AppFlux: Taming App Delivery via Streaming

Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilowska, Karsten Schwan, Adam Allred
Georgia Institute of Technology

Abstract

The number of apps downloaded for smart devices has surpassed 80 billion, with trends suggesting continued substantial increases. The resulting volume of app installs and updates puts pressure on the existing app-delivery infrastructure due to interactions of end user devices with app stores via the Internet that involve app stores' data center, content delivery network's (CDNs) points of presence and the Internet Service Provider (ISP) pipes. This paper presents 'AppFlux' – a novel app streaming approach to app delivery which reduces the load app delivery poses on the app infrastructure, and relieves users from having to deal with unnecessary updates potentially saving bytes on their costly data plan. By leveraging the emerging 'edge cloud' tier of the Internet, the AppFlux approach provides 'just-in-time' delivery of apps and their updates while (i) reducing the traffic due to app installs and updates seen in *the last mile of Internet* by up to 70%, (ii) facilitating twice as fast app delivery compared to CDNs and (iii) App streaming can potentially lead to 20% improvements in the app load times on devices. With its implementation for the Android app ecosystem, AppFlux achieves this in a completely invisible manner i.e., without requiring any changes by app developers or any explicit actions by end users.

1 Introduction

The number of active smart phones worldwide has surpassed 1 billion, and is forecast to cross the 3 billion mark by 2018. In addition, tablet sales are expected to reach 300 million by 2016, and wearable devices like smart watches, health monitoring devices, etc., are all showing signs of rapid growth. Coupled with these tremendous increases in the number of active devices is an explosion in the apps available to end users, with roughly 80 billion app downloads reported for 2013, set to reach a staggering 200 billion by 2017 [1].

App installs and more so, app updates, which our studies have found to exceed installs by more than a factor of 3x (see Figure 2(i)), affect users both directly, by potentially consuming costly bytes in their data plans and indirectly, by congesting *the last mile* of the Internet known to be a bottleneck for delivered service quality [10]. The current app delivery model, where apps are delivered directly from app stores, ignores such issues, which suggests that a conservative approach to rolling out app updates would lessen burdens on end users. Unfortunately, the fast paced nature of today's app market mandates frequent app updates to retain users, by offering new features, changing app aesthetics, enhancing usability, dealing with bugs, improving security and performance. The consequences are undesirable increases in data plan costs for end users and increased congestion in the last mile, even in the presence of Content Delivery Networks (CDNs) to ease pressure on back haul bandwidth.

The *AppFlux* approach presented in this paper and implemented for the Android app ecosystem is an additional path for app delivery. It offers an alternative way to handling the delivery of apps and their updates via app streaming. AppFlux offers novel system support in Android for *streaming* apps, which can eliminate the need for explicit installs and enable *just-in-time* delivery of apps and usage based app updates. AppFlux exploits an ongoing evolution (§2) in how mobile end devices access and interact with the Internet, via the emerging 'edge cloud' tier situated near end-users, beyond the conventional 'the last mile'. Mobile devices are increasingly posed to interact with 'nearby' rapidly accessible machines stationed in local environments in place of and/or in addition to interacting directly with the remote cloud. Examples of such machines or 'edge boxes', referred to as *eBoxes* in the remainder of this paper, include small cells, WiFi routers, or cloudlet servers [29], that could

[1] Data Source: http://mobithinking.com/

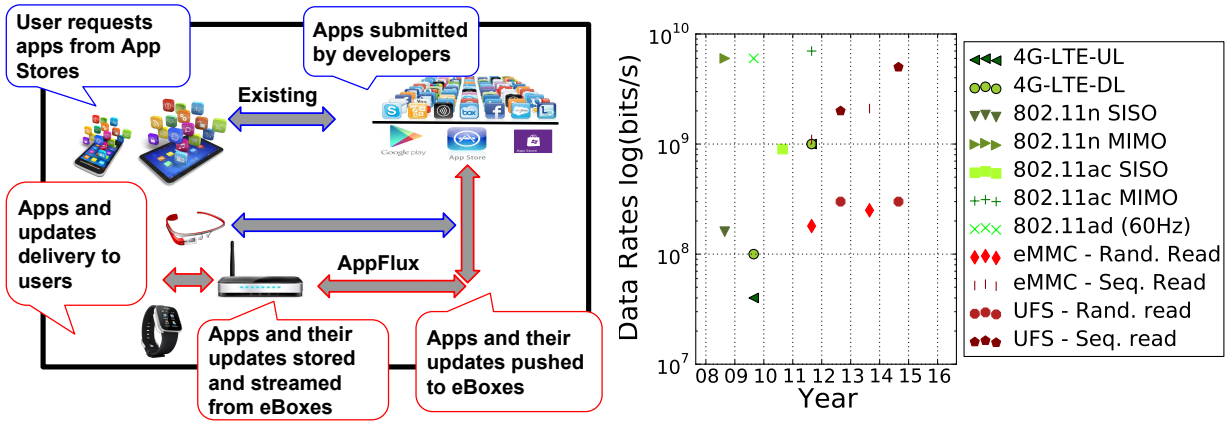

Figure 1: (i) showing app ecosystem with AppFlux; (ii) showing comparison of projected data rates between cellular, WLAN technologies and flash memories available for mobile devices. Sources: (a) ISSCC Trends 2013 (b) JEDEC Mobile Memory Technology Road map 2013.

be located in homes or neighborhoods, stores, malls, offices, etc. Their viability is evident from new hardware developments at leading companies [16, 17] and deployment plans for WiFi routers and/or small cells designed for shared access by larger numbers of end devices by service providers [4, 5, 13], and in recent research exploring opportunities for new edge services [24, 28, 29, 22].

AppFlux uses eBoxes to cache apps and app updates on behalf of end users to reduce last mile traffic volume by leveraging (i) their better connectivity to app stores and end user devices, and (ii) the redundancy inherent in app traffic. Gains are derived from the fact that on the eBox, an app can be updated frequently, without requiring any action from a mobile device. At the same time, with a nearby eBox, the device always sees the newest version of the app, but triggers an install on mobile device only when the app is actually being run, rather than when updates are being pushed out by app stores. Hence, end devices or users are not burdened by update handling.

AppFlux complements the current app delivery model, as shown in Figure 1(i), in ways that can remain invisible to end users and app developers. AppFlux relies on eBoxes to interact with remote clouds, to obtain and cache app updates when appropriate, and to then 'stream' the newest versions of apps to devices, whenever the apps are run. If no eBox is available, the device simply runs the current app version obtained earlier and stored on its local storage. If an explicit update is requested, it happens in the same manner as existing app updates. By thus replicating app updates on eBoxes, we limit update frequencies to those based on app usage rather than app presence on devices, resulting in improvements visible to end users depending on eBox availability.

AppFlux's implementation for Android is new, but the concept of application streaming has seen acceptance and use in other contexts [6, 11, 14, 19, 8], discussed further in § 13. AppFlux operation within the Android ecosystem is shown in Figure 1(i), which at one end, supports battery operated and resource constrained end client devices, and at the other end, is powered by cloud based app stores with practically unlimited compute and storage capabilities. In this ecosystem, eBoxes are positioned between these two ends, with superior connectivity to data-centers than that seen by end devices. eBoxes are not constrained by power, but have limited resources compared to app stores' data-centers.

AppFlux exploits the emerging 'edge cloud' tier in infrastructure and targets a highly active large scale mobile app ecosystem. This demands new system attributes and mechanisms not obvious in prevalent client/server architectures to be a practically viable solution as follows:

Invisiblity. A practically deploy-able solution targeting a large scale ecosystem like that of mobile apps must be invisible from its partners (i.e., app developers) and its customers (i.e., app users). For AppFlux, the app streaming mechanisms are confined to mobile OSes (i.e., Android), thus not requiring changes in apps, i.e, maintaining the existing interfaces/APIs and the functionality OSes expose to apps and users.

