A Case for Web Service Bandwidth Reduction on Mobile Devices with Edge-hosted Personal Services

Yongshu Bai, Pengzhan Hao, and Yifan Zhang
Department of Computer Science
SUNY Binghamton
{ybai4, phao3, zhangy}@binghamton.edu

***Abstract*—We show that many popular mobile services suffer from excessive network bandwidth consumption. The root cause is that the existing mobile/cloud communication interfaces are designed and optimized for service providers rather than end user devices. Solving the problem is challenging, because of the conflicted interests of service providers and mobile devices. We propose Edge-hosted Personal Service (EPS), with which device-oriented solutions can be easily deployed without affecting service providers. EPS also enjoys other notable advantages, including enabling new mobile services, reducing loads on the cloud, and benefiting delay-sensitive applications. We demonstrate the usefulness of EPS by designing *ETA* (Edge-based web Traffic Adaptation), an effective solution for the excessive bandwidth consumption problem, and deploy ETA with a prototype EPS system. By exploring lightweight virtualization techniques, our EPS prototype system is highly scalable in terms of concurrent EPS instances, and secure in terms of resource isolation. The real-world evaluation shows that our *ETA* EPS can effectively reduce bandwidth for mobile devices with small overheads.**

I. Introduction

Web technologies are important for mobile service providers to deploy their services on to mobile devices. For example, most cloud-based services use HTTP(S) as the application layer protocol to carry their service contents between cloud servers and mobile devices. As a result, efficiencies related to web service on mobile devices, such as network efficiency and energy efficiency, play an important role in providing good user experience for mobile devices. However, we observe that the *current web based interfaces used by mobile service providers can incur significant excessive bandwidth consumption on mobile devices*. In this paper, we investigate *reducing the bandwidth requirement of web services on mobile devices* with *practical* solution.

We briefly introduce the two observations motivating this work in the following, and will elaborate later in §II.

• First, we find that many popular web based cloud services incur what we call *hifasp* (i.e., *hi*gh *f*requency *a*nd *s*mall *p*ayload) traffic on mobile devices, and most of the traffic is unnecessarily spent on transmitting repetitive or ineffective contents. Specifically, these web services, which usually deal with frequent content and state updates, generate a high volume of HTTP(S) messages in a short amount of time, and the payloads of these messages are small compared to the message sizes. As a result, the HTTP(S) message headers account for a majority portion of the traffic generated. In the meantime, due to the statelessness property of the HTTP(S) protocol, most of the header fields are the same across different messages, causing significant unnecessary network bandwidth consumption on repetitive contents.

For example, to provide instant backup and seamless collaboration experience, Google Docs [1] aggressively synchronizes (e.g., once per second) document changes when user is editing a file. According to our measurement, when a user is typing with the Google Docs app in a normal speed, about 2 KB of web traffic is generated for *each* character input. Among the traffic generated, close to 80% is for transferring repetitive contents in the HTTPS message headers. We have measured several additional well-known online document editing services, as well as other popular cloud-based mobile services, such as instant message and ride sharing, and obtained the similar observation.

• Second, we observe that the RESTful development APIs, which are commonly used in many web-based cloud services, can cause considerable management and network traffic overheads for mobile apps. These overheads arise from the statelessness property of the RESTful APIs, which, although facilitates service deployment and scalability, incurs extra bandwidth consumption compared to a stateful design.

The above observations are two examples showing that *existing mobile device/cloud interfaces are designed in a way optimized for service providers, but not for end user devices*. Optimizing for mobile devices in this case is challenging, as the solutions often conflict with the interests of service providers. For instance, stateful designs of communication protocol and development APIs can address the previous problems for mobile devices. But they can also significantly increase design complexity and harm service scalability on the infrastructure side. To address this challenge, we propose to introduce computation/processing entities on the network edge to bridge end user devices and cloud services, such that device-friendly designs can be easily deployed for mobile devices, while keeping the original infrastructure-friendly interfaces for service providers. The resulting architecture is called Edge-hosted Personal Service (EPS), which runs different computation/processing entities (called EPS instances) on edge nodes to provide different functionalities for different users. EPS is not only ideal for deploying device-oriented optimizations without harming the interests of service providers, it also can enable many new services, reduces processing server loads for service providers to keep up with the exploding demand, and benefits a wide range of delay sensitive applications.

Enabling EPS in practice is challenging, mainly because of two reasons. *First*, since each EPS instance is supposed to serve an individual user for a specific functionality, it is desirable to run a large number of EPS instances on an edge node at the same time, which is challenging for embedded device edge nodes with constrained computation resource. *Second*, since different users' data need to be processed on the same edge node, the ability of secure resource isolation among EPS instances is important to protect user data security and privacy. We have explored lightweight virtualization techniques to address the challenges. We will demonstrate our effort by first presenting the design of *ETA* (Edge-based web Traffic Adaptation), which is our solution of addressing the excessive mobile device bandwidth consumption problem for web services with *hifasp* traffic. *ETA*'s main idea is to enable stateful and HTTP(S)-compliant communication on mobile devices to avoid transmitting the lengthy and repetitive contents in HTTP(S) message headers. Then we introduce our experiences of implementing *ETA* and deploying it with EPS in a lab environment. Our EPS edge implementation was carried out on an x86 based laptop and an ARM development board, both of which normally functioned as WiFi access points. We explored two lightweight virtualization techniques, VM-based unikernel and OS container, in implementing the EPS edge runtime framework. The experiment evaluation of our prototype system shows that *ETA* is able to effectively reduce mobile device bandwidth consumption for web services with *hifasp* traffic, and that our lightweight virtualization based EPS runtime framework can efficiently support a large number of concurrent *ETA* EPS instances with small overheads.

In summary, the contributions of this paper are as follows.

