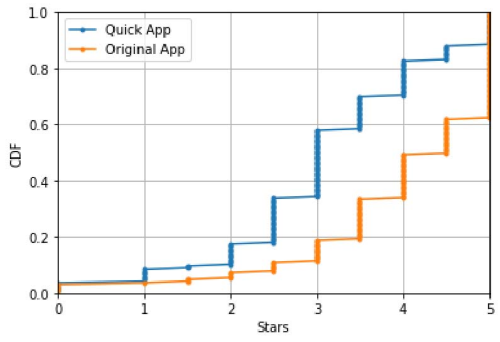

Figure 5: Ratings of Quick Apps and Native Apps

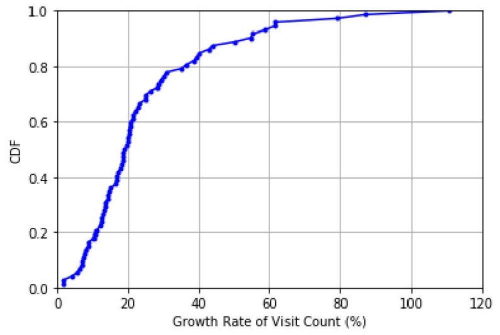

Figure 6: The Growth Rate of Quick Apps Within One Month

one month. We find that 40% of quick apps have a growth rate over 20%, which shows the great vitality of the new ecosystem. Although the new ecosystem is really young and has some shortcomings, quick apps can win more and more users since their small sizes and high performance once the robustness of system-level support is guaranteed.

B. *Performance of Quick Apps*

Following the experiment setup described in the previous section, this section focuses on the runtime performance evaluation of native apps and quick apps. We focus on performance of consuming Web services and responsiveness when launching native apps and quick apps. We wonder if quick apps and native apps have different behavior when consuming Web services, and if quick apps really perform better than native apps.

Figure 7 shows the distribution of traffic size when launching quick apps and native apps, respectively. In the median case, a quick app only downloads 131.91 KB data, but a native app downloads 1812.78 KB data. The latter one is about 13.74 times larger than the former one.

Figure 8 shows the distribution of numbers of pairs of requests and responses when launching quick apps and native apps, respectively. We can see that a native app initiates much more requests, which is about 7.5 times more than a quick app in the median case (75 requests & responses for a native app VS 10 requests & responses for a quick app). We also find that Web services consumed by both

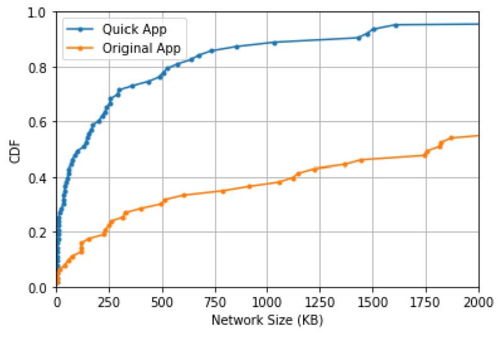

Figure 7: The Traffic Size of Launching Quick Apps and Native Apps

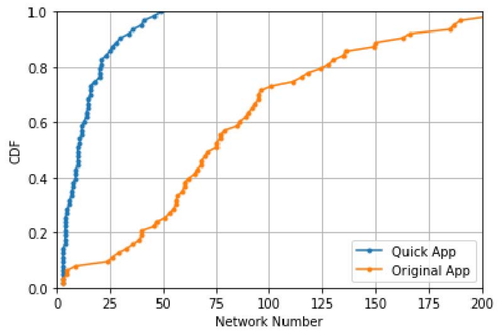

Figure 8: The Request # of Launching Quick Apps and Native Apps

native apps and quick apps are all RESTful-fashion, which is lightweight and flexible. Most requests and responses are based on HTTP/HTTPS protocol, but we find that some native apps apply WebSocket protocol [22], which can facilitate real-time data transfer from and to the server with lower overheads. Meanwhile, we also find that both native apps and quick apps have applied HTTP/2 protocol to improve their performance of loading data.

We also find that developers often use different URLs of RESTful Web services on native apps and quick apps, given the same functionality. In fact, a quick app displays less content than the corresponding native app, and the redundant response of a Web service will [15] waste users' limited data plan. It is better to provide a succinct Web service for quick apps.