Sync Agnosticism. Existing approaches to app delivery are on-demand (for installs) or require explicit synchronization (for updates) between devices and app stores, causing inconvenience to users. To address this, AppFlux operates transparently, in the background, without requiring any explicit actions on end user devices.

Operation Duality. Leveraging new infrastructure elements like eBoxes while continuing support for existing app store based mechanisms for app delivery, requires

that lower layers of the systems software stack recognize the two operating models, and choose necessary actions accordingly. With AppFlux, client-side systems software (e.g., Dalvik) maintains dual stacks that support existing (from local storage) and AppFlux-enabled (via network) methods for access to necessary app components (e.g., classes, assets, etc.).

Prognostic mechanisms. Conventionally, mechanisms for loading applications, particularly on mobile devices, are not considered in critical path of performance because the executable assumed to be resident on local storage can be loaded with negligible overhead compared to application execution. This assumption does not hold when accessing application components over the network. A concrete example implemented in this paper is repackaging app classes based on their signatures to fit TCP payload lengths, prompting a prognostic approach to app loading.

Overall, the technical contributions of this paper are:

1. Novel app streaming: Design (§5) and implementation (§6) of app streaming for the Android app ecosystem showcasing a concrete realization of AppFlux and its ability to improve app delivery by reducing pressure on the app delivery infrastructure.

2. Unique app traffic characteristics: We measured Android app installs and updates at the network tap of our organization for 3 months period (§3). We believe that this measurement is first of its kind and potentially the only one given that all the app traffic has moved to https since then.

3. Trends in app anatomy: We analyzed popular apps (§4) spanning over a period of one year, i.e., approximately 5,000 apps which include top 100 free apps in each category from the Google Play Store to capture the trends in app evolution.

Our analysis of app traffic measurements suggests that AppFlux can reduce traffic in the last mile by up to 70% (§9) and result in 2x faster app delivery compared to conventinal CDNs (§10). We envision eBoxes usage in combination with CDNs can potentially provide lower latencies in future app access. Concerning app performance, for Android apps, we show that by using WiFi specifically 802.11ac for app class and static asset loading, AppFlux can obtain up to 20% improvements in loading app components (§11), compared to when running the app from the device's local storage that hold even in case of multiple concurrent clients (§12). The extent of these improvements are subject to the quality of the wireless connection and/or the affordability of cellular technologies. AppFlux derives its advantages from the following facts established as part of this research:

1. For a modern device and a WiFi accessible eBox, we show app streaming is not only to be feasible, but potentially faster than when loading apps from the device's local storage, subject to wireless connectivity (§4, §11).
2. Traffic measurements conducted at our site indicate that a typical app is updated multiple times over a small window of time (§3, §9).
3. Micro-benchmarks show that an eBox can concurrently serve many devices, even under the low cost constraints expected for such systems (§12).

2 Motivation

In this section, we discuss technology trends and use cases that motivated this work. Specifically, how these trends support the AppFlux idea and the novel usecases it potentially enables.

2.1 Technology supporting AppFlux

Increasingly Fast Wireless Networks

Figure 1(ii) shows the theoretical data rates of cellular and wlan (802.11x) vs. flash memory. Evident from the graph is that flash memory technology can be an order of magnitude slower than the wireless network. This indicates a strong potential performance benefit from accessing app code via the network vs. a device's local flash device. While such ideal numbers cannot be taken at face value, given practical issues in achieving those data rates, measurements performed in less dynamic environments [21] along with new adaptive methods for wireless communications [32, 30], however, demonstrate that such practical issues can be overcome. Our hypothesis is that *performance benefits will likely be derived from streaming apps via the wireless network vs. from on-device storage*.

Processing and Storage at eBoxes

Increasingly powerful platforms are being deployed in network elements like wireless routers, small cells, etc., due to reduced costs of compute resources. This implies the likely availability of spare cycles in eBoxes constructed from such devices, leading to spare capacity for tasks like app streaming proposed in this paper. EBox storage benefits from similar technology trends: (i) reduced cost of storage capacity and (ii) ubiquitous presence of file system support built into devices like routers [18, 17]. *These trends make it straightforward to adapt these elements to implement app caching in eBoxes, and for each eBox to efficiently perform tasks like app streaming for a reasonable number of clients*.

Predictable Patterns in App Usage

Mobile app usage is known to be clustered with respect to time and location. For example, email apps are often used at work locations in the morning, whereas games are more often used at home in the evening. Other user habits also contribute to making app use predictable,

highlighted by the trigger-follower feature in [40], and validated using actual user data published in [36]. Further, previous work [39] on characterizing app usage suggests the presence of a long tail in app usage on smart phones: this bolsters the case for app streaming, since it means that many of the apps installed on a device are not used frequently, yet, in the current ecosystem, they are updated nonetheless, hence burdening end users and their devices. These facts suggest that app streaming can rely on location clustering and app usage prediction to pre-populate distributed eBox caches with apps. *More importantly, the predictability in app usage can be used to warm up eBox's in-memory cache with appropriate apps so as to mitigate the effect of slower storage and limited memory on eBoxes.*

Mobile app delivery pain points

When end user requests an app from an app store, the path app traverses include the following elements of Internet fabric to reach end client device:

1. App store or back end data center which houses apps uploaded by app developers respond either by sending the app package directly or a more commonly redirect clients to their content delivery networks. Similarly, the update notifications are delivered by app stores to devices via push notifications that trigger end user devices to request the app updates which get delivered in same way as app installs. This can happen automatically or involve users' approval based on end user preferences.

2. When clients request apps or their updates, the app packages are delivered from CDN cache which is either privately owned (e.g., Google CDN) or standalone (e.g., Akamai in case of iOS apps) from their geographically distributed points of presence.

3. The app packages from CDNs are transmitted over an internet typically operated by internet service provider (ISP) or Telecommunication companies via their network pipes. It is important to note that each request for any app or its update results in separate traffic flow in ISP pipes. This is first part of what is often referred as 'the last mile'.

4. The second part of the last mile is the wireless connection (Wi-Fi or 4G) to which device is connected. The app or its update travels over wireless (Wi-Fi or 4G) connection to reach end client devices where it is stored and consumed by the end user.

Important to note here is that traditional CDNs cannot mitigate redundancy of traffic in the last mile simply because they operate on its edge facing ISPs as opposed to eBoxes which operate on the client side. In addition, the ISPs often opt to run their own CDNs to exploit redundancy in traffic passing through their pipes. Since, the apps and their updates are normally flagged as non-cachable content using HTTP headers, cookies etc., due to pay-per-download nature and issues related to intellectual property protection, these ISP CDNs are unable to mitigate the app traffic in the last mile. *The pain point in the app delivery is the congestion in the last mile of internet. eBoxes operating past ISP near end users can potentially effectively leverage redundancy in traffic due to apps and updates.*

2.2 Novel Usecases

One of the key contributions of this paper is the realization and evaluation of app streaming for mobile devices operating in the so called 'last mile' of the Internet. Also, important about app streaming, however, is its enablement of entirely new app functionality for mobile devices, described next.

1. *MTaaS Device Cloud Enabler*: With the push towards reducing the time to market for apps, to guarantee quality apps to end users, app testing prior to shipping has shifted from individual devices to device clouds, like those offered by companies like Apppurify etc., offering mobile testing-as-a-service. With app streaming enabled devices, it is straightforward to dynamically provision devices and/or virtual instances of Mobile OSes at a finer time granularity and without requiring app installation for each app. Using eBox-initiated app streaming and execution on connected devices, coupled with eBox-level observations of app behavior, AppFlux can potentially reduce the complexity and cost of device cloud setup and maintenance. In fact, a very similar approach is used to carry out experiments for evaluating AppFlux presented in this paper.

