1

AppSachet: Distributed app delivery from the edge cloud

Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilovska, Karsten Schwan

{*kbardwaj6, pragya.agarwal*}*@gatech.edu,* {*ada, schwan*}*@gatech.edu*
Georgia Institute of Technology, Atlanta GA 303332, USA.

Summary. With total app installs touching 100 Billion in 2015, the increasing number of active devices that support apps are posed to result in 200 billion downloads by 2017. Data center based App stores offering users convenient app access, however, cause congestion in the last mile of the Internet, despite use of content delivery networks (CDNs) or ISP-based caching. This paper explores the new paradigm of eBoxes, situated in the 'edge cloud' tier beyond the last mile, which can be used to alleviate this congestion. With redesigned app caches – termed AppSachet – such edge cloud based distributed caching can achieve a hit ratio of up to 83%, demonstrated on real-world Internet traffic. The redesign leverages proposed new caching policies, termed p-LRU and c-LRU, specifically targeted at eBoxes' limited storage and for the traffic caused by app installs and updates. A cost benefit analysis shows that the additional cost required to deploy AppSachet on eBoxes can be recovered within the first three months of operation.

Key words: App delivery, Internet traffic measurement, Edge cloud, Caching

1.1 Introduction

The number of active smart phones worldwide is posed to cross 3 billion by 2018, and additional increases in mobile devices stem from wearable and embedded devices, like smart watches and glasses, devices supporting smart vehicles, etc. Coupled with that is a continuing explosion in the number of apps available to end users, with roughly 100 billion app downloads reported in 2015, set to reach a staggering 200 billion by 2017. App installs and more so, app updates, therefore, place measurable pressure on the Internet infrastructure used for their delivery, currently relying on Internet Service Provider (ISP) links to reach remote datacenter-based app stores or the Content Delivery Networks (CDNs) they use. [1].
Specifically, the issue is congestion in *the last mile* of the Internet, which is well known to be a bottleneck for delivered service quality [7]. For apps,

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

CDNs cannot mitigate this bottleneck because they operate behind ISPs and cannot consolidate app requests on their behalf. At the same time, ISP-based caching is difficult if apps and updates are flagged as non-cachable content due to their pay-per-download nature, issues related to intellectual property protection, etc. Our previous work [11] showed the feasibility of using devices operating at the 'edge cloud' tier beyond the last mile of the Internet, to deliver apps/updates. Examples of such devices – termed eBoxes – include small cells[9, 10], WiFi routers [4, 5, 8], or cloudlet servers [20], shown useful in recent research for supporting new edge services [16, 18, 20, 12]. In that work, we also developed novel app streaming technology, which, without any disruption to how apps are currently developed and used, permits users to install apps or app updates directly from eBoxes with 2x faster speed, while also reducing last mile congestion by up to 70%. Such work, however, focused on the client-facing eBox capabilities, and its obtained benefits relied on age-based (i.e., LRU-based) eBox-resident resource management mechanisms. However, that approach leads to comparatively inefficient use of limited eBox resources (e.g., storage capacity), limiting eBox benefits. In comparison, this paper seeks to answer the following questions:

- How to best cache apps and/or updates – *AppSachets* – on eBoxes?
- How to efficiently use the eBox's limited resources (i.e., storage capacity) to maximize hit ratio or minimize caching cost for Internet traffic due to apps and their updates?
- How to articulate cost vs. benefit of AppSachet deployment on eBoxes?

This paper presents AppSachets – a system for distributed app delivery from the edge cloud. Based on our analysis of real world Internet traffic due to Android apps and their updates, we highlight the cacheability characteristics of app traffic. Based on those characterisitcs, we propose two new caching policies implemented as part of AppSachet: (i) p-LRU which takes into account local app popularity and (ii) c-LRU which takes into account the cost of caching apps on eBoxes. We present an end-to-end system design that caters to end client devices using AppSachets and fits in the existing Android app ecosystem, without requiring any changes from app developers or any changes visible to end users. Further, we present a cost model for eBox based AppSachets operation inspired by the pricing model of CDNs. Overall, the technical contributions of this paper can be summarized as follows:

1. **Cacheability of app traffic**: We establish cacheability (§1.6.2) in app traffic using real world measurements of Android app install and updates (§1.2). We highlight the long tail in app access and updates, which also exhibits the peculiar characteristic that the popular apps in the long tail change on an hourly basis.
2. **Efficient app cache:** We show experimentally that the proposed p-LRU cache and c-LRU cache outperform other popular cache policies in terms of hit ratio (§1.6.3). While p-LRU maximizes hit ratio – 83%, c-LRU minimizes the cost associated with caching apps on eBoxes.

3. **Cost-benefit analysis for AppSachets:** We show that the cost of deploying AppSachets on eBoxes can be fully recovered by app stores within the first 3 months (§1.6.5) of its operation. We estimate the additional cost of deploying AppSachets in terms of the cost of storage required(§1.6.4), while benefit is estimated based on the pricing of CDNs.