The quick app mainly sends requests to fetch displayed contents and images, but the native app sends extra requests to check for updating, prefetch ads, upload data for analysis, and so on. Meanwhile, the homepage of a native app trends to display more contents, which may result in more requests. Figure 9 also shows the distribution of numbers of connected domains when launching native apps and quick apps, respectively. We can find that native apps connect much more domains. In the median case, a native app connects 12 domains, but a quick app connects only 3 domains. In other words, the native app integrates more third-party Web services, but the quick app is concentrated on its own

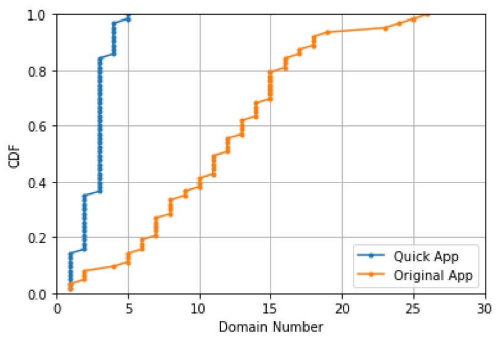

Figure 9: The Connected Domain # of Launching Quick Apps and Native Apps

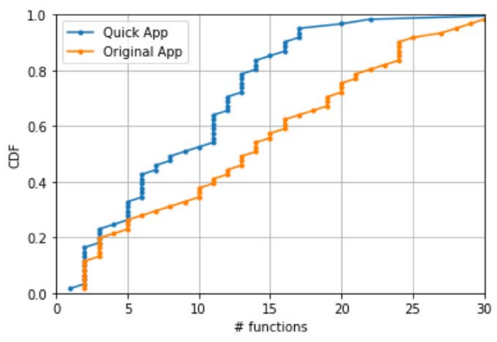

Figure 11: The Feature # in Homepage of Quick Apps and Native Apps

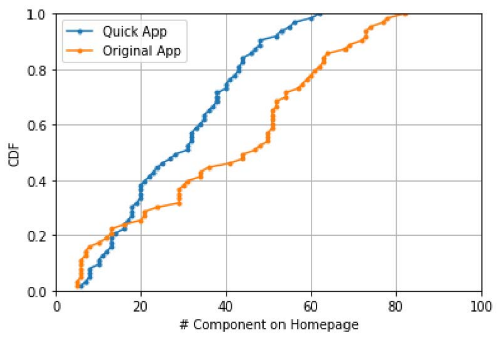

Figure 10: The Component # in Homepage of Quick Apps and Native Apps

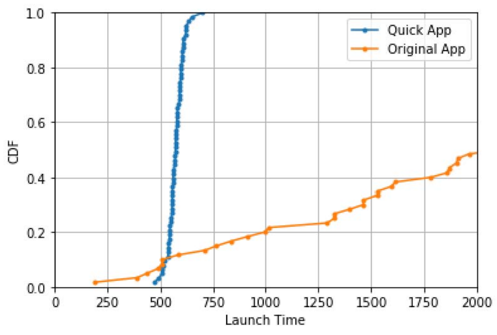

Figure 12: The Launching Time of Quick Apps and Native Apps

contents.

Figure 10 shows the distribution of number of nodes in homepages for quick apps and native apps, respectively. Both quick apps and native apps render their page like DOM tree of a Web page, and we can get the rendering result via ADB tool. We find that most quick apps contain less nodes compared to their corresponding native apps, which means native apps have more complicated page structure and display more contents. In the median case, a quick app has 31 components and a native app has 47 components.

Words on the UI components can describe the features well [5], we also compare the difference of features between the native app and quick app as shown in Figure 11. In the median case, a quick app has only 9 features in its page, but a native app has 14 features in its page. Developers prefer to put less features in quick apps so that the app can fetch content and display them as quick as possible to optimize users' experience.

Figure 12 shows the distribution of launching time of quick apps and native apps, respectively. In the median case, we need to wait 571 ms to open a quick app, but 2,255 ms to open a native app. We find that most native apps cost much more time to launch, which is about 3.9 times longer than quick apps in the median case. In fact, native apps often initiate many third-party libraries at launching time, which will occupy more memory and increase the time spent on looking for classes that are randomly distributed in `dex` files (bytecode of Android apps). For those native apps that have less launching time, we find that they are clean with only a few of third-party libraries and do not start many services at launching time. Such finding tells us that we can also efficiently optimize native apps by activating less launching processes.

C. Constructing Quick Apps from Native Apps

In our previous analysis, we have an intuitive cognition that quick apps are more concise and display less content. We wonder how should developers choose pages to construct a quick app from an existing native app. We manually select 300 of pages from quick apps and corresponding native apps, and 50 native pages that are not implemented in the corresponding quick apps. These pages cover a wide variety of areas, including news, social network, shopping, etc.