2. *Corporate App Stores*: One view of an eBox is as a local extension (or branch) of an app store. Another view of eBoxes, however, constrains their generality in terms of what apps are supported and how they are run, to suit the goals of specific organizations. An enterprise-level eBox, for instance, could provide end users with rapid access to 'corporate apps', tested for sufficient levels of security, offering features not available for public apps, designed to efficiently exploit certain device features (given that those devices may have been provided by the corporation), etc. and the access to apps can be limited to office location only. Such functionality can be supported with eBox-level user groups.

3. *Improved App Revenue*: By judiciously removing certain app components from the device and instead, keeping them on eBoxes, app streaming can help combat app cloning/repackaging and the consequent reduction in

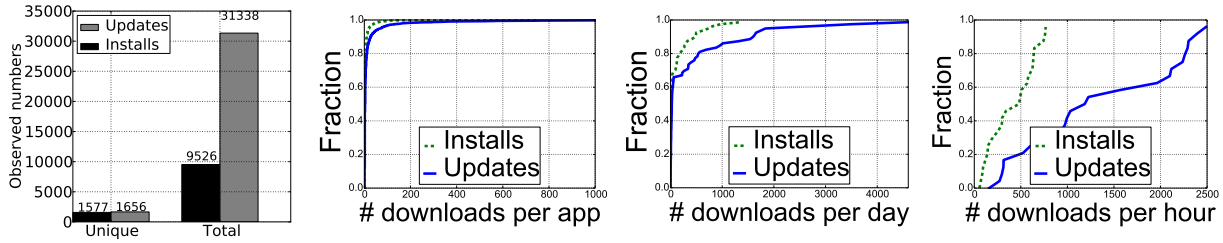

Figure 2: Android app traffic measurements over the period from May 19, 2014 to Aug 21, 2014 (excluding two weeks in July - a total of 11 weeks): (i) showing the total number of observed and unique app installs and updates ;(ii) CDF showing distribution of app installs and updates observed per unique app (iii) CDF showing distribution of app updates and installs for each day; (iv) CDF showing distribution of app installs and updates per day; (iii) CDF showing distribution of app installs and updates at a particular hour during a day.

revenue obtained by app developers [25]. Furthe,r leveraging the context of an eBox, e.g., its geographical location and the app usage profiles collected by app streaming, affords opportunities to capture end user attention for new apps and their unique capabilities, thus aiding app discovery. We plan to explore this and similar ideas in the future.

Next, we present the details about the method used to carry out app traffic data, our observations and analysis of the collected data followed by the analysis of app anatomy that motivated AppFlux design.

3 Traffic due to app delivery

We have collected data about all users in our institution i.e., Georgia Institute of Technology, over an extensive, representative time period (from May 19, 2014 to Aug 21, 2014). The data is obtained from a network tap also used for security research, and is filtered to determine whether traffic was due to the installation or updates of apps from the Google Play store and includes apps and/or updates accessed by entire population including students, faculty and staff at our institution during the collection period.

Data Collection

App installs or updates are not directly identifiable in the traffic traces available to us. Instead, we observe that whenever a device initiates an install or update of an app from the Google Play store, this leads to a HTTP 301 response code from the store, which points to the location of the app within Google's CDN or server. This 301 response contains a location URL that points to the domain "play.google.com", and contains the URL path element "/market/". The URL also contains in its parameters the name and the version number of the app being installed/updated. The version information is either a single version number if installing a new app or if an app update results in removal of old version and installation of a newer one, or a colon separated list of two version numbers, i.e., the current and new versions. This information is sufficient for determining the overall set of IP addresses for which play.google.com resolves, as many portions of Google's overall infrastructure (including app distribution) are served via their CDNs.

Prior to obtaining traces, we systematically resolved the IP for play.google.com over a multi-week period and recorded all resolved IP addresses. We configure our collection server with this IP information, to collect all packets that have any of these derived IPs in the source address section of the IP header and that utilize the TCP source port 80 (HTTP). After collecting all such traffic, we then use tshark to perform TCP packet reassembly, filtering out all traffic that does not fit the parameters of Google Play HTTP 301 responses. The resultant set of response codes represent all detectable Google Play app installs and updates for that two week period within our organization's network. While traffic collection is ongoing, we use softflowd to generate netflow information for the network. At the conclusion of the data collection process, we use nfdump to read in, aggregate, and produce total bandwidth utilization for the time period of collection. The resulting measurements report a total of 2 Terabytes of 301 requests pcaps for this period from the Google Play Store. Unfortunately, updates and installs over encrypted connections (e.g., HTTPS) cannot be detected in this fashion and are not included in the data presented because the necessary information to detect an app install or update requires the contents of the HTTP 301 response code.

Observations

1. *App updates dwarf the number of app installs.* Figure 2(i) shows that the total number of app updates is approximately three times the number of app installs. Figures 2(iii) and 3(i) show the distribution of app traffic aggregated on a per day basis. Figures 2(iv) and 3(ii) show the distribution aggregated on a particular hour during the day. One might find it surprising that the number of installs also increases on the same days when updates

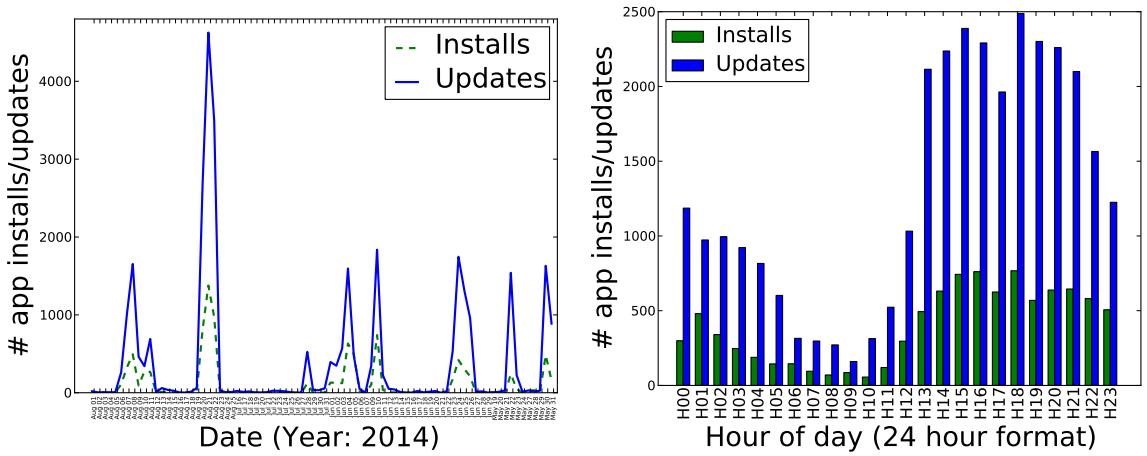

Figure 3: Android app traffic measurements over the period from May 19, 2014 to Aug 21, 2014 (excluding two weeks in July - a total of 11 weeks): (i) showing number of app installs and updates observed per day; (ii) showing number of app installs and updates observed during hours of a day.

are transmitted. This is due to the way we distinguish between updates and installs in the traffic traces added to actual installs. Also, we note that in some instances, updates are also transmitted as full apps, if the diff between the updated and installed app versions is large. In any case, it is clear that app updates introduce significant traffic flowing in the last mile for apps resident (previously installed) on devices, regardless of whether those apps are actually being used or not. The long tail in the number of unique app updates and installs seen in Figure 2(ii) (clipped on the x axis to highlight the lines) suggests a small eBox-resident app cache would suffice to provide significant savings in last mile traffic.