1.2 Internet Traffic due to app delivery

To assess the impact of app-related traffic on the Internet and assess the improvement opportunities that can be provided via a solution like AppSachet, we collected data about all users at Georgia Institute of Technology over an extensive, representative time period (from May 19, 2014 to Aug 21, 2014). We next describe the methodology of data collection and the findings these measurements that are most relevant to the design of AppSachet.

1.2.1 Data collection methodology

Android app installs or updates are not directly identifiable in the traffic traces available to us. Instead, we observed 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 the 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 used tshark to perform TCP packet reassembly, filtering out all traffic that does not fit the parameters of Google Play HTTP 301 responses. The resulting 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 used 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

	per app		per day		per hour		Total
	max	90%	max	90 %	max	90 %	
Installs	755	10	1377	420	217	11	9536
Updates (raw)	1288	19	4626	1540	443	44	31338
Updates (versioned)	895	20	1377	420	443	44	31338

Table 1.1. Summary of measured traffic due to app intalls and updates.

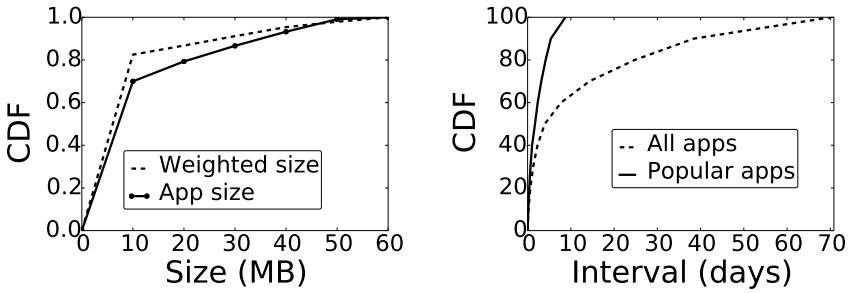

Fig. 1.1. Showing CDF of the (i) Size of the apps at the time data was collected; (ii) caching benefit i.e., number of days between successive app installs and updates observed in the measurements.

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 information to detect an app install or update requires the contents of the HTTP 301 response.

1.2.2 Observations

Table 1.1 summarizes our app traffic measurements. The results are further divided to show the number on app installs or updates observed per app, number of installs and updates observed per day, and finally, number of apps and updates seen during a particular hour of a day. Figure 1.1.(i) shows the distribution of app sizes and the distribution of weighted app sizes where weights for an app is derived from its access frequency seen in our measurements. Figure 1.1(ii) shows the distribution of app access with respect to interval at which apps and/or their updates are accessed shown for all apps and popular apps separately. We derive an app's popularity by ranking apps on their access frequency. It is clear that all popular app updates are finished within 10 days of the first roll out suggesting that the app updates occur in cycle of 10 days. We were not able to find an exact reason for this cycle, but intuitively, it is likely either due to app store's scheduling of app updates or a period arising out of different developers pushing out updates for their apps. In any case, this suggests there is a significant period for eBoxes to absorb updates. Further, the difference in max and 90th%ile shown in Table 1.1, clearly highlights the

bursty nature of app traffic in which app updates outnumber app installs by a factor of 3. The above observations bolsters our hypothesis about the benefits of caching app traffic on eBoxes. However, leveraging this redundancy in app traffic, to reduce congestion in the last mile, requires careful design of the app cache and caching algorithms. Regarding which we derive the following two hypothesis:

1. The gap between popular apps and all apps seen in Figure 1.1.(ii) leads us to hypothesize that a caching scheme that explicitly considers app popularity in its operation can provide the best cache hit ratio.
2. The difference in weighted app size and app size seen in Figure 1.1.(i) leads us to hypothesize that a caching scheme based on (a) cost derived from storage and time an app resides on an eBox and (b) benefit derived from reduction in bytes transferred, can provide good hit ratio while limiting caching costs and hence, pave way for a cost model for edge cloud services.

In addition to the above mentioned technical challenges, AppSachet also requires changes in the way android devices handle app updates. We discuss the design of AppSachet system that addresses all those concerns next.

1.3 App Sachet System Design

AppSachet acts as source of the latest apps and their updates to connected end client devices in similar ways as existing app stores and is placed in the app eco-system as shown in Figure 1.2. AppSachet sees all requests made for apps and/or their updates to app-stores. Its goal is to leverage redundancy in app traffic and provide benefits to end-users and reductions in last-mile bandwidth use, while operating efficiently within limited eBox resources. AppSachet operation starts as a simple LRU cache of web responses (from the app stores) which contain the actual binaries of apps and/or updates requested by end clients connected to the AppSachet enabled eBox. If an end user's request cannot be fulfilled by AppSachet i.e., a cache miss is observed then, it proxies the request to remote app stores and saves the response in its local storage. To ensure high hit ratio, AppSachet ranks the seen apps after a pre-definded bootstrapp time, and based on that ranking segments its own cache into two parts. The segment created are either based on app popularity (i.e., in case of p-LRU cache) or cost of caching (in case of c-LRU cache) or simple LRU. AppSachet syncs or pre-fetches popular or cost effective apps and their updates from remote app stores. Thereafter, the popular or most cost effective apps are updated proactively and pre-fetched every hour. When an end client device that supports AppSachet connects to that eBox, the device starts by sharing information about installed apps on-device to which an eBox response in form of apps and/or updates available at the eBox, depending on user-preferences.