- We experimentally demonstrate the problem of excessive mobile device network bandwidth consumption in many popular web-based mobile services that generate high frequency and small payload (*hifasp*) traffic.
- We analyze that the root cause of the above problem, which is the conflict between the interests of mobile devices and service providers. We propose Edge-hosted Personal Service (EPS), a novel architecture, which can not only solve the problem for mobile devices without affecting service providers, but also enjoy several other notable advantages.
- We design *ETA*, an effective solution of solving the excessive network bandwidth consumption problem for mobile devices.
- We implement a prototype EPS system exploring lightweight virtualization techniques on two platforms based on x86 and ARM respectively. We deploy the *ETA* solution with our prototype EPS system, and evaluate with real-world experiments.

II. Motivation Study

Web based interface is the most commonly used approach for mobile service providers to connect their services to the users. However, we observe that the existing web interfaces used on mobile devices can cause significant network bandwidth consumption, and therefore harming end user experience (e.g., shorter battery life, greater financial cost). We detail these two observations in the following in §II-A and II-B.

A. Observation 1: hifasp traffic is common for many web services, and it has significant amount of repetitive contents.

Due to the highly mature and widely available world wide web infrastructure, HTTP(S) is predominantly used as the application layer network protocol to carry service contents between mobile service servers and end user devices. We observe that many cloud based mobile services (e.g., online document editing, instant messaging, and ride sharing) generate HTTP(S) messages with *hi*gh *f*requency *a*nd *s*mall *p*ayload (short as *hifasp*), and there are a considerable amount of *repetitive* and *ineffective* contents in those *hifasp* traffic, causing unnecessary bandwidth consumption on mobile devices. Briefly speaking, the direct causes of the observation are as follows.

- The *hifasp* traffic arises from one common characteristic of the services, which is they all deal with frequent content and state updates.
- The *repetitive* contents are caused by the stateless nature of the HTTP(S) protocol, which requires every HTTP(S) message to contain all the necessary information (e.g., authorization info, cookies, service/app specific info) to allow the server to keep track of the ongoing communication session without storing any state information about the client.
- The *ineffective* contents are caused by the inability of web services to adjust the contents of HTTP(S) messages according to the capabilities of the clients they are communicating with.

According to our study, the average ratio of repetitive/ineffective content is usually larger than 50%, and for some services it can be as high as 85%. Next, we take Google Docs [1], a highly popular online word processing web service, as an example to give an in-depth discussion of our observation.

TABLE I
Web traffic generated for typing a paragraph of 1,016 characters using the Google Docs Android app.

		On-screen keyboard	Bluetooth keyboard
Total HTTPS req/rsp msg pairs		806	527
HTTPS msg pair for *each* char typed		0.79	0.52
Req HTTPS msgs	Avg msg size	1,025 bytes	1,034 bytes
	Avg msg payload size	138 bytes	146 bytes
	Avg msg payload ratio	0.13	0.14
Rsp HTTPS msgs	Avg msg size	1,373 bytes	1,375 bytes
	Avg msg payload size	146 bytes	148 bytes
	Avg msg payload ratio	0.11	0.11
Total HTTPS traffic generated		1.94 MB	1.26 MB
HTTPS traffic for *each* char typed		1.96 KB	1.27 KB

Google Docs web traffic is *hifasp*, and the traffic volume is high: Google Docs provides two forms of interfaces for mobile users to use the service: mobile app interface and web browser interface (i.e., a web portal for the service). Both interfaces use HTTPS as the application layer protocol to transfer user document changes between clients and cloud servers. We have performed a series of experiments to investigate the web traffic generated by the Google Docs service on mobile devices. In the first experiment, we asked an everyday smartphone user to normally type a paragraph of English text containing 152 words (1,016 characters in total) into a Google Docs document via the Google Docs Android app. The typing was done using the on-screen keyboard on a smartphone. The smartphone was connected to the Internet via a web debugging proxy, which has the SSL proxying feature allowing us to capture and record the HTTPS traffic generated. The second column of Table I shows the summary of the resulted web traffic. The typing process generated 806 pairs of HTTPS request/response messages, which was high considering that only 152 words were typed (i.e., *the frequency of HTTPS messages was high*). The average sizes of HTTPS request and response messages were 1 KB and 1.3 KB respectively. On average HTTPS message body accounted for only about 10% of the message size for both cases (i.e., *the payloads of the messages were small*). In total, about 1.94 MB of web traffic generated for typing the 1,016 characters, or about 1.96 KB web traffic for *each character* typed, which was quite high considering that it is not uncommon to have thousands or more characters in a typical word document.

The reason that Google Docs web traffic is *hifasp* originates from its goal of enabling instant synchronization of doc changes between clients and the server. Specifically, when the user is working on a document, the Google Docs client (app or web based) updates doc content changes to the server almost instantly, and the server immediately further propagates the changes to all other active clients on the same document. According to our experience, the update interval is around one second. As a result, a high frequency of HTTPS messages are generated when a Google Docs document is being edited, and the content change carried in each HTTPS message is small. This aggressive content change synchronization enables not just instant doc content modification backup, but more importantly also the seamless collaboration experience on shared document between different users, which is one of the main reason for Google Docs' popularity.

Another factor affecting the volume of Google Docs' web traffic on mobile devices is the speed of user's typing. Generally speaking, the faster the user types, the smaller the generated web traffic would be, because more doc content changes can be included in each HTTPS message. We performed the same experiment as described previously, but using a physical Bluetooth keyboard connected to the phone. The third column of Table I summarizes the results. We can see that the average size of request HTTPS messages was slightly larger, and the total number of HTTPS request/response message pairs were smaller, than those in the on-screen keyboard case, because of the reason just discussed. However, the traffic generated for each character typed was 1.27 KB, which is still significant for typing large documents.