2. *Bursty and cyclic nature of app updates.* Looking at the temporal patterns in the collected app traffic data, Figure 3(i) shows the app traffic observed per day. App updates show a bursty nature with respect to days with burst period of 10 days. This suggests in a significant period for eBoxes to absorb updates. Variations in the absolute heights of the peaks can be attributed to the variation in the number of devices present in our organization (from May 19th through August 15th) and the arrival of students in the fall (from August 15th). App traffic in a particular hour during a 24 hour period is shown in Figure 3(ii). The diurnal traffic pattern suggests that even if updated apps have already been pushed to the app store by developers, actual updates coincide and are centered around the Internet rush hour (7-9PM) [9]. An updated app resident on an eBox can alleviate this pressure at peak times. The CDF of the number of app updates and installs per day in Figure 2(ii) highlights the long tail distribution of app traffic with respect to days, i.e., there are a few days with heavy updates, but the majority of days are without any updates. Also, shown in Figure 2(iii) is the CDF of app traffic per hour of day, to provide a complete picture of the data collected as part of this work. An interesting element is that app updates are distributed throughout the hours of a day which seems contradictory to results in 3(ii). But this is a result of the cumulative effect caused by combining hourly distributions during peak and non-peak days. *AppFlux can alleviate pressure on peak hours and at the same time, reduce the interruptions seen by end user on their devices due to absorption of updates by the eBox which will be seen only if the app is actually used.*

4 App Anatomy

The design goals of AppFlux are (i) to provide end users with fresh app versions without the need for explicit or unnecessary updates, (ii) to ensure that no degradation is observed due to app streaming, while also (iii) laying the basis for efficient and scalable app caching on eBoxes.

We first determine the viability of app streaming for Android devices, by inspecting the top 100 Android apps in each category available in the Google Play Store, for a total of 4,969 apps. This inspection identifies app constituents, but also aims to quantify their evolution over a year and finally, the impact of this evolution on the app update process and on app performance. We also measure the time spent in loading classes and assets compared to the app's total execution time, to gauge the impact of streaming on app loading performance.

Previous efforts geared towards application streaming for end consumers have been thwarted limiting their use by enterprise customers only, e.g., Windows application virtualization [11, 14]. This was attributed to the inability of the end user applications to achieve 100% compliance due to their run-time dependence on non-Windows (or third party) services. By design, Android apps are

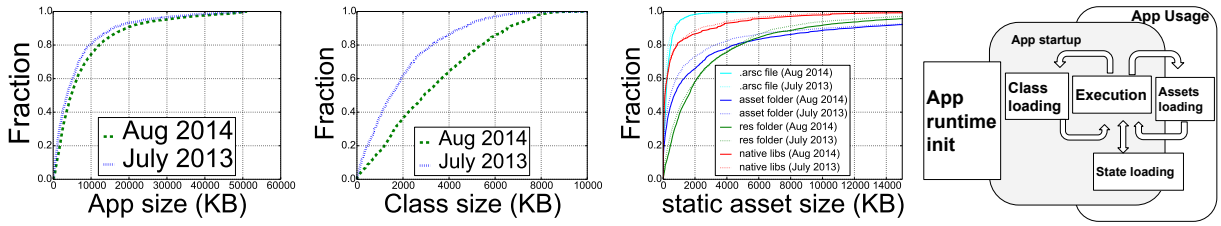

Figure 4: App anatomy analysis for the Top 100 apps in each category available in the Google Play Store in July 2013 and August 2014: (i) showing CDF of total app sizes (apk file size); (ii) showing CDF of uncompressed class sizes in apps; (iii) showing CDF of other static assets (asset folder, res folder, arsc file, native libraries); (iv)Showing app execution phases concerning app streaming.

standalone entities, running as separate (uid, gid) pairs, and rely solely on Android platform features for their implementation. So, for Android, if functionality other than what is available on an Android platform is needed by an app (e.g., specific API libraries, etc.), it is packaged in the app itself, and inter app communication is explicit (via API or supported Intent functionality). *Therefore, run-time dependency is not an issue for Android apps.*

Classes, Static Assets – Main App Constituents

Figure 4(i) shows the cumulative distribution function (CDF) of the total compressed size of downloaded apps, i.e., apks. It clearly show that app size is increasing over time, up to (but not exceeding) the 50MB limit imposed by Google. However, Google also provides two expansion files (2GB each) that can be used by developers to provide any additional resources required by an app. We conservatively analyze only apks and their components, despite the fact that any update on either expansion file also triggers an app update. Figures 4(ii) and (iii) show the CDF of the apps uncompressed class size, and that of apps' other static assets. Figure 4(ii) shows that app class size has increased for all the apps, hinting at increased app complexity, a trend expected to continue as more complex functionalities are introduced by newer apps. We limit the max value of the x axis in Figure 4(iii) to clearly highlight that the sizes of other static app components do not vary much, with only slight increases. This does not mean that these components do not change, but rather, those changes are not visible as changes in component sizes (e.g., changes in UI images or icons, etc.). It is also clear that app anatomy varies significantly. *AppFlux should not only consider app executable, but should handle streaming of static app assets as well.* This also highlights an important point about why mechanisms like on-demand class loading or JAVA reflection do not suffice for mobile app streaming.

Class and Asset Loading Affects App Performance

For purposes of app streaming, an app's execution can be divided into four phases, as depicted in Figure 4(iv): (i) initialization of the application run-time, provided by the OS on the device (e.g., Dalvik for Android); (ii) loading the app executable from local storage (i.e., the internal eMMC or the external sdcard); (iii) loading app state either from the network or from local storage; and (iv) actual app execution based on user inputs and other loaded app assets (e.g., background images, A/V, etc.). In addition to system delays seen during app launch (e.g., Dalvik VM launch, loading system classes etc.), there are also app-specific contributions to delay during app execution. From AppFlux's perspective, these delays are of interest only insofar as the loading of app-specific classes via app streaming contributes to these delays. For our analysis, we used class prefixes in class URIs (e.g., com.android used for App framework's internal classes, com.google.android for Google's app classes like Maps, commands.monkey for Monkey runner classes, etc.) to filter app-only classes, which are typically part of app updates, and to extract performance metrics pertaining to app components present in apk files. Measurements in Figures 7(i) and (ii) in § 11 highlight an important fact: *class loading and static asset loading are important factors in app execution performance*. The outcome of these evaluations is a practical upper bound on potential performance improvements due to reduced class load times: if load times are reduced to zero, performance can be improved by up to 50%.

Summarizing, given higher access bandwidth of WiFi than flash memory and class/asset loading being an important factor in app performance, app streaming can potentially improve app performance.

5 AppFlux Design

AppFlux encompasses the eBox-remote app store interfaces needed to obtain updates. The outcomes of eBox interactions with a remote cloud or app-store is that eBoxes locally cache those apps that are more likely to be used by nearby mobile devices. AppFlux complements existing mechanisms for "on device" app installs and/or updates with instant eBox-based, just-in-time streaming of app executable, whenever apps are run. A desirable side-effect of running an app from the nearby eBox is

a consequent update to the app resident on the device, but as this is done asynchronously with the app's execution, end users do not directly perceive such an update, nor do they experience the delays of update actions (they will see the battery power consumed by such updates, however). AppFlux operates by relying on device-resident functionality to transparently intercept requests for app components like classes, static assets like images, XML, etc., and then redirect such requests to network-accessible eBoxes or to the device's local storage. The latter is so that non-eBox enhanced devices operate just like today's devices, relying on updated apps stored locally. Devices enhanced with app streaming, however, can use apps stored in the accessible eBox, with the desirable outcome being that users will use the newest app version currently available on the eBox. In case of disconnection while streaming an app, there are two cases that arise: (i) subsequent usage of the app requires only the already fetched app components in which case user would not see any interruption and (ii) subsequent usage requests an app component which hasn't been streamed yet. In that case, user will see an interruption and would have to restart the app.