For completeness sake, we outline a simpler version of our vision for how AppSachet is integrated in the Android app ecosystem, by focusing only on

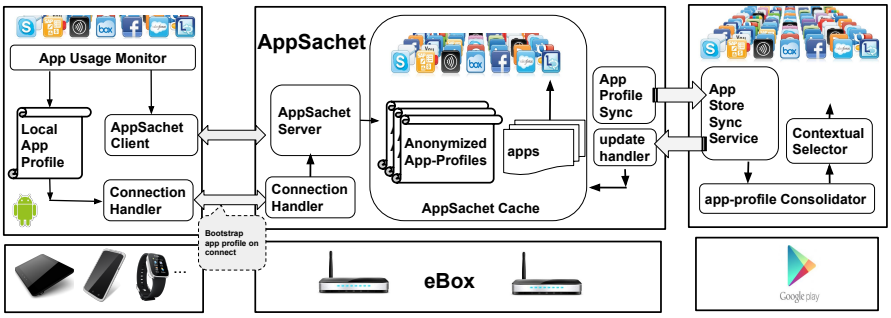

Fig. 1.2. AppSachet design showing different system components, their interactions and their placement in app ecosystem.

interactions related to app installs and updates. These mechanisms are useful even in the context of the existing app download/install/upgrade model, but their benefits can be further enhanced through systems support for app-streaming, developed in our previous work [11]. AppSachet achieves its goals via the following four components:

1. **AppSachet Cache:** An eBox resident module that houses a cache containing apps, updates, and anonymized app-profiles on its local storage.
2. **AppSachet Server:** An eBox resident server that services end client devices' request for apps and/or updates, and collects anonymized app-profiles from the connected devices.
3. **AppSachet Sync:** An eBox resident module pro-actively fetching apps, handling update notifications from app stores and notifying app stores about the delivered apps and/or updates.
4. **AppSache Client** is a module embedded in the Android app framework that enables handling of app installs and/or updates from an AppSachet server.

1.3.1 AppSachet Cache

The AppSachet cache is an eBox-resident module that maintains an indexed repository of app and update binaries fetched from app stores. It houses aggregate app usage information – referred to as *app-profile* – from all connected clients, and a list of delivered apps and/or updates mapped to particular user, used for required app-store notifications. Although the policy used for app cache management can be as simple as an age-based LRU policy, we demonstrate significant gains from targeting the cache management policy to the characteristic of the app traffic. In response, we define two policies – p-LRU and c-LRU – described in greater detail in §1.4. The updates of the app cache rely on AppSachet's Sync service.

In addition to apps and their updates, AppSachet also maintains per-app *App Profiles*. An App Profile is simply a relational structure containing the

state collected from end user devices on connection. It includes the following user specific persistent information from device: (i) a list of apps, (ii) their versions and (iii) usage patterns of installed on end user device. It also contains session specific device configuration, e.g., current IP address of the device needed to deliver an app or update, and the current App Sachet user preferences indicating how user wants his device to interact with AppSachet enabled eBox. For instance, the preferences can indicate whether a user wants to update all available app updates or to disable updating specific app from eBoxes, of is a user wants to see new contextual apps available for installation from eBox, etc. An app-profile is exchanged during the bootstrapping when a device first connects to an AppSachet-enabled eBox.

App-profiles are also kept on eBoxes in another cache instance. The rationale behind keeping a cache vs. a persistent copy of app profiles is first based on the limited amount of storage on eBoxes, and the fact that app-profiles are synced with app-stores anyway. Second, considering the predictability of human movement, i.e., we often go the same places at particular times, e.g. office, coffee shop, etc., creates opportunities for applying proactive and predictive caching algorithms.

Note, however, that sharing this information about a device poses a potential privacy threat; it is avoided by sharing only anonymized app profiles with eBoxes. The anonymization of app profiles is designed to be carried out on the device, in the App usage monitor, vs. on the eBox, to prevent privacy concerns. Another concerns is mismatch in app version installed on device and the one known by backend app stores due to eBox based updates. For the current prototype, it is a non issue because of the way eBox based AppSachet fetches apps and their updates on behalf of an end user effectively syncing the current version of app on device and known by app stores. But we posit that a delegation of authorization from end user to eBox could be used in real-world deployments.

1.3.2 AppSachet Server

AppSache server residing on an eBox carries out interaction with a device. It is responsible for bootstrapping device-eBox interaction on connection by presenting a valid certificate which established that eBox as as valid provider of apps and updates. Another choice is to have remote app stores involved during the bootstrapping process but that leads to longer bootstrapp process as the device and eBox have to reach out to app stores, which then can issue a common token which can be used to verify identity of an eBox. The server interacts with the cache of apps and shared app profiles, and updates and considers user preferences, e.g., to create a tailored response for the device.
Actual App Delivery from an eBox is facilitated by Android Debug Bridge (ADB) over Wi-Fi to connect to the device and carry out actual app installs and updates when requested by a device resident AppSachet client, an app at a time. The decision to not batch multiple app updates from eBox to end

client device is to ensure correctness of updates on a device, and also not to overwhelm the end user device's network with large number of updates.