The majority of Google Docs' web traffic is repetitive or ineffective: For web-based services whose traffic is *hifasp*, such as Google Docs, HTTP(S) message headers account for the majority of the web traffic. We find that *a large portion of Google Docs' HTTPS message header contents are either repetitive or ineffective*. Specifically,

TABLE II
REPETITIVE AND INEFFECTIVE HTTPS MESSAGE HEADER CONTENT IN THE WEB TRAFFIC OF GOOGLE DOCS.

		On-screen keyboard	Bluetooth keyboard
Req HTTPS msgs	Avg msg size	1,025 bytes	1,034 bytes
	Avg header repet. content size	851 bytes	851 bytes
	Repet. content ratio	0.83	0.82
Rsp HTTPS msgs	Avg msg size	1,373 bytes	1,375 bytes
	Avg header repet. content size	600 bytes	600 bytes
	Avg header ineff. content size	440 bytes	440 bytes
	Repet./ineff. content ratio	0.76	0.76
Overall repetitive/ineffective content ratio		**0.79**	**0.78**

• *Repetitive* contents refer to those message header fields that have the same value in different messages. For example, in upstream synchronization with Google Docs, the header fields of `Authorization` and `User-Agent` in the request HTTPS messages always have the same and long strings (more than 200 bytes each) within the same session. This is because the `authorization` header field carries the authentication credential, which is generated at the beginning of an HTTP(S) session, and remains unchanged throughout the whole session. The `User-Agent` field records the type of the user agent, and thus also keeps unchanged for the same client.

As discuss previously, the cause of the repetitive contents in HTTP(S) headers is the statelessness of the protocol: sender needs to provide all the necessary info (repeatedly) in every packet to allow the receiver to function properly without the need of memorizing sender's states.

• *Ineffective* contents refer to the message header fields that have no use for the HTTP(S) session. They are usually caused by web server's inability to adjust the header fields in the HTTP(S) message they sent out based on the capabilities of the clients they talk to. For example, in upstream sync in Google Docs, a large `Set-Cookie` header field (440 bytes) is included in every HTTPS response message sent by the cloud server, regardless the type of the client (i.e., browser- or app-based). However, this header field does not have any effect for the app based clients, as they do not support cookie.

Table II summarizes the results about repetitive and ineffective contents in our Google Doc web traffic measurement experiments. We can see that both the HTTPS request and response messages in Google Docs' web traffic have a significant amount of repetitive and ineffective contents (i.e., about 80% of all the web traffic generated), which can consume a considerable amount of unnecessary network bandwidth when inputing large documents.

We have performed experiments to examine other Google online office suite products, such as Google Sheets and Google Slides, as well as several other popular online editors, such as Microsoft Word with cloud sync, Dropbox Paper [2], ShareLatex [3], and obtained the similar results. Also, we have evaluated many other popular web-based mobile services with the need of frequent content/state update, such as Google Hangouts (instant messaging), Facebook Messenger (instant

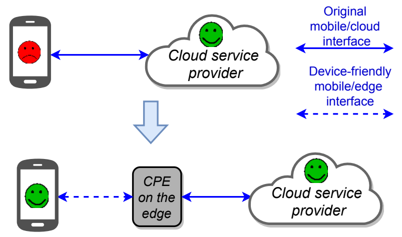

Cloud service provider
Original mobile/cloud interface
Device-friendly mobile/edge interface
CPE on the edge
Cloud service provider

Fig. 1. Solution high level idea: introducing an communication/processing entity (*CPE*) on the network edge to enable device-friendly interface for mobile devices.

messaging), and Lyft (ride sharing) and also obtained the similar observations as those on Google Docs. We skipped the details of these experiment results due to the space limit.

B. Observation 2: RESTful web APIs cause notable management and network traffic overheads on mobile clients.

The previous observation reveals the considerable unnecessary bandwidth consumption caused by repetitive/ineffective contents in HTTP(S) message headers of many web services with *hifasp* traffic. We also find that the payloads of HTTP(S) messages can also be bloated because of the way how the mobile services are deployed. Specifically, many cloud service providers are now offering development APIs, which is a popular way to deploy their services onto mobile devices. For example, popular cloud storage providers, such as Dropbox, Google Drive, and OneDrive, all provide their development web APIs allowing third-party applications to make use of their services. These web APIs are usually designed to be RESTful [4] for many benefits. We emphasize two of them, which are related to our observation:

• With RESTful APIs, third-party app developers can have all the info to determine how the apps should behave based on the responses from the server. This way, service providers can enjoy fast and easy deployment of their services by erasing the need of providing their own client implementation to deploy the services onto mobile devices.

• The characteristics of the RESTful architecture, such as statelessness, enable efficient and scalable services [5].

However, the RESTful web API approach can lead to notable management and network traffic overheads for third-party apps. The overheads arise mainly from the statelessness property of the RESTful architecture, which requires app developers to spend extra effort to take care of the service/app state acquisition and management. For instance, to keep a local file consistent with the cloud copy in a RESTful cloud storage service, app developers need to poll the file state on the cloud side frequently, causing unnecessary network bandwidth consumption. The reason is that the cloud server does not retain any information about client states, and thus it is the app's job to track and manage all the file states.

As a comparison, traditional distributed file systems (DFS), such as AFS [6], [7], Coda [8] and NFSv4 [9], keep callbacks for open files on file servers, so that file changes made by one client can be instantly and efficiently propagated to all other clients opening the same file. However, this stateful design is

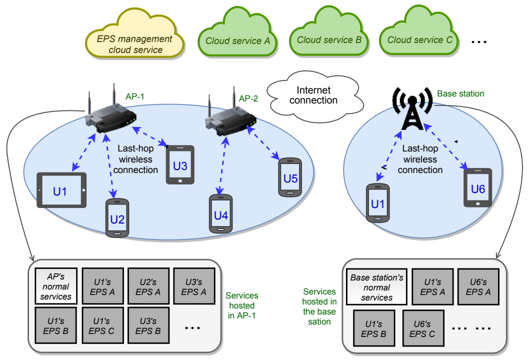


Fig. 2. Overview of Edge-hosted Personal Service (EPS).

not scalable for the existing cloud storage services, which need to deal with a much higher concurrent clients than traditional distributed file systems. Therefore, finding a *balanced solution*, which satisfies both *the need of improving service scalability for service providers* and *the need of improving network bandwidth usage efficiency for mobile devices*, is the central goal of our solution design to be discussed next.