We propose a machine learning based approach to help developers to choose pages from a native app to implement the quick app. We apply the Support Vector Machine (SVM) classifier model to predict if a page should be migrated to the quick app. We have evaluated a number of alternative modeling techniques, including regression [26], K-Nearest neighbor [25], decision tree [24], etc. We chose SVM because it gives the best performance, and can model both linear and non-linear problems.

Table II: Selected Features

# Component Tag	TextView, ImageView, ListView, RecyclerView, GridView, EditText, Button, ImageButton,WebView,CheckBox
# Component Attr	checkable, clickable, focusable, long-clickable, scrollable, content-desc, resource-id
Activity Info	exported, deep-linking-enabled
Other Info	Page tree depth, #Page components, #Page features

Our predictor is based on a number of features extracted from native pages, including runtime page structure, feature number, activity information. These are chosen based on our intuitions of what factors can affect developers' choice when constructing quick apps. The runtime page structure (e.g. the number of UI components, depth of the page tree, and component tags) determines the complexity and layout of the page; The feature number represents the actual functionalities in the page. Developers may prefer to transform concise pages. The activity information (e.g. whether is exported and can be visited directly via deep linking [23]) can reflect the importance of a page. Such deep-linking-enabled pages can be widely accessed, and are always the key pages in an app.

We can get hundreds of raw features to train our proposed model from native pages. We try to reduce the number of features by removing features that have few or redundant information. For example, we have removed features of component tags or attributes that have little impact on page selections such as *LinearLayout*, *RelativeLayout*, etc. We have also constructed a correlation coefficient matrix to quantify the correlation among features to remove redundant features. We removed features that have a correlation coefficient greater than a threshold (+/- 0.8) to any of chosen features. For instance, the component attributes *checkable* and *checked* often appear as pairs. Finally, we get 22 representative features in total as shown in table II.

To perform the prediction task, we randomly divide the data set into two subsets, i.e., a training set with 297 pages, a test set with 53 pages. We evaluate our approach with three metrics, including precision, recall, and F-measure. We can achieve 84% of precision, 92% of recall, and 88% of F-measure, which is promising to help developers to determine wether migrate a page of the existing native app to the quick app.

D. Threats to Validity

Limited dataset. Quick app platform is really young ecosystem, so we can only get limited set of quick apps. However, we are the first to have a deep analysis on how quick apps perform, and the differences compared to native apps. We find that quick apps benefit from their concise page structure and less service requests. Meanwhile, our proposed approach demonstrates the potentiality of mining existing repository of quick apps and native apps to help developers to construct a new quick app.

Consistency of Functionalities. In order to make a fair comparison between a native app and its corresponding quick app, we select corresponding pages with same functionalities. However, it is impossible to get two pages with exactly same functionalities. There exists some differences between the corresponding pages more or less. Actually, quick apps benefit from less functionalities in a single page, hence they can provide better performance.

E. Discussion

Privacy concern. We find that users are not aware of permissions requested by a quick app unless they run them. The Huawei App Store should provide such info as native apps to give users a straightforward understanding of the privacy risk before they try them.

Inspiration. Although we just analyze quick apps to understand instant service consumption, our proposed methodology and model can also directly apply to other customized apps. We also plan to make a comprehensive comparison among more customized apps, including mini programs, Web apps, quick apps, etc. Meanwhile, we can further dig out more knowledge by mining existing repositories of various customized apps.

V. Related Work

Platforms for Instant Service Integration. There also exist other platforms to help service providers to integrate instant service, such as React Native [8] from Facebook, Weex [2] from Alibaba, Mini Program [21] from Tencent. All of them enable developers to easily construct an app with Web-based technologies. However, mini programs are actually Web apps, and have limited capacity and worse performance than native apps. The former two platforms also apply native rendering, but are integrated in the application layer. Nevertheless, the quick app engine is integrated at system-level, and can take full advantages of low-level capacity of hardwares and system.

Cross-platform Analysis. Existing studies focus on comparing differences between Web apps and native apps. Liu et al. [15] analyzed the performance of consuming same Web services using Web apps and native apps, respectively. Lee et al. [12], Leung et al. [13], and Papadopoulos et al. [18] focused on comparing the privacy implications of consuming Web services with Web apps and native apps. We focus on comparing performance analysis between quick apps and native apps. There also exist some work focused on identifying cross-platform features for Web apps [6][11][28] and for native apps [9][27]. Choudhary et al. [6] identified the difference of features by analyzing the network traces, but others focused on the visual and structural similarity. In this paper, we focus on comparing the features between quick apps and native apps, and we get features by extracting text contents from pages.