1.3.3 AppSachet Sync

App-Sachet's Sync service is responsible interacting with existing app-stores on behalf of end clients. Its interaction involves (i) fetching apps and/or updates not cached on an eBox and (ii) periodically checking and pre-fetching updates for apps based on p-LRU or c-LRU policy. It supports a pull-based mechanism for update distribution for which AppSachet on eBox registers a push notification handler, i.e., **update handler**, listening to push notifications from the app store for apps present in its app cache. When a notification arrives, the AppSachet sync service fetches the updates.

App stores transmit app as full apks to end clients devices but updates are transmitted either (i) as full apks if there exists is a wide gap in version of app installed on device vs. app version that is currently available app store or (ii) as incremental updates [22] which are binary diff of previously installed app apk and the current version of apk submitted by developer at app store. AppSachet supports incremental updates to end clients and also handles incremental updates for its own cache. To ensure correctness of incremental updates, app cache follows a 2 phase commit approach i.e., it commits an update to the app cache only when there are no current users installing the app or its update to avoid misalignment of app versions, but once committed, the update is immediately available to eBox connected devices.

A push-based approach to app cache updates, allowing app stores to dynamically push apps or their updates to a device, could leverage global context, e.g., trending apps, important updates, etc. However, given that our current implementation is limited by the existing unoffical Google Play API, AppSachets are restricted to a pull-based approach explicitly requesting apps and updates from the store.

The Sync component is also responsible for aggregating and propagating to the app store notifications about delivery of an app or an update. These notifications are sent asynchronously to app stores to avoid causing slowdowns in AppSachet-end user device interaction but still ensuring consistency in the versions of apps installed on end user device and what is known to remote app-stores. The choice of lazy and asynchronous reporting to remote app stores by eBoxes ensures that devices are not burdened to communicate with remote app stores. It also avoids making remote interactions between eBoxes and app stores a bottleneck while eBox updates are ongoing. However, this may be problematic for apps that require payments. We posit that to support paid apps on AppSachets app stores, either this communication would have to be made synchronous or the eBox must be enabled to process payments. We believe there are additional challenges related to authorization and authentication of eBoxes, which we plan to explore in our future work.

AppSachet relies on app store-resident functionality to provide the aforementioned callbacks or eBox-initiated sync operations, and leverages app profiles and other information gleaned from eBox usage patterns to guide the distribution of app updates across eBoxes, or to otherwise allow app stores to benefit from the presence of eBoxes in the end-to-end app ecosystem. Even though this paper has not yet explored challenges concerning the efficient operation of an eBox-App Store interface, we believe that with ∼100 apps installed on a average device [3, 2] and an update cycle of 10 days, there are significant opportunities to reduce considerable overhead from app stores. By using eBox based app stores, congestion is reduced by (i) providing flexibility in scheduling app store interactions and updates, and (ii) by distributing the app and app update delivery load across a number of eBoxes, which then can handle per device installs/updates.

1.3.4 AppSachet Client

The AppSachet client resides deep in the Android's app framework on the end user device. It is responsible for starting the bootstrapping process when a device first connects to an AppSachet enabled eBox. Mechanisms like Wi-Fi beacons or a central registry based service discovery etc. can be used to kick-off bootstrapping. However, our current AppSachet client prototype does this by listening to wpa supplicant connection notifications and simply querying a AppSachet server running on pre-defined IP:Port combination. A similar approach is deployed on most Wi-Fi routers that provide the control panel of that router over a predefined address, e.g., 192.168.1.0 etc.

On connection, it establishes an eBox's integrity as a valid supplier of apps and/or updates by requesting a CA issued certificate from eBox. On successful verification, the device resident AppSachet client shares an anonymized app profile with eBox. After successful completion of the bootstrapping, the client component is also responsible for requesting and acknowledging individual apps and/or updates from eBox by choosing from those available in list shared by an eBox as app-profile.

The AppSachet client also includes a **App Usage Monitor** interfaces with Android's package manager to get the list of installed apps and uses native hooks to app usage APIs [1] to create anonymous app profiles, stored in a separate file on the device's file system. It is run lazily in the background when the device is locked by the end user. The decision to invoke the app usage monitor lazily ensures that (i) mining relevant information doesn't impact user experience when the device is being actively used and (ii) utilizes the period between user locking the device and system's decision to put device in a deep sleep state to minimize its its impact on device's battery usage. The app-profile is anonymized by passing it through a filter to ensure that information shared with eBox is clear of any personal information. In the current prototype, this simply removes keywords provided by users in their preference, but better anonymization techniques could be deployed for improved privacy guarantees.