The realization of AppFlux developed in our work addresses the technical challenges enumerated earlier, with a design that splits the app eco-system into two components: (1) a front end end user device – eBox interface, and (2) a back end component interfacing the eBox with a remote cloud or app store. As shown in Figure 5, the main elements of AppFlux are (i) the *AppFlux Client* – embedded in app framework of mobile OSes, (ii) the *AppFlux Server* – providing on-demand streaming of app components, and (iii) a *Device Cell Link (DCL)* – that comprises a protocol for bootstrapping and app usage reporting which then, drives the prognostic adaptation in app streaming. For completeness, (iv) we also sketch the expected interactions between eBoxes and the remote cloud, such as populating app caches in eBoxes (push- or pull-based) and app-profile consolidation, but defer to future work topics like caching policies, governing policies about how app executable, static assets, and associated app state are managed – jointly termed *Code State Cache (CSC)*.

The implementation of AppFlux uses either available Android platform components or open source technologies. Specifically, (i) the device-side AppFlux client elements are implemented as a patch for Android (specifically, modification and/or enhancements in cutils, dalvik, androidfw, and init.rc), and (ii) the eBox-resident elements are implemented using the node.js TCP socket API. (iii) Additional elements of the eBox-app store interface can be implemented using HTTP API exposed by app stores like Google Play.

6 AppFlux Client

The AppFlux front-end component provides the device-eBox interface and is designed as a split module, one half residing deep inside the mobile OS and the other resident on an eBox, as shown in Figure 5(i).

***Operation Duality* enabling Connection Manager.** The connection manager is responsible for communication setup when an end device first connects to a streaming-enabled eBox. This entails (i) exchange of a list of installed apps, their versions and signatures, and (ii) exchanging the streaming server's configuration, i.e., port number, IP address, and apps available on the eBox. In the current prototype, a file in the Android file system, i.e., /data/AppFlux/app-list, is used to store a list of apps available on the currently connected eBox. On-device population of this information is implemented as a patch in Android's cutils exposing a wpa-supplicant listener that raises a new Wi-fi connection notification which then triggers AppFlux's bootstrap process.

***Prognosis Enabling* App Signature.** App streaming maintains a concise representation of each app, termed *app signature*. It is a list of the app's executable components and/or assets that are loaded during its execution on the end client device. The device-resident app signature module is responsible for maintaining these signatures for all device-accessible apps. In Android, this is achieved by keeping a list of classes and/or static assets to be loaded during app execution. The signature is stored as an app-specific file in the Android file system, specifically, in the configuration folder for app streaming on the device, i.e., /data/AppFlux/*app-name*/signature. App signatures are shared with eBoxes via the connection manager when the device first connects to an app streaming-enabled eBox. This is used by eBox to adapt the streaming content to include components that would be needed by that app by a particular user.

***Operation Duality and Sync Agnosticism* in App Proxy.** In a streaming-enabled device, the app proxy is responsible for intercepting executable and/or static asset requests (at the app run-time layer) and redirecting them to the network or to local storage, the latter based on connectivity information in the connection manager that actively monitors the device's connectivity to an eBox. Such interception makes it possible for app streaming to be transparent to app developers, i.e., no changes are required to existing apps. The app proxy also interacts with the app-signature module, to store and/or retrieve history-based app signatures. In our Android prototype, the app proxy is implemented as a module in the Dalvik VM (for app classes) and the androidfw module (for static assets). In Android, when an app requests a class via JNI, it is loaded from the classes.dex file or from dex-optimized classes cached in an app specific directory.

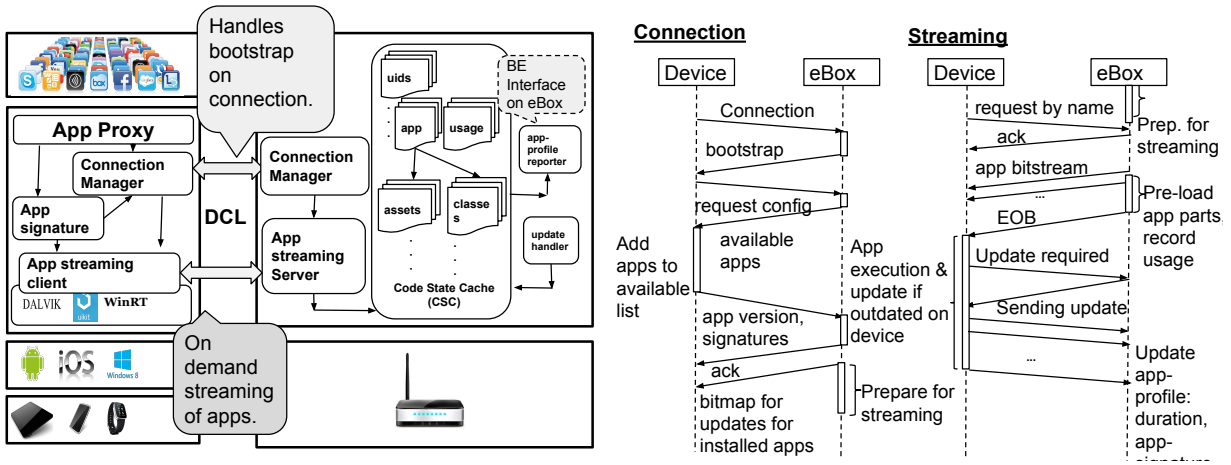

Figure 5: AppFlux Design: (i) AppFlux device-eBox interface; (ii) DCL protocols for connection and app streaming.

When a static asset is requested, it can be loaded from the multiple locations, i.e., res folder, arsc file, asset folder, icon directory, etc., present in the app's apk installed on the device after passing through an unzip data stream. A URI and/or buffer mode can be used for loading static assets. The app-proxy spans Dalvik and the Androidfw library, and forwards request to the app streaming client or uses existing loading mechanisms to load from local storage. App-proxy provides URIs for classes, e.g., com.adobe.reader.*classname*, which are identical to those specified to the class loader by its JAVA framework or app's locale, vendor information and asset's name, to allow eBox to correctly select and stream required assets, as explained in the next sections. With this design and implementation, since there are no changes in the interfaces used by apps, even for static asset loading, app streaming operates without requiring app changes and invisible to end clients. The app proxy is also responsible for handling update triggers from eBoxes. On the trigger, it uses existing interfaces (i.e., Android's package manager) to trigger app update on device from an eBox. The incremental updates when delivered from app stores are consolidated at eBox for streaming but are delivered as incremental updates only to devices which are handled in the same way as they are handled by existing devices.

***Invisible* App streaming Client.** The app streaming client is simply a TCP socket client that accepts requests from the app proxy and then handles app streaming responses from the server on the eBox. It is embedded in dalvik and the androidfw library to cover all app components. TCP sockets are used (vs. HTTP) to minimize connection and packet overheads during streaming. The implementation modifies Android's cutils to expose APIs to dalvik and androidfw library that allow it to request classes or static assets from eBox.

7 AppFlux Server

***Sync Agnosticism* enabling App streaming Server.** The app streaming server resides on an eBox, listening to requests from end client devices for app executable and static assets, on a port number exchanged during connection setup. It is implemented as a node.js-based TCP socket server, and interfaces with an in-memory cache for classes and static assets – part of the code-state cache (CSC). It translates class names to directory entries, which in turn are pre-populated based on eBox-app store interactions, where apps are de-compiled on the eBox into the same directory structure as the source. The loading of static assets is more complex, because (i) there are more locations to search for an asset including the asset folder, the .arsc file, the icon directory, xml files, etc., (ii) the same asset may refer to a different file depending on locale information, and (iii) some apps may choose to load assets in a raw buffer and parse individual assets for performance reasons. This leads to some additional computational overhead imposed on the eBox' app streaming server – rather than the end client device – to maintain an index and to search and load static assets. An important server action is to compare the app version on the device to the one resident in its own repository. If the app on the device must be updated, the server also issues an update request, but only after the user stops streaming the app. This triggers the app proxy to perform a background update on the device-resident app.