III. Solution Design

A. High level solution idea

The problems demonstrated above share the same *root cause*, which is *web services/protocols are usually designed in a way optimized for service providers/servers, but not for end user devices*. As discussed previously, it is *challenging to design a communication scheme, which is optimized for both mobile service providers and end user devices at the same time*. In addition, it is also *challenging to design a solution that can benefit the existing mobile service providers and devices by requiring no or minimal changes on both sides*.

Our idea of addressing the above challenges (illustrated in Figure 1) is to introduce communication/processing entities on network edge (e.g., WiFi access points, cellular base stations, dedicated edge servers) between mobile devices and service providers, such that device-friendly interfaces (e.g., communication protocol with small bandwidth requirement, traditional DFS-like file management for mobile devices) can be enabled between mobile devices and the edge, while keeping the original cloud-friendly mobile device/cloud interfaces for the service providers. This way, we can effectively address most of the challenges discussed above, except for the no/minimal change to mobile device requirement, which we solve with a set of engineering techniques described later in §IV.

B. Edge-hosted Personal Service (EPS)

We propose Edge-hosed Personal Service (EPS), an edge-assisted mobile-cloud computing architecture, where different computation/processing entities (called EPS instances) are running on network edge to provide different functionalities for individual users. Figure 2 shows an example scenario of EPS architecture. In the example, three cloud services `A`, `B` and `C` are utilizing the EPS infrastructure. Two local wireless networks are illustrated: a WiFi network with two access points, and a cellular network within one base station. In this case, the edge nodes are the two APs and the base station, in which EPS instances are running to serve the mobile users within the local wireless network. Several key EPS properties are: (1) An EPS instance is to serve an individual user for a specific purpose. (`U1`'s EPS `A` in AP-1 is for helping user `U1` use the cloud service `A`.) (2) A user can start multiple EPS instances on the edge node for accessing different cloud services. (User `U1` is running three EPS instances to access the three cloud services respectively.) (3) An EPS instance is created/destructed on demand as the user starts/stops accessing the corresponding cloud service. (4) An EPS instance is migratable across the edge nodes as the user is roaming within the same network. (5) With the help from the EPS management cloud service, EPS instances (e.g., those storing user data) can be saved on their destruction, and can be resumed later in the same or a different edge node. (6) An edge node needs to provide its normal services (e.g., routing for wireless routers) in addition to running EPS instances.

EPS advantages: EPS has many advantages. For example, as discussed previously, EPS can naturally facilitate the deployment of device-oriented optimizations, such as our mobile device bandwidth reduction solution to be introduced later, without harming the interests of service providers. Meanwhile, many promising applications and optimizations can be enable with EPS, such as mobile devices computation edge offloading, edge-based web caching and optimization for mobile devices, and new edge-based services targeting users in a local area. Also, with EPS, cloud services can be easily distributed to edge nodes to reduce the load pressure on cloud servers, making it much easier for service provider to keep up with the rapid demand growth. In addition, the short distance between mobile devices and edge nodes greatly reduces EPS response time. Thus, many delay-sensitive mobile applications can effectively take advantage of EPS for enhanced performances.

EPS challenges: Enabling EPS in practice is not trivial. Two important challenges need to be addressed. The *first* challenge arises from the need of supporting tens or even hundreds of concurrent EPS instances (i.e., in the same order of magnitude as the maximum possible number of users supported by the edge node), which is challenging considering that many edge nodes are embedded devices with constrained computation resources. The *second* challenge is that, since EPS instances of different users are running on the same edge node, securing sensitive user information handled by EPS instances and preserving user privacy are critical.

To address the above challenges, we explore lightweight virtualization techniques to enable lightweight and secure EPS runtime on edge nodes (details in §IV). We have designed and implemented an EPS, which is used to deploy our solution, named *ETA* (Edge-based web Traffic Adaptation), for addressing the problem of excessive mobile device network bandwidth consumption for web services with *hifasp* traffic (§II-A).

C. The design of ETA

Recall that the excessive mobile device network bandwidth consumption shown in §II-A originates from two sources: one is the significant amount of repetitive contents in HTTP(s) message headers, and the other is the ineffective contents in HTTP(S) message headers. Therefore, the approach of *ETA* to solve the problem is twofold: (1) enabling a stateful and HTTP(S)-compliant communication between mobile devices and *ETA* EPS instances, such that the lengthy and repetitive contents in HTTP(S) message headers can be converted to the concise states shared between the two sides; and (2) adjusting header contents of the messages sent from EPS instances to mobile devices, based on the mobile devices' capabilities observed in their messages.

Enabling stateful communication between mobile devices and *ETA* EPS instances: We use a simple example to explain *ETA*'s approach. Suppose in an online document editing web service, only two header fields `A` and `B`, and two other header fields `C` and `D`, are used in the HTTP request messages and response messages respectively. Figure 3 shows a scenario of a mobile device updating document changes to cloud servers using the HTTP `POST` method. When editing the same document, the header fields in both request and response messages are kept the same (which is close to what we observed in Google Docs). As a result, due to the *hifasp* nature of the web traffic, the majority of the mobile device network bandwidth is consumed on transmitting the repetitive header contents in the headers of the HTTP messages.