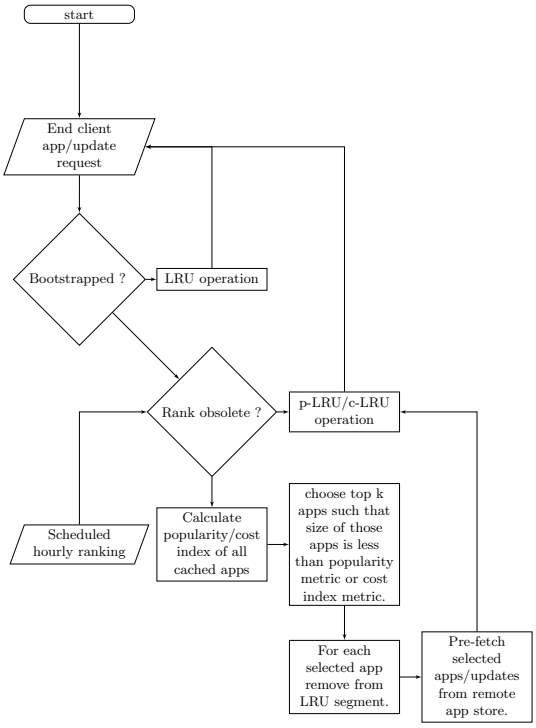

Fig. 1.3. Showing the operation of AppSachet on eBoxes.

1.4 AppSachet Cache Policy Design

We present the two novel cache policies for managing the cache of apps and app updates on AppSachet eBoxes. Policies are specifically defined based on opportunities observed from the app-traffic characteristics captured in our measurements. The two policies – p-LRU and c-LRU – are described next, and the overall description of the cache management operations with either policy follows the same operating flow illustrated in Figure 1.3.

1.4.1 Popularity-aware Caching: p-LRU

p-LRU cache is designed to operate based on app popularity, observed as an important characteristics of app traffic. p-LRU divides the available storage space for caching in two parts: (i) LRU based and (ii) popularity based. The size of each segment is decided based on popularity metric which is defined as percent of storage space allocated to popular apps on an eBox. p-LRU cache is similar to a segmented LRU (SLRU) [19] in the way it keeps two separate

segments of cache, but differs in the eviction strategies in the LRU-vs. the popularity based segment. The p-LRU cache works as follows:

During p-LRU bootstapp period, e.g., the first 24 hours, p-LRU acts as simple LRU. After that, apps are ranked according to the apps that were accessed in the past 24 hours based on the number of times they were accessed or *popularity metric*. For instance, if we have a cache of size 1 GB, and we see that 40% of apps are being accessed repeatedly, we set popularity metric at 40%. This will result in reserving 40% storage space, i.e., 400 MB, for storing popular apps and 60%, i.e., 600 MB, for storing recently used apps.

The popular apps and their updates are then pre-fetched until the popular segment is full. If the app is present in both LRU and popular segment, it is kept in the popular segment, so that LRU can accommodate more apps. Note that there are many apps that although not popular, not caching them would result in a considerable reduction in hit ratio, also highlighted by the gap in all apps and popular apps in Figure 1.4(ii). Since, there are considerable number of apps that are often not popular but not caching them would result in a considerable reduction in hit ratio also highlighted by the gap in all apps and popular apps in Figure 1.4(ii). Once p-LRU is bootstrapped, app ranking is repeated every hour and popular apps are pre-fetched for that hour.

1.4.2 Cost-aware Caching: c-LRU

Similar to p-LRU cache, **c-LRU cache** divides the available storage space for caching in two parts: (i) LRU based and (ii) cost of caching based. It uses a cost index to quantify the cost of caching an app on eBox. Intuitively, the cost of caching can be derived from the following metrics: (i) The number of times it is downloaded when compared to all the apps downloaded from that eBox or the *download ratio*; (ii) the time for which a particular app is kept on eBox's storage compared to its first download or *utilization ratio*; (iii) the time an app has already spent in the cache without actually being requested by end users or *recency ratio*; and (iv) the size of the app that needs to be stored. e.g., if any particular app whose size is 50 MB and is accessed 10 times and we have two other apps whose sizes are 20MB and 30MB, and are accessed 5 and 8 times respectively in the same interval, then we should give preference to caching the two smaller apps than one large app. One exception to this rule is that c-LRU must handle updates and installs separately because updates are always smaller than installs and this would lead to installs never being cached on eBox. We started with giving equal weights to each metric, and the value of each is normalized i.e., varies from 0 to 1. The app with the lowest cost caclulated this way is considered the most suitable one for caching at an ebox. After experimenting with different combinations of weights and metrics, we zeroed to the below mentioned definition of cost index of an app stored on eBox:

Cost index $= [DR * (1/Appsize) + UR + RR)]^{-1}$, where,
Download ratio (DR) = number of downloads of that app / total number of downloads
Utilization ratio (UR) = hours spent in cache / hours since first download
Recency ratio (RR) = 1/hours since last download

The lower cost index results in lower cost associated with storing and hence, higher benefit, because the app may be accessed too frequently or uses very little space or a combination of both. The c-LRU cache works as follows: during c-LRU bootstrapp process, i.e., the first 24 hours, c-LRU acts as a simple LRU. After that, apps are ranked according to the cost index of apps accessed in the past 24 hours. The segmentation, pre-fetching and eviction in c-LRU work similarly to p-LRU except the use of cost index vs. popularity.