VI. Conclusion

In this paper, we conduct a comprehensive empirical study to understand instant service consumption on a newly-proposed platform called quick apps. We reveal the characteristics of how quick apps consume Web services and how they perform compared with their corresponding native apps. Then, we propose a machine-learning based approach to helping developers to transform a native app into a quick app with high precision (84%).

Acknowledgements

This work was supported by the National Key Research and Development Program under Grant No. 2018YFB1004403 and Peking University Information Technology Institute, Tianjin Binhai, China.

References

[1] Apps solidify leadership six years into the mobile revolution. http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution#.VMdCU7H9PeM/.

[2] alibaba. A framework for building high-performance mobile applications with a modern web development experience. https://weex.incubator.apache.org//, 2018.

[3] H. Archive. The http archive tracks how the web is built. http://httparchive.org/, 2018.

[4] Charles. Web debugging proxy application. https://www.charlesproxy.com/, 2018.

[5] X. Chen, Q. Zou, B. Fan, Z. Zheng, and X. Luo. Recommending software features for mobile applications based on user interface comparison. In *Requirements Engineering*, pages 1–15. Springer, 2018.

[6] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: accurate identification of cross-browser issues in web applications. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 702–711, 2013.

[7] V. Dom. Virtual dom. https://en.wikipedia.org/wiki/React_(JavaScript_library)#Virtual_DOM, 2018.

[8] Facebook. Build native mobile apps using javascript and react. https://facebook.github.io/react-native/, 2018.

[9] M. Fazzini and A. Orso. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 308–318, 2017.

[10] Google. Native android apps, without the installation. https://developer.android.com/topic/google-play-instant/, 2018.

[11] M. He, G. Wu, H. Tang, W. Chen, J. Wei, H. Zhong, and T. Huang. X-check: A novel cross-browser testing service based on record/replay. In *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 123–130, 2016.

[12] J. Lee, H. Kim, J. Park, I. Shin, and S. Son. Pride and prejudice in progressive web apps: Abusing native app-like features in web applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1731–1746, 2018.

[13] C. Leung, J. Ren, D. R. Choffnes, and C. Wilson. Should you use the app for that?: Comparing the privacy implications of app- and web-based online services. In *Proceedings of the 2016 ACM on Internet Measurement Conference, IMC 2016, Santa Monica, CA, USA, November 14-16, 2016*, pages 365–372, 2016.

[14] T. J. Li and O. Riva. Kite: Building conversational bots from mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2018, Munich, Germany, June 10-15, 2018*, pages 96–109, 2018.

[15] Y. Liu, X. Liu, Y. Ma, Y. Liu, Z. Zheng, G. Huang, and M. B. Blake. Characterizing restful web services usage on smartphones: A tale of native apps and web apps. In *2015 IEEE International Conference on Web Services, ICWS 2015, New York, NY, USA, June 27 - July 2, 2015*, pages 337–344, 2015.

[16] Y. Ma, Z. Hu, Y. Liu, T. Xie, and X. Liu. Aladdin: Automating release of deep-link apis on android. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1469–1478, 2018.

[17] T. Manifest. Mobile app usage statistics 2018. https://themanifest.com/app-development/mobile-app-usage-statistics-2018, 2018.

[18] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos. The long-standing privacy debate: Mobile websites vs mobile apps. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 153–162, 2017.

[19] G. Play. Shrinking apks, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2, 2017.

[20] Statista. Number of apps available in leading app stores as of 3rd quarter 2018. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2018.

[21] WeChat. The mini programs provided by wechat. http://tencent.com/en-us/articles/15000551479986174.pdf, 2016.

[22] Wikipedia. Websocket. https://en.wikipedia.org/wiki/WebSocket/, 2011.

[23] Wikipedia. Mobile deep linking. https://en.wikipedia.org/wiki/Mobile_deep_linking/, 2017.

[24] Wikipedia. Decision tree. https://en.wikipedia.org/wiki/Decision_tree, 2019.

[25] Wikipedia. K-means clustering. https://en.wikipedia.org/wiki/K-means_clustering, 2019.

[26] Wikipedia. Linear regression. https://en.wikipedia.org/wiki/Linear_regression, 2019.

[27] G. Wu, Y. Cao, W. Chen, J. Wei, H. Zhong, and T. Huang. Appcheck: A crowdsourced testing service for android applications. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 253–260, 2017.

[28] Z. Xu and J. Miller. Cross-browser differences detection based on an empirical metric for web page visual similarity. *ACM Trans. Internet Techn.*, 18(3):34:1–34:23, 2018.