With the help from EPS instances, *ETA* converts the stateless HTTP(S) protocol to an equivalent but stateful communication protocol, which is called *ETA* protocol, for mobile devices. The idea is to allow the mobile device and its communication counterpart, which is an EPS instance, to synchronously record the repetitive header contents on their own key-value stores, so that the lengthy and repetitive headers contents in the original HTTP messages can be adapted to the concise keys in *ETA* messages. Figure 4 shows how *ETA* reduces mobile device bandwidth consumption for the example scenario. Before the first HTTP `POST` message leaves out of the device, it is intercepted by *ETA*, which finds that the contents of the all the header fields shown in the message do not exist in the device *upstream* key-value store. Therefore, the contents of those header fields are added to the device upstream key-value store (step ①). Then *ETA* sends out the original HTTP message, with the keys piggybacked to the corresponding header field contents in the message (step ②). Upon receiving the first HTTP `POST` message from the device, the EPS instance adds the key/value pairs shown in the received message to its own upstream key-value store (step ③). The EPS instance then removes the piggybacked keys from the received message to restore the original `POST` message, and sends it to the cloud server (step ④). When receiving the response message from the cloud server (in step ⑤), the EPS instance finds all the header fields in the message do not exist in its *downstream* key-value store. Thus it builds the key-value store entires accordingly (step ⑥), and sends the response message with downstream keys piggybacked to the device (step ⑦). The device propagates its downstream key-value store according to the information in the response

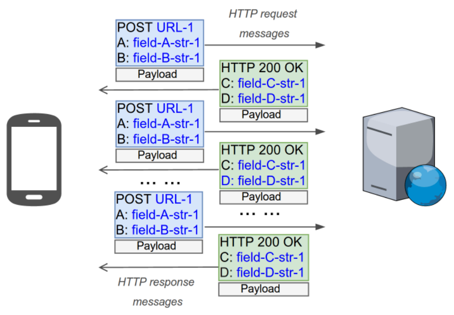


Fig. 3. An simplified example of the excessive mobile device network bandwidth consumption problem for web services with *hifasp* traffic.

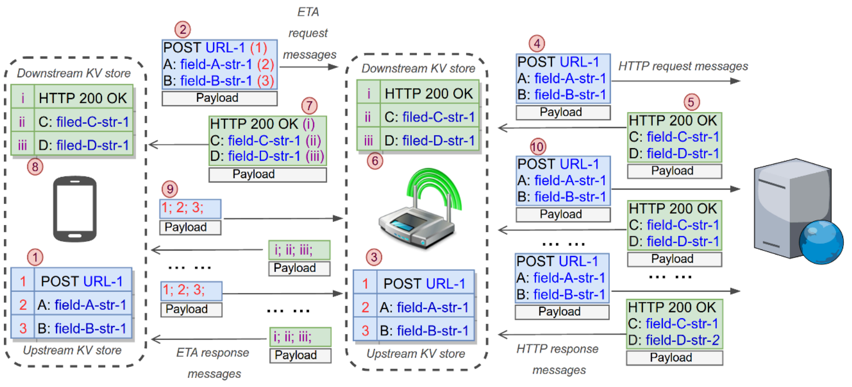


Fig. 4. An illustration of *ETA*'s approach of enabling stateful communication between a mobile device and an *ETA* EPS instance for the example scenario in Fig. 3.

message received (step ⑧). When the app sends out the second HTTP `POST` message, *ETA* intercepts it, replaces all the header fields, whose contents already exist in the device upstream key-value store, with the store keys to form an *ETA* request message, and send the resulted message to the EPS instance (step ⑨). When the EPS instance receives the *ETA* request message, it looks up its upstream key value-store, replaces the keys with the corresponding values to restore the original HTTP `POST` message, and sends to the cloud server (step ⑩). The following downstream messages will follow the similar process. With this design, we can significantly reduce the bandwidth consumption on the mobile device caused by transmitting the lengthy repetitive header contents.

Adjusting message header contents based on the capabilities of clients: To eliminate ineffective header contents in HTTP(S) messages, before sending out the intercepted/received HTTPS request/response messages, *ETA*-enabled devices or *ETA* EPS instances would examine the messages, and remove those ineffective fields based on the information in the response/request messages they previously received. In our implementation, this is achieved by building and checking a lookup table, which records the causal relationship between different HTTP(S) request/response header fields based on the HTTP(S) specification. For example, to deal with the problem of long and ineffective `Set-Cookie` header field in Google Docs' web traffic (§II-A), an *ETA* EPS instance would remove the `Set-Cookie` header fields from the HTTPS response messages before sending them to the mobile device, because it has not observed any `Cookie` field in the HTTPS request message they received from the mobile device (meaning the client does not support cookie).

IV. *ETA* EPS PROTOTYPE IMPLEMENTATION

Implementation overview: We have implemented the proposed *ETA* mechanism and deployed it in a lab environment in the form of EPS. The prototype system consists of two parts: the edge node EPS implementation running *ETA* functionality and the *ETA* device component. An overview of the prototype system is shown in Figure 5.

• The edge node EPS implementation has three major components: (1) the *EPS instances* running the *ETA* edge functionality for different users; (2) the *EPS manager*, whose job is to manage the life cycle of EPS instances (e.g., creation and destruction); and (3) the *EPS runtime framework* to support the efficient execution of a large number of EPS instances, as well as secure resource isolation among them. The EPS edge node implementation has been performed on two hardware platforms: an x86-based laptop equipped with a quad-core 3.4 GHz CPU/32 GB memory, and a ARM-based ODROID XU3 board equipped with big.LITTLE ARM Cortex A15 quad-core and Cortex-A7 quad-core CPUs/2 GB memory [10]. Both platforms were normally functioning as a WiFi access point in our implementation and evaluation experiments.

• The *ETA* device component's job is to intercept the HTTP(S) traffic generated by the apps, and adapt those traffic of interest into *ETA* message stream. It was implemented on a Samsung Galaxy S4 smartphone.

In the following, we describe the notable implementation challenges and our approaches of addressing them.