The cost function described above tries to maximize the utilization of eBox resources. However, the model permits for additional cost functions, including ones that incorporate consideration of different value generated from different apps. The ability to attach a value to an app in case of AppSachet or generally a service running on an eBox can pave the way to creating a quantifiable economic model for the upcoming 'edge cloud' infrastructure, a concern of utmost importance regarding edge cloud deployment, which hasn't been addressed in any of the recent edge cloud research [16, 18, 20, 12].

1.5 AppSachet Implementation

The implementation of AppSachet uses either available Android platform components or open source technologies. Specifically, (i) the eBox-resident elements are implemented using the node.js and python API on top of an OpenWRT router, (ii) the device-side AppSachet client elements are implemented as a patch for Android, and (iii) the additional elements of the eBox-app store interface are implemented using the unofficial HTTP API of the Google Play Store. With our limitations to evaluate eBox-AppStore interface owing to it requiring changing app-store's internals and its interface, we present detailed evaluations of the other components of AppSachet mechanisms next.

1.6 Evaluation

1.6.1 Experimental testbed

App traffic mesurements are obtained from a network tap that has the capability of logging all traffic flowing in and out of our institution. We used offline analysis to filter the data after logging. The AppSachet is deployed on an eBox emulated with a Core2Duo machine housing apps in its local storage

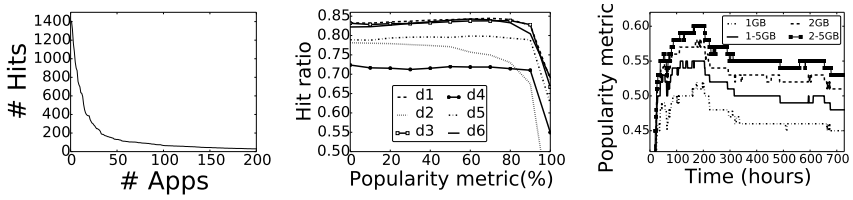

Fig. 1.4. For the measured app traffic showing (i) the number of times each app is accessed in the complete dataset; (ii) temporal caching characteristics of app traffic workload by dividing observed traffic in 6 small periods; (iii) number of popular apps vs. other apps accessed from the cache per hour;

and connected to a Linksys wrt 1900ac router via a Gigabit port. We generate a representative app traffic workload for our experiments using captured app traffic. The AppSachet client is prototyped using a Nexus 5 phone running Android (CyanogenMod 11.2 $\sim$ Android KitKat).

1.6.2 Cacheability of app traffic

Figure 1.4(i) shows the number of times a particular app or its update in accessed from the measured app traffic. The most popular app was accessed 1403 times and then access frequency decreases exponentially. Specifically, the 100th app was accessed only 68 times, showing a clear long-tail distribution of apps and their updates. To gain insights into finer temporal cache characteristics of app traffic, we divided the complete dataset into 6 equal smaller periods – where each line in Figure 1.4(ii) d_i corresponds to a different period – and found that the caching characteristics persist for small periods as well as for the overall traffic trace.

Going a step further, we analyse the observed app traffic on a per-hour basis to capture local popularity of apps on an eBox based cache. Figure 1.4(iii) shows that for every hour, 40%-60% of apps are accessed from what we call the local popular app cache. We notice that every hour, the local popularity of apps on an eBox changes, requiring hourly updates to keep the app cache clear of outdated apps, also seen from the pattern in the Figure 1.4(iii). *These results establish that traffic due to apps and their updates is suitable for caching on an hourly basis and provided justification for our rationale behind the design of the p-LRU and c-LRU caching policies.*

1.6.3 p-LRU and c-LRU Cache Performance

We compared the proposed p-LRU and c-LRU with a number of popular cache policies, i.e., LRU, Random and Belady's optimal eviction policy. The experimental results, summarized in Figure 1.5(i) and Table 1.2 are obtained using eBoxes with up to 2.5GB of cache storage.

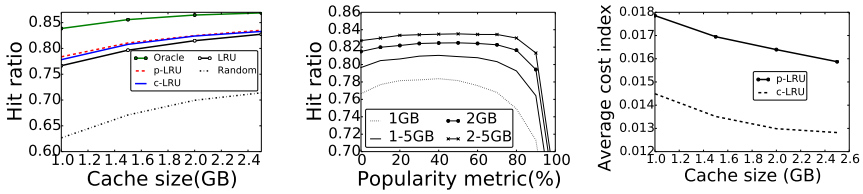

Fig. 1.5. For the measured app traffic showing (i) the comparison of caching policies; (ii) p-LRU cache performance with different size of eBox based cache; (iii) the average cost index observed (i.e., by all the apps stored on eBox at a time) when using p-LRU and c-LRU caching policy while varying cache size.