***Prognostic* Code State Cache (CSC).** The CSC is an eBox-resident in-memory app repository. It can be populated using policies that govern (i) preparation for app streaming, e.g., by pre-fetching in memory those apps that will be used soon based on user behavior, (ii) app refresh on the eBox, and (iii) app replacement to conserve limited eBox resources. We have not yet explored

this functionality in detail, but submit that its likeness to how CDNs operate [1] strongly suggests the feasibility of its realization. In the Android prototype, for all (uid, app) pairs, there exist executable and static assets in the CSC (e.g., app background images, embedded A-V content in apps, etc.). In order to minimize memory usage on the user's device and the number of requests to app streaming server on the eBox during streaming app execution, we have explored multiple approaches for storing executable on the eBox. We first used existing single classes.dex file and streamed it at the start of an app, but that needlessly stresses device memory and increases delay, as classes.dex typically span multiple initial packets. Second, we de-compiled apps and create dex file for each class that can be streamed as requests arrive. This is good for device memory but bad for the number of requests to the eBox. We therefore, maintain multiple dex files for a single app, each including a subset of classes from classes.dex, where the size of each dex file depends on the TCP/IP payload limit, i.e., 65536 bytes, and the order in which these dex files are delivered is derived from the app signature. This requires pre-processing of app classes on the eBox (which remains transparent to app developers). Currently, CSC population is performed manually, using the dex2jar [7] and dx tool from Android SDK to extract and repackage dex files which we plan to automate using inotify callbacks on app repository on eBox's file system.

***Invisibility* enabling Device Cell Link (DCL).** The purpose of the DCL protocol is (i) to realize seamless access to app executable and static assets on the eBox, and (ii) to reliably stream app executable components. In Android, DCL operates by referring to classes by their qualified names, as seen on the device, using hooks implemented in the Dalvik VM as it gets launched before any loading request is issued by an app. Reliability in streaming is obtained by implementing DCL on top of TCP/IP sockets (vs. using UDP). Operationally, as also shown in Figure 5(ii), a connection phase, is followed by a streaming phase in which app classes, assets are streamed to the device running the app.

8 AppFlux eBox-App Store Interface

The AppFlux back-end components span the eBox and the remote cloud hosting app-store functionality (e.g., Google Play Store). As stated earlier, its realization can leverage CDN functionality, but we have not yet explored challenges concerning its efficient operation. The key components in eBox-App Store interactions needed for AppFlux include the following.

An eBox-resident *app profile reporter*, shown in Figure 5(i), is responsible for creating and maintaining app-profiles for the users employing its app streaming services. An app-profile is a relational structure containing user identifier (uid), device identifier, app id, and app usage-related information maintained per user on an eBox. Beyond using such profile data for optimizing eBox operation and improving end user experiences for app streaming, of additional interest may be to sync such profile data with app stores.

To support a push-based mechanism for update distribution, AppFlux requires an eBox-resident *update handler*, operating much like asynchronous callbacks triggered by the remote app-store on relevant app updates. An update is committed only when there are no current users for this app, to avoid misalignment of app versions, but once committed, the update is immediately available to eBox users. The implementation maintains a uid bitmap for each app that specifies whether an app is being used by a certain user.

In addition, AppFlux relies on *app store-resident functionality* to provide the aforementioned callbacks or eBox-initiated syn operations, to leverage AppFlux's app profiles and other information gleaned from eBox usage patterns to guide the disitribution of app updates across eBoxes, or to otherwise allow app stores to benefit from the presence of AppFlux enabled eBoxes in the end-to-end app ecosystem.

As stated earlier, this paper focuses on understanding app streaming costs and opportunities from the perspective of end clients. We leave additional detail and evaluation of eBox components for future work.

9 Traffic Reduction in the Last Mile

Figure 6(i) shows the estimated reduction in traffic if the app installs and updates are cached at the eBox for 1 to 15 days. Our analysis of Internet traffic shows the use of eBoxes to result in potentially saving up to 75% of the last mile traffic due to app updates and up to 42% of traffic from app installs. We see such huge benefits because when two or more devices near the same eBox update or install the same app, this leads to duplicate requests from all those devices to go to Google Play Store via the Internet. Those apps are then delivered directly or by Google's CDN, but in both cases this leads to repeated delivery of the same apps binaries to different devices in the last mile which holds true for app updates as well. With an eBox, redundant last-mile traffic is eliminated by re-using the single update or install already present on the eBox. *This clearly shows that an eBox-based app cache can significantly reduce traffic in the last mile of the Internet.*

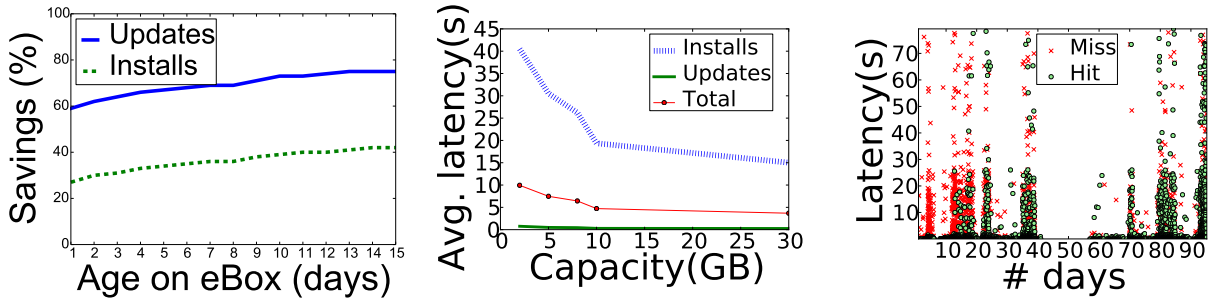

Figure 6: (i) Traffic reduction in the last mile due to apps and their updates with the age of app on eBox i.e., app is stored on the eBox for those many days after it has been seen for the first time; (ii) Average response latency in fetching an app with changing eBox storage capacity: On a cache hit, app is fetched from an eBox else from a remote cloud based app store of different capacity; (iii) Latency variation with age of apps/updates as number of days with 10GB of LRU cache at eBox.

10 Faster App Delivery

We analyze the impact of eBoxes on app delivery by considering eBoxes as a 'nearby' addition to existing CDNs. We simulate an eBox and a single nearby CDN POP (point of presence) using the open sourced Globule CDN implementation [35] deployed on our OpenStack-based cloud infrastructure. Our setup represents the best case for any CDN node because it is present within the same ISP (i.e., Georgia Tech's network) compared to CDN which are situated near ISP's nodes. We use captured app traffic traces to generate a representative app traffic workload. Figure 6(iii) shows an achieved cache hit rate of 93% with 10GBs of eBox storage. The left part of the figure shows the cold start of the eBox cache, where subsequent requests to the same app result in cache hits and Figure 6(ii) shows the average download latency from an eBox with a 10GB of cache. In case of hit, the app is fetched from an eBox and an app is fetched from a remote cloud based apps store in case of a miss. The app install and update times can be significantly reduced in case of cache hit as can be seen from red and green dots in Figure 6(iii). From Figure 6(ii), we can see that the average install and update times can be reduced by a factor of 2x, i.e., from 40.29 to 19.362 secs., when the eBox cache size varies from 2GB to 30GB. From these results, we see that a 10GB cache can absorb most redundancy in the last mile due to app traffic. Important points to note here are (i) AppFlux does not compete with the CDN, but complements its operations. (ii) We must consider relative improvements in bandwidth and latency measured as opposed to the absolute values measured in our simulated experiments. *An eBox based cache with reasonable storage, i.e., 10GB, can provide up to 2x faster app delivery than CDNs.*