Exploring lightweight virtualization techniques for lightweight and secure EPS edge runtime: Recall that the goal of the EPS edge runtime is twofold: one is to efficiently support a large number of EPS instances for different users and different functionalities (e.g., *ETA* as we show in this paper); the other is to securely isolate the resource of different EPS instances, which may posses and handle private information for different users. Therefore, achieving lightweight and secure EPS edge runtime is critical for the success of the EPS architecture.

We have explored using lightweight virtualization techniques, which can well satisfy the lightweight and the resource isolation requirements of EPS edge runtime. There are two lightweight virtualization approaches suitable for EPS: *container* [11]–[13] and *VM-based unikernel* [14]–[17].

• The container approach (the lower right part in Figure 5) utilizes OS level virtualization method to provide a virtualized OS environment to host individual applications or OSes.

• The VM-based unikernel approach (the lower left part of Figure 5) adopts the library OS concept to eliminate the management overhead caused by traditional OSes on specialized applications, such as EPSes in our case. It integrates application logic (e.g., EPS logic), the corresponding libraries

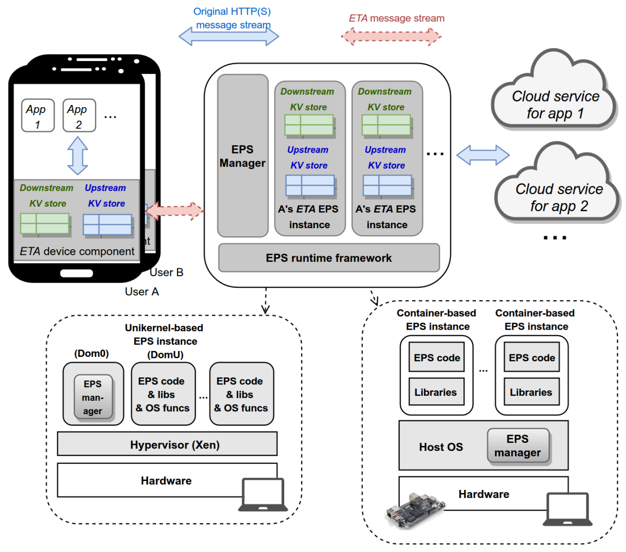


Fig. 5. Prototype implementation of the *ETA* EPS system. The edge node implementation was performed on two hardware platforms (x86 laptop and embedded ARM development board) using two lightweight virtualization techniques (VM-based unikernel and OS container).

(e.g., SSL/TLS), and related system functionalities (e.g., memory allocation, networking. etc.) into one tiny (e.g., a few megabytes) bootable VM image, which can directly run on a bare-metal hypervisor.

Our implementation with the container approach was performed on both the laptop platform and the ARM development board platform. We use Docker [11] as the container environment in the implementation. The EPS manager was implemented using the relevant Docker utilities. The normal services of the AP are provided by the host OS. The implementation with VM-based unikernel approach was performed on the laptop platform using the Rumprun unikernel framework [18]. The *ETA* EPS instances directly run on virtual machines (DomU) created through the Xen hypervisor. The normal services of the AP are performed in the full-fledged Linux-based host OS running in Domain 0. The EPS manager is also running in Domain 0. We are in the process of building our own unikernel framework that can run on ARM environment. The existing unikernel implementations for ARM are rudimentary and cannot support complex libraries such as SSL/TLS. We expect to be able to report our finding and experience in the near future.

Intercepting HTTPS traffic on mobile devices without modifying mobile OS or apps: To allow our system to be practical and easily deployable to existing mobile devices, an important goal when implementing the device component is to achieve HTTP(S) traffic interception and adaptation without requiring modifying either the mobile OS or the apps. There are multiple ways to achieve the goal. Our approach is implementing a user level SSL proxy using the OpenSSL library. The device component relies on the widely available system network proxy feature on all major OSes to route all the WiFi or cellular traffic to our SSL proxy, which in turn deciphers the traffic and performs the traffic adaption according to the *ETA* design. The second approach is utilizing Linux's TUN/TAP driver to intercept network traffic at layer 3 or layer 2. The above two approaches enable system-wide traffic routing, it requires installing our own SSL certificate on both the mobile device and the edge node. To avoid this limitation, we can also use the runtime hooking approach [19], [20] to intercept apps' calls to HTTP library function calls, perform the traffic adaptation, and resume the normal call flow of the apps. According to our experience, the latter two approaches have the similar performance as the first one (i.e., the system network proxy approach).

TABLE III
HTTPS MESSAGE ROUND TRIP TIME BREAKDOWN (UNIT: MILLISECOND)

Edge node platform	T_{dev}	T_{EPS}	T_{other}	**Overall**
Original (w/o EPS)	-	-	-	**197.28**
Unikernel on x86	0.64	0.15	175.76	**176.55**
Docker on x86	0.93	0.22	234.31	**235.47**
Docker on ARM	0.75	2.74	308.05	**311.55**

Managing the key-value stores on mobile devices and edge nodes: The key-value stores on both device component and EPS instances are memory based. Currently we use a simple strategy to manage the key-value stores: a fixed amount of entries are allocated for each key-value store, and these entries are filled linearly. The filling process wraps to the first entry and continues once the store is full.

V. SYSTEM EVALUATION

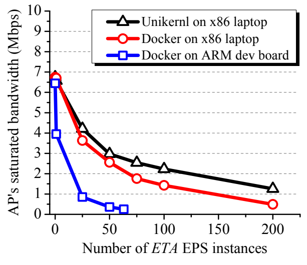
***ETA*'s effectiveness in mobile device bandwidth reduction**: We first evaluated *ETA*'s performance on reducing bandwidth consumption for web services with *hifasp* traffic. For example, we asked the same user to type the same paragraph of text (as described in §II) using the Google Docs app on the *ETA*-enable Samsung Galaxy S4 smartphone, and measured the traffic generated. With the on-screen keyboard, only 310 KB of traffic generated. Compared with the 1.94 MB traffic generated for typing the 1,016 character, our system saved about 84% of traffic. With the Bluetooth keyboard, our system reduced the bandwidth consumption by 83%. It is worth noting that the actual bandwidth saving ratio was larger than the redundant/repetitive content ratio measured in the motivation study. This is because the new line characters in HTTP message headers were not counted when calculating the redundant/repetitive content ratio. But these characters can be removed by our *ETA* system.