It is clear that p-LRU outperforms all other policies and is closest to the optimal cache closely followed by c-LRU policy. Figure 1.5(ii) shows the overall performance of p-LRU cache for varying sizes of app cache which shows that the best ratio is obtained when the cache size for popularity metric is between 40%-60% which drops drastically after 80%. This also shows why one segment must be assigned as a LRU cache, i.e., LRU also plays a very important role in maintaining a high cache ratio whereas the popularity metric or cost index ensures that the popular apps or apps with high cost index are always cached, even on their first access. *We conclude that efficient use of the capabilities of upcoming edge cloud platforms (e.g., for caching) would require defining new application specific metrics (e.g., popularity, cost) and/or implement new mechanisms (e.g., p-LRU, c-LRU).*

	Cache Size	1 GB	1.5 GB	2 GB	2.5 GB
Cache Policies	Oracle	0.8386	0.8558	0.8647	0.8688
	p-LRU	0.7837	0.8105	0.8247	0.8352
	c-LRU	0.7782	0.8078	0.824	0.8328
	LRU	0.7665	0.7965	0.8149	0.8274
	Random	0.6266	0.671	0.6994	0.714

Table 1.2. For the measured app traffic, showing comparisons of cache policies with varying cache sizes.

1.6.4 Storage Requirements on eBox

Figure 1.5(ii) shows the variation of cache hit ratio of the p-LRU cache with increasing cache size. Figure 1.5 shows that using a p-LRU cache on an eBox with a capacity of 2.5 GB results in the highest hit ratio; this is also closest to the optimal Belady's algorithm shown as Oracle. We conclude that with 2.5 GB of additional storage at eBoxes and a p-LRU cache, AppSachet achieves a 83% hit ratio. Figure 1.5.(iii) shows the average cost index observed, i.e., average of cost indexes of all apps stored on the eBox, updated hourly, while

running through the complete workload and using p-LRU and c-LRU. As apparent, c-LRU beats p-LRU consistently in terms of lower cost index and hence, lower cost of caching resulting in higher benefits, while still slightly sacrificing hit ratio as seen from Table 1.2 This highlight a trade off in cache performance vs. cost particularly for eBoxes with less storage due to cost constraints. *Generally, for edge services (e.g., AppSachet) deployed on edge cloud platforms with resource constraints (e.g., storage capacity), designing mechanisms (e.g., c-LRU policy) must consider other factors (e.g., cost) vs. just performance (e.g., hit ratio).*

1.6.5 Cost benefit analysis of deploying eBox based AppSachets

Without real world deployments of eBoxes and in the absence of any real cost models for eBox revenue, we base our cost-benefit analysis on the retail cost of SSD storage and the benefit of the latest pricing information about content delivery networks prices. Simply put, the benefit from an AppSachet on an eBox is directly proportional to the reduction in volume of traffic served by an eBox. Consider the following:

1. There is a wide range on prices offered by CDNs[6], e.g., typically \$0.01 per GB to \$0.05 per GB depending on the volume of traffic.
2. Additional storage cost of 2.5 GB flash storage varies from \$3-\$20 based on its quality. Assuming that we also add 2GB DDR3 RAM as well to the eBox, which costs anywhere from \$10-\$20, this would result in a maximum increase of \$40 in eBox cost.
3. Based on the size of app installs/updates in §1.2, the total amount of bytes served by an app store are ∼2.6TB. With a 83% hit ratio shown to be achieved by p-LRU cache would serve ∼2TB from eBoxes.

Conservatively, using \$0.01 per GB, an eBox can save (\$0.01 x 2000 = \$20) in three months, i.e., an eBox would be able to recover the additional cost of storage within 3 months of its deployment. Even if we consider the additional RAM as a cost increase in eBoxes, it will be recovered in the first 6 months of eBox deployment. *From this, we want to highlight the value proposition of edge cloud based services (e.g., AppSachet) in terms of reduced operational costs for cloud based services.*

1.7 Discussion and Future work

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

eBox Deployment Model. Given that realworld deployments of eBoxes don't exist as yet, there are open questions about their ownership – individuals, businesses or public infrastructure? Security and Trust aspects of apps

from AppSachet on those eBoxes? Another open area is DRM of the app cache on eBox. We posit that authentication and authorization methods can be deployed on eBoxes, theoretically but those authorization and authentication assume a human user which is authenticated or authorizes which is not the case AppSachet operating on eBox. We believe that this is an interesting problem which plan to pursue in future.

Privacy Concerns. Privacy concerns arising out of sharing app profiles and methods to anonymize app profiles leaves out an open question – First, is it possible to fingerprint users based on app users and if so, what obfuscation methods can be applied to avoid those concerns. However, it is important to note that sharing app profiles is not invasive than use current app usage API in Android [1] (which AppSachet also uses) which lets developers to track app usage. However, this aspect certainly needs a detailed evaluation.

System software on devices and eBoxes. Without any standard definition of am eBox, their deployment mechanism and consequently new functionalities, e.g., app caching etc., provide an wide open space for research. In our future work, we are exploring additional functionality that can further improve the app ecosystem and their automatic provisioning eBoxes.

Device side evaluation. We also carried out experiments to gauge the benefit of AppSachets on end client devices. However, we did not observe any significant benefits or any new finding other than what is already reported in our previous work [11] so we omitted those results from this paper.