11 Streaming Improves App Performance

Experiments are run on a Nexus 5 phone with Android (CyanogenMod 11.2 $\sim$ Android KitKat). The eBox is emulated with a Core2Duo machine housing apps in its local storage and connected to a Linksys wrt 1900ac router via a Gigabit port. A local wireless LAN using only the 5GHz frequency band (with up to 1.3 Gbps capacity) provides 802.11ac WiFi connections for app streaming. The phone is placed near the router for experiments, with only two entities connected on this network during experiments, i.e., the phone and the eBox. Interactions between the eBox and the app store are emulated by downloading 4,969 apps from the Google Play Store using its unofficial Python API. Figure 7 depicts the time spent in loading app components during execution. The data shown is the median of five measurements, resulting from a total execution time of 10 secs. after app startup, for the Top 100 free apps in each category available from the Google Play Store. Apps are run using the Monkey runner available as part of the Android app ecosystem for app start up, with a fixed numbers of random events (20 UI events with a spacing of 500 ms) mimicking a real use of an app by an end user. We exclude the first run when device-specific optimization is applied by the existing Android dexopt tool, for apps installed for the first time.

Figures 7(i) and (ii) show the CDF of the time spent in loading app-specific classes and assets, respectively. As seen in Figure 7(i), load times for app-specific classes vary from a few milliseconds to slightly more than 2 seconds, for the 99th percentile of apps. Similarly, Figure 7(ii) shows the time taken to load app-specific static assets varying from a few milliseconds to 2 seconds for the 99th percentile of 10 secs. The measurements in Figure 7(iii) suggest that the time taken to load system classes, for the 99th percentile, is around 5 secs. These

measurements support our assumption in § 4 about the impact of app loading on app performance. This also suggests that these (surprising) results are primarily due to (i) Android's dex file format, which packs all app classes into a single file to reduce the app's storage footprint, (ii) Android's class look-up method, which uses a hashmap and utilizes a memory mapped dex file to load classes, and (iii) finally, the effective flash memory access bandwidth available to the class loader. The consequent technical approach chosen for app streaming addresses these issues by (i) eliminating the need for compression in the dex file format, by keeping uncompressed app classes and static assets in the memory-rich eBoxes, thus trading storage capacity in eBoxes for performance, (ii) shifting class lookup overheads to eBoxes, by modifying the existing Dalvik class loading mechanisms, and (iii) leveraging the improved access bandwidths of future WiFi networks, as highlighted in Section 2. A positive additional side effect of improving app class loading is a reduction in system class load times, shown in Figure 7(iii), also bolstering app performance. *Using streaming apps and under ideal network conditions, app loading performance can be improved by up to 20%.*

12 App Streaming Server - Multi-tenancy

For eBox micro-benchmarks, we use a powerful (Intel core i7, 16 GB DDR3 RAM) and 802.11ac capable laptop to simulate concurrent clients connected on the same network link. To show that a single eBox can simultaneously serve a modest number of end user clients, with no or little degradation in app streaming performance, we measure the variation in latency of app streaming with different numbers of concurrent eBox clients, averaged over a fixed number of requests per client (set to 20 per client – because most apps end up fetching 20 chunks of components, including classes and assets). The concurrent client load are simulated Node.js processes, and the bandwidth achieved is limited to 270 Mbps, the latter because of a limitation in the laptop's Wifi adapter that supports a maximum of 8x40 Mbps simultaneous connections on 5GHz. Conservatively using full dex file response payloads for every request (the actual AppFlux implementation does not have such large responses), Figure 7 (iv) shows that our current (un-optimized) eBox implementation can handle 100 concurrent clients with reasonable latency. Beyond that number, we observe the bottleneck to shift from network to storage I/O on the eBox, which leads to degradation in response time, as shown in the figure. This suggests the need for careful eBox implementation and optimization for client multi-tenancy, which we plan to undertake in the future. Nevertheless, the micro-benchmark highlights that *even with an un-optimized implementation, an eBox can handle app streaming for a reasonable number of clients, thus establishing the viability of eBox-based app streaming*.

13 Related Work

Application Streaming. Prior work on app streaming includes (i) 'thin' desktop clients [6, 11, 14, 19] streaming pixels to low cost front end displays. (ii) Numecent's cloud paging [15] streams memory pages to end clients. (iii) Instart logic's [8] webapp streaming exploits the order of relevance of app components using a browser-resident nano-visor to optimize web app delivery. A recent offering by Amazon performs mobile app streaming using a wrapper SDK. AppFlux differs from such efforts because (i) it focuses on mobile apps vs. desktop applications, and (ii) it exploits the emerging eBox tier.

Mobile app analysis. Earlier work has analyzed mobile apps from various perspectives, e.g., quality [27], unsafe exposure [26], security [41], privacy [38], etc. We provide a performance-centric analysis of mobile app structures relevant to enabling app streaming.

Mobile app architectures. There are three prevalent app architectures: (i) Native Apps, (ii) Web Apps, and (iii) Hybrid Apps. We propose an alternate approach that can reap the benefits of both native apps and web apps, by streaming native code to end client devices.

Online application update. Dynamic software updates [37, 33, 31] leading to reference management tools [20] have been explored for enterprise IT teams, with a focus on managing system updates for large organizations [3, 12]. But no such tools/technique exist to benefit end users of mobile devices and apps. AppFlux addresses this gap. Recent work to reduce mobile app update traffic proposes micro app updates [23] which would complement AppFlux.

14 Discussion and Future work

This paper leaves a number of open questions on device side, eBox's deployment model, privacy and required system software changes. Ones which we plan to undertake in future are discussed below:

Energy consumption on devices. With almost all apps requiring network access (98.9% in the apps we used for evaluation) and hence, turning on the network connection, We posit that app streaming can potentially piggyback the tail energy [34], causing minimal affect on overall energy consumption which requires detailed evaluation.

Cellular network. An important question that is left open in this paper is whether AppFlux is relevant for cellular technologies because all of the experiments are run using Wi-Fi. We posit that similar opportunities exist in

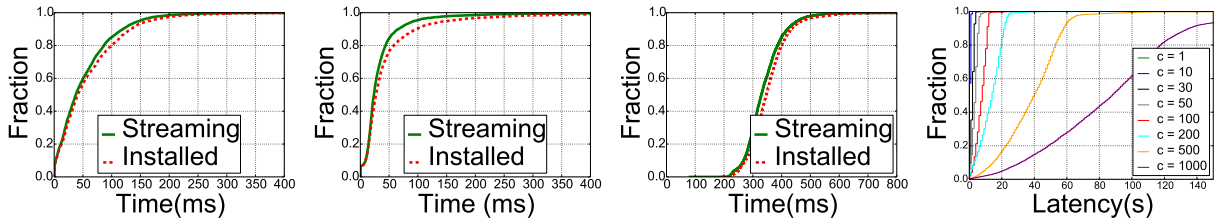

Figure 7: App loading performance comparison (streaming vs. installed) (i)showing CDF of time taken to load app specific classes (ii) showing CDF of time taken to load static assets of app (iii) showing CDF of time taken to load system classes. (iv)showing CDFs of latency in total response time per request varying with number of concurrent clients (c).

future cellular technologies using small cells as eBoxes. However, we could not experiment with small cells because of lack of open small cell platforms that we could use carry out experiments as most commercially available small cells are closed platforms (sold by cellular providers) making it close to impossible to insert app streaming functionality on them. Subject to availability of open cellular platforms, we are exploring ways to include evaluation of eBox functionality on cellular technologies.