HTTPS message round trip time: We evaluated the average HTTPS message round trip time, which was calculated as the time between sending out an HTTPS request message and receiving the response message on the device. We instrumented our system and measured the following components of the the round trip time.

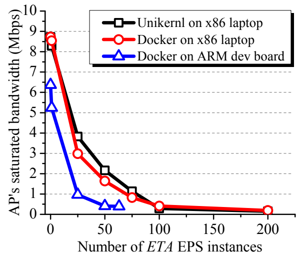
• T_{dev}: time spent in the *ETA* device component, which is mainly the traffic adaptation time spent on the smartphone.

• T_{EPS}: time spent in the *ETA* EPS instance, which is mainly the traffic adaptation time spent on the edge node.

• T_{other}: all other time, which consists of the time spend on WiFi transmission, Internet transmission, web server process-



(a) Uplink bandwidth



(b) Downlink bandwidth

Fig. 6. The saturated bandwidth of the AP (i.e., the edge node) as the number of *ETA* EPS instances was scaled up.

ing, and network processing on both the smartphone and the edge node.

In the experiment, we used the Google Docs app to send 50 HTTPS messages, and received 50 response messages from the cloud server. The average size of the HTTPS messages are as reported in Table II. Only one EPS instance was running on the edge onde. The average round trip times are shown in Table III. We can see that our implementation with unikernel on the x86 laptop essentially incurred no overhead regarding the round trip time. The overall round trip time for the unikernel implementation was even smaller than that without using EPS, because of the much smaller traffic for the WiFi transmission. This result suggests that unikernel is highly lightweight to be used as the EPS edge runtime. The Docker based implementation on the ARM board incurred about 100 ms extra time, which was mainly due to the slow network processing on the board.

EPS instance scalability on edge node: We evaluated the capability of our prototyped edge EPS runtime to support concurrent EPS instances. To enable a large number of concurrent active EPS instances on edge node, we develop a program running on PC to simulate individual Google Docs users. The PC was connected to a server, which was located in the local network and replied to the simulated Google Docs users with Google Docs-like HTTPS response messages. In the experiment, we simulated different numbers (i.e., 1, 25, 50, 75, 100 and 200) of users, so that the corresponding number of active *ETA* EPS instances can be initiated.

We used two metrics to evaluate the feasibility of using *ETA* EPS in real-world scenario, which are, when a large number of active EPS instances are running on the edge node at the same time, how (1) the normal edge node functionality and (2) the HTTP(S) message round trip time can be affected. For evaluating the first metric, we measured the saturated bandwidth using `iperf` as we scaled up the number of active EPS instances. Figure 6 shows the results, where we can have two findings: (1) The ARM development board was not able to run more than 63 EPS instances. We are now working on optimizing for the ARM board to improve its performance. (2) Unikernel based implementation worked better than the Docker based one. For the second metric, we measure the average HTTPS message round trip time as we increased the number of active EPS instances on the edge node. Figure 7 shows the result, from which we can have the similar

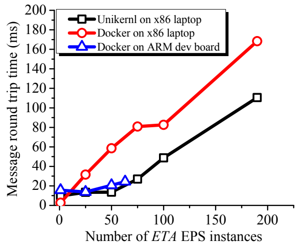


Fig. 7. HTTPS round trip time as the number of *ETA* EPS instances was scaled up.

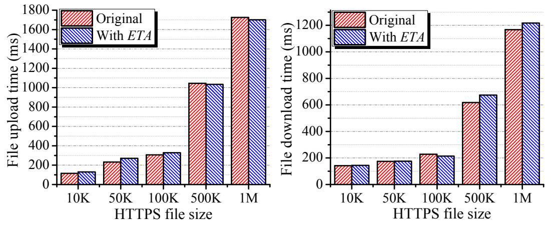
conclusion as the previous metric.

EPS system overheads: The overheads mainly occur on mobile devices as the HTTP(S) traffic needs to be routed to the device component for processing. We performed two experiments to evaluate the impacts. In experiments, we sent a stream of files using HTTPS on the *ETA*-enable smartphone. However, the resulted HTTPS traffic was not adapted by *ETA*. By doing so, we simulated HTTPS traffic that is not *hifasp*, and thus is not handled by *ETA*. In one experiment, we measured the transmission time (results in Figure 8); and in the other experiment, we measured the device system power (results in Figure 9). From the results we can see that our prototype system incurred a small amount of time and power overheads.

VI. Related Work

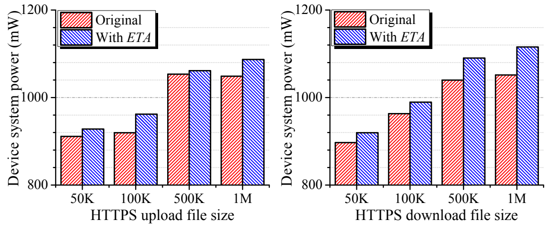
Utilizing edge infrastructure for mobile usage improvement: Edge computing has received an fast increasing amount of attention recently due to its various advantages [21]–[23]. There have been several works focusing on exploiting edge infrastructure for improving user experience on mobile devices. For example, to overcome the constraints of limited computation resources and long communication latency on mobile device, Satyanarayanan et al. proposed to offload computation from devices to VM-based cloudlets [24]. To improve the user experience for existing cloud services, AirBox [25] moves delay-sensitive cloud tasks from data centers to network edge. ParaDrop [26] facilitates third-party developers to deploy their services on the edge for the users to enjoy low communication latency between to/from the services. We have previously explored the EPS concept [27], [28]. In [27], we first proposed the idea of EPS. In [28], we leverage EPS to solve the problem of Office suite document synchronization in cloud storage services. In this paper, we address the problem of excessive bandwidth consumption in mobile web services with EPS, and prototyped the system using real ARM-based hardware.

