SCoP: Smartphone Energy Saving by Merging Push Services in Fog Computing

Shang GAO*, Zhe PENG*, Bin XIAO*, Qingjun XIAO†, Yubo SONG†
*Department of Computing, The Hong Kong Polytechnic University
†School of Information Science and Engineering, Southeast University
{cssgao, cszpeng, csbxiao}@comp.polyu.edu.hk, {csqjxiao, songyubo}@seu.edu.cn

***Abstract*—Energy saving solutions on smartphone devices can greatly extend a smartphone's lasting time. However, today's push services require keep-alive connections to notify users of incoming messages, which cause costly energy consuming and drain a smartphone's battery quickly in cellular communications. Most keep-alive connections force smartphones to frequently send heartbeat packets that create additional energy-consuming radio-tails. No previous work has addressed the high-energy consumption of keep-alive connections in smartphones push services. In this paper, we propose Single Connection Proxy (SCoP) system based on fog computing to merge multiple keep-alive connections into one, and push messages in an energy-saving way. The new design of SCoP can satisfy a predefined message delay constraint and minimize the smartphone energy consumption for both real-time and delay-tolerant apps. SCoP is transparent to both smartphones and push servers, which does not need any changes on today's push service framework. Theoretical analysis shows that, given the Poisson distribution of incoming messages, SCoP can reduce the energy consumption by up to 50%. We implement SCoP system, including both the local proxy on the smartphone and remote proxy on the "Fog". Experimental results show that the proposed system consumes 30% less energy than the current push service for real-time apps, and 60% less energy for delay-tolerant apps.**

I. Introduction

Motivation. The computing ability (CPU, GPU, etc.) of smartphones has increased dramatically in the past few years, which requires more energy for computing. However, the increment of battery capacity is not significant. As a result, smartphones