Deployment Model. Given that eBoxes deployment doesn't exist as yet, there are open questions like who will own the eBoxes – individuals, businesses or will they be part of public infrastructure ? How to securely run app streaming servers on those eBoxes ? How can an end user trust an app from a particular eBox ? Another aspect is mechanisms to manage and ensure DRM of the app repository on eBox. We posit that with existing authentication and authorization methods deployed on eBoxes, this can be addressed theoretically. But it leaves out one concern that is not addressed in existing methods i.e., current authorization and authentication assume a human user which is authenticated or authorizes which is not the case for eBox. We believe that this is an interesting problem which plan to study in details.

Privacy Concerns. Sharing app signature with an eBox may raise privacy concerns as it effectively represents how an app is being used by a particular user. We argue that this would not be more invasive than current support available in apps. Specifically, with recent rollout of app usage API in android[2] which allows developers to track app usage, it seems that sharing app signature is more obfuscated than what is guranteed in latest version of android. However, this aspect certainly needs a detailed evaluation.

System software on devices and eBoxes.. Further, we plan to push in two directions: (i) Current design of executable file format is driven by the assumption that executables are stored in the local storage of devices but for streaming apps, these assumptions do not hold. We are pursuing exploration of the effects of app layout e.g. dex for android app classes, elf for native code, etc. for streaming apps and (ii) There is no standard way of deploying eBox based functionality e.g., app caching and streaming etc. In our future work, we are considering exploring additional functionality that can further improve the app ecosystem and also ways to automatically provision those on eBoxes.

15 Conclusions

AppFlux is an app streaming service for the Android ecosystem. Its use relieves end users from having to explicitly update their numerous on-device apps, yet ensures their ability to always run the newest app versions. AppFlux's implementation leverages the growing 'edge tier' of the Internet, by relying on edge boxes – eBoxes – that can stream apps to end clients wishing to run them, cache the newest versions of popular apps, and interact with remote app stores.

The design of AppFlux is based on extensive experimental measurements of app traffic and analysis of app anatomy. These measurements document the bandwidth consumed by such app traffic, and they show that AppFlux can reduce traffic volume by up to 70% with 2x improvement in their delivery latency over CDNs. Appflux achieves this without compromising app performance or requiring changes to apps by developers and/or changes in how end users employ these apps.

Acknowledgement

This work is partially supported through grants Intel, VMware, and NSF CNS1148600.

References

[1] Akamai technologies @ http://www.akamai.com/.

[2] Android app usage api @ https://goo.gl/39dqlu.

[3] Apple enterprise management tool @ https://www.apple.com/ipad/business/it/management.html.

[4] Att small cell deployment plans @ http://www.att.com/common/about_us/pdf/small_cell.pdf.

[5] Att wifi hotspot locations @ https://www.att.com/maps/wifi.html.

[6] Citrix xenapp @ http://www.citrix.com/products/xenapp/how-it-works/application-virtualization.html.

[7] Dex2jar tool @ https://code.google.com/p/dex2jar/.

[8] Instart logic web application streaming @ http://instartlogic.com/.

[9] Internet rush hour @ http://goo.gl/yknna5.

[10] Level 3 cdn reports last mile as new bottleneck @ http://blog.level3.com/open-internet/heads-isps-win-tails-you-lose/.

[11] Microsoft appv @ http://www.microsoft.com/en-us/windows/enterprise/products-and-technologies/mdop/app-v.aspx.

[12] Microsoft system center @ http://technet.microsoft.com/en-us/library/cc180239.aspx.

[13] Mobile world congress - small cells @ http://www.mobileworldlive.com/operators-eye-greater-small-cell-deployment.

[14] Novell application virtualization @ http://www.novell.com/products/zenworks/applicationvirtualization.

[15] Numecent cloud paging @ http://goo.gl/k34n4d.

[16] Qualcomm small cells @ http://goo.gl/e6vv71.

[17] Qualcomm smart gateways @ https://www.qualcomm.com/media/documents/files/smart-gateway-analyst-presentation.pdf.

[18] Reducing cost of storage @ http://www.zdnet.com/storage-in-2014-an-overview-7000024712/.

[19] Symantec workspace streaming @ http://www.symantec.com/workspace-streaming.

[20] Wikipedia - reference management tools @ http://goo.gl/c7pbha.

[21] ALI, Q. I., AND AL-WATTAR, A. Z. S. Bandwidth-delay measurements of a wireless internet service providing (wisp) system. *Int. Arab J. e-Technol. 1*, 1 (2009), 83–89.

[22] BHARDWAJ, K., SREEPATHY, S., GAVRILOVSKA, A., AND SCHWAN, K. Ecc: Edge cloud composites. In *Proceedings of the 2014 2Nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering* (Washington, DC, USA, 2014), MOBILECLOUD '14, IEEE Computer Society, pp. 38–47.

[23] CHEUNG, A., RAVINDRANATH, L., WU, E., MADDEN, S., AND BALAKRISHNAN, H. Mobile applications need targeted micro-updates.

[24] DIXON, C., MAHAJAN, R., AGARWAL, S., BRUSH, A., LEE, B., SAROIU, S., AND BAHL, P. An operating system for the home. In *NSDI* (April 2012), USENIX.

[25] GIBLER, C., STEVENS, R., CRUSSELL, J., CHEN, H., ZANG, H., AND CHOI, H. Adrob: examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 431–444.

[26] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2012), WISEC '12, ACM, pp. 101–112.

[27] IVAN, I., AND ZAMFIROIU, A. Quality analysis of mobile applications.

[28] JANG, M., SCHWAN, K., BHARDWAJ, K., GAVRILOVSKA, A., AND AVASTHI, A. Personal clouds: Sharing and integrating networked resources to enhance end user experiences. In *INFOCOM, 2014 Proceedings IEEE* (April 2014), pp. 2220–2228.

[29] KOUKOUMIDIS, E., LYMBEROPOULOS, D., STRAUSS, K., LIU, J., AND BURGER, D. Pocket cloudlets. *ACM SIGPLAN Notices 47*, 4 (June 2012), 171.

[30] LIU, H., AND EL ZARKI, M. An adaptive delay and synchronization control scheme for wi-fi based audio/video conferencing. *Wireless Networks 12*, 4 (2006), 511–522.

[31] LYU, J., KIM, Y., KIM, Y., AND LEE, I. A procedure-based dynamic software update. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on* (July 2001), pp. 271–280.

[32] MÜLLER, C., LEDERER, S., AND TIMMERER, C. An evaluation of dynamic adaptive streaming over http in vehicular environments. In *Proceedings of the 4th Workshop on Mobile Video* (New York, NY, USA, 2012), MoVid '12, ACM, pp. 37–42.

[33] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 72–83.

[34] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 29–42.

[35] PIERRE, G., AND STEEN, M. V. Globule: A platform for self-replicating web documents. In *Proceedings of the 6th International Conference on Protocols for Multimedia Systems* (London, UK, UK, 2001), PROMS 2001, Springer-Verlag, pp. 1–11.

[36] SHEPARD, C., RAHMATI, A., TOSSELL, C., ZHONG, L., AND KORTUM, P. Livelab: measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev. 38*, 3 (Jan. 2011), 15–20.

[37] SUBRAMANIAN, S., HICKS, M., AND MCKINLEY, K. S. *Dynamic software updates: a VM-centric approach*, vol. 44. ACM, 2009.

[38] WETHERALL, D., CHOFFNES, D., GREENSTEIN, B., HAN, S., HORNYACK, P., JUNG, J., SCHECHTER, S., AND WANG, X. Privacy revelations for web and mobile apps. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 21–21.

[39] XU, Q., ERMAN, J., GERBER, A., MAO, Z., PANG, J., AND VENKATARAMAN, S. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 329–344.

[40] YAN, T., CHU, D., GANESAN, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 113–126.

[41] ZHOU, W., ZHOU, Y., GRACE, M., JIANG, X., AND ZOU, S. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2013), CODASPY '13, ACM, pp. 185–196.