Proxy middleware for web services and mobile devices: The proposed EPS architecture is essentially a middleware architecture. Using middleware design is common in many solutions of optimizing performances for web services and mobile devices. Google's Flywheel [29] and Baidu's TrafficGuard [30] are web proxies helping reducing web traffic on mobile devices. Scepter [31] is a proxy system to reduce overall network traffic for mobile devices. It shares the similar idea as *ETA*, which is to turn the network communication occurring on mobile devices to stateful to eliminate the redundant



(a) Upload time overhead (b) Downloud time overhead

Fig. 8. Time overhead on the smartphone for transmitting normal HTTPS traffic not adapted by *ETA*.



(a) Upload power overhead (b) Download power overhead

Fig. 9. Power overhead on the smartphone for transmitting normal HTTPS traffic not adapted by *ETA*.

or repetitive content in each network packet. Our work is different in that we focus on experimentally demonstrating and analyzing the problem of excessive bandwidth consumption problem for web services on mobile devices, and we also concentrate on the broader context of extending existing mobile-cloud computing infrastructure with the introduction of EPS. There are also works on client-side middleware to improving performances for mobile devices. For example, CacheKeeper [32] is a client-side system-wide web caching system for mobile applications. QuickSync [33] adopts the middleware approach to optimize cloud storage sync performances on mobile devices based on wireless network conditions.

VII. Conclusion

In this paper, we demonstrated the problem of excessive client-side network bandwidth consumption problem for mobile services with *hifasp* traffic, and designed *ETA*, an effective solution for the problem. We performed in-depth analysis to illustrate the challenges of deploying the solutions like *ETA*. We proposed EPS, a novel architecture to enable device-oriented solution deployment without affecting cloud services. We implemented a prototype EPS system exploring lightweight virtualization techniques, and demonstrate the effectiveness and efficiency of the system by deploying *ETA* with it.

Acknowledgements: We thank the anonymous reviewers for their tremendously valuable feedbacks. This work was supported in part by NSF Award #1566375.

References

[1] Google Inc., "Google Docs," https://www.google.com/docs/about/.
[2] Dropbox Inc., "Dropbox Paper," https://www.dropbox.com/paper.
[3] ShareLaTeX, "ShareLaTeX, Online LaTeX Editor," https://www.sharelatex.com/.
[4] Wikipedia, "Representational state transfer," https://en.wikipedia.org/wiki/Representational_state_transfer.
[5] J. Purushothaman, *RESTful Java Web Services: Design scalable and robust RESTful web services with JAX-RS and Jersey extension APIs*. Packt Publishing Ltd, 2015.
[6] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The itc distributed file system: Principles and design," in *ACM SOSP*, 1985.
[7] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West, "Scale and performance in a distributed file system," in *ACM SOSP*, 1987.
[8] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on computers*, vol. 39, no. 4, pp. 447–459, 1990.
[9] Network Working Group, "Network File System (NFS) version 4 Protocol," https://www.ietf.org/rfc/rfc3530.txt.
[10] Wikipedia, "ODROID," https://en.wikipedia.org/wiki/ODROID.
[11] Docker, Inc., "Docker," https://www.docker.com/.
[12] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM EuroSys*, 2007.
[13] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *ACM EuroSys*, 2016.
[14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX NSDI*, 2014.
[15] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels." in *USENIX NSDI*, 2015.
[16] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM ASPLOS*, 2013.
[17] A. Kivity, D. L. G. Costa, and P. Enberg, "Osv - optimizing the operating system for virtual machines," in *USENIX ATC*, 2014.
[18] Rump Kernel, "Runprun unikernel," https://github.com/rumpkernel/rumprun.
[19] Collin Mulliner, "Android DDI: Dynamic Dalvik Instrumentation," http://www.mulliner.org/android/feed/mulliner_ddi_30c3.pdf.
[20] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime," in *The Workshop on Innovations in Mobile Privacy and Security (IMPS) at ESSoS*, 2016.
[21] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, 2016.
[22] M. Satyanarayanan, "The emergence of edge computing," *IEEE Computer*, 2017.
[23] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *ACM Workshop on Mobile Big Data*, 2015.
[24] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.
[25] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2016.
[26] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the networkâĂŹs extreme edge," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2016.
[27] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "Poster: EPS - Edge-hosted Personal Services for Mobile Users," in *ACM MobiSys*, 2017.
[28] P. Hao, Y. Bai, X. Zhang, and Y. Zhang, "EdgeCourier: An Edge-hosted Personal Service for Low-bandwidth Document Synchronization in Mobile Cloud Storage Services," in *ACM/IEEE Symposium on Edge Computing (SEC)*, 2017.
[29] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin, "Flywheel: Google's data compression proxy for the mobile web," in *USENIX NSDI*, 2015.
[30] Z. Li, W. Wang, T. Xu, X. Zhong, X.-Y. Li, Y. Liu, C. Wilson, and B. Y. Zhao, "Exploring cross-application cellular traffic optimization with baidu trafficguard." in *USENIX NSDI*, 2016.
[31] J. Hare, D. Agrawal, A. Mishra, S. Banerjee, and A. Akella, "A network-assisted system for energy efficiency in mobile devices," in *International Conference on Communication Systems and Networks*, 2011.
[32] Y. Zhang, C. Tan, and L. Qun, "CacheKeeper: A System-wide Web Caching Service for Smartphones," in *ACM UbiComp*, 2013.
[33] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," in *ACM MobiCom*, 2015.