1.8 Related Work

Previous work on characterizing Internet traffic workload has mainly focussed on video e.g., youtube access patterns [13, 23, 14, 21, 17] etc, and web but, there has been no work done in collecting the android app access patterns. Our paper is the first of its kind to capture and analyse the download behaviour of android apps. Using dynamic caching for prefetching content, Gandhi et al [17] suggested that k-means clustering used more intervals while reducing error rate compared to dynamic programming. However, based on our android app access pattern, k-means clustering gave a cache-hit ratio of 75%, which is lower than the cache-hit ratio of the proposed p-LRU algorithm. Zink et al [23] observed a similar pattern for youtube videos and proposed a caching policy based on LRU and popularity of a movie. The results showed an improvement in the cache-hit ratio. However, their algorithm depended on a global list of popular movies. Access to a global list may or may not be there. Also, it might happen that the global popularity list might differ from local lists [23], where they proved that there is no strong correlation is observed between global and local popularity and video clips of local interest have a high local popularity. The p-LRU algorithm addresses these issues, as it generates the popularity list by learning the access patterns locally.

Prior efforts have considered support app execution via edge-cloud platforms [20, 16, 18, 12] to leverage resource-rich execution environment to partially or fully offload app execution from resource constrained devices. But none of them considered use of edge cloud platforms for app delivery which is the focus of AppSachets approach. Recent work to reduce mobile app update traffic proposes micro app updates[15, 22] which would complement AppSachets.

1.9 Conclusions

AppSachet is a distributed app delivery system for the Android ecosystem that shows that deploying proposed app caches (p-LRU, c-LRU) on eBoxes in the 'edge cloud' tier can recover the already modest additional costs within 3 months of its deployment. The design of AppSachet is based on extensive experimental measurements of app traffic and keeping in mind practical deployment concerns, i.e., not requiring changes to apps by developers and/or changes in how end users employ these apps. More generally, we conclude that that while moving conventional services to the edge cloud can have benefits in terms of latency and bandwidth but designing services for edge cloud platforms requires more than just running existing backend cloud services in the edge cloud. There remains interesting tradeoffs to be explored and new mechanisms to be developed leading to efficient use of future edge clouds providing a fertile ground for systems research.

Acknowledgement

This work was partially supported through research grants from Intel, VMware, and NSF CNS1148600.

References

1. Android app usage api @ https://goo.gl/39dqlu.
2. Android apps per device - yahoon avaite @ http://goo.gl/3zb3ob.
3. Android apps per device @ http://goo.gl/zkaxsx.
4. Att small cell deployment plans. `http://goo.gl/XdfkfH`.
5. Att wifi hotspot locations. `http://goo.gl/XdfkfH`.
6. Cdn pricing 2014– http://goo.gl/7l7fkd.
7. Level 3 cdn reports last mile as new bottleneck @ http://goo.gl/3ir9kg.
8. Mobile world congress -small cells. `http://goo.gl/cWaARN`.
9. Qualcomm small cells. `http://goo.gl/HEpudP`.
10. Qualcomm smart gateways. `http://goo.gl/BwPc7f`.
11. Bhardwaj, K., Agarwal, P., Gavrilovska, A., Schwan, K., and A., A. Appflux: Taming mobile app delivery via streaming. In *2015 Conference on Timely Results in Operating Systems (TRIOS 15)* (Monterey, CA, USA, 2015), USENIX Association.

12. 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.
13. Braun, L., Klein, A., Carle, G., Reiser, H., and Eisl, J. Analyzing caching benefits for youtube traffic in edge networks x2014; a measurement-based evaluation. In *Network Operations and Management Symposium (NOMS), 2012 IEEE* (April 2012), pp. 311–318.
14. Cheng, X. Understanding the characteristics of internet short video sharing: Youtube as a case study. In *Procs of the 7th ACM SIGCOMM Conference on Internet Measurement, San Diego (CA, USA), 15* (2007), p. 28.
15. Cheung, A., Ravindranath, L., Wu, E., Madden, S., and Balakrishnan, H. Mobile applications need targeted micro-updates.
16. 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.
17. Gandhi, A., Chen, Y., Gmach, D., Arlitt, M., and Marwah, M. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proceedings of the 2011 International Green Computing Conference and Workshops* (Washington, DC, USA, 2011), IGCC '11, IEEE Computer Society, pp. 1–8.
18. 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).
19. Karedla, R., Love, J. S., and Wherry, B. G. Caching strategies to improve disk system performance. *Computer 27*, 3 (Mar. 1994), 38–46.
20. Koukoumidis, E., Lymberopoulos, D., Strauss, K., Liu, J., and Burger, D. Pocket cloudlets. *ACM SIGPLAN Notices 47*, 4 (June 2012), 171.
21. Krishnappa, D. K., Khemmarat, S., Gao, L., and Zink, M. On the feasibility of prefetching and caching for online tv services: A measurement study on hulu. In *Proceedings of the 12th International Conference on Passive and Active Measurement* (Berlin, Heidelberg, 2011), PAM'11, Springer-Verlag, pp. 72–80.
22. Samteladze, N., and Christensen, K. Delta: Delta encoding for less traffic for apps. In *Proceedings of the 2012 IEEE 37th Conference on Local Computer Networks (LCN 2012)* (Washington, DC, USA, 2012), LCN '12, IEEE Computer Society, pp. 212–215.
23. Zink, M., Suh, K., Gu, Y., and Kurose, J. Watch global, cache local: Youtube network traffic at a campus network: measurements and implications. vol. 6818, pp. 681805–681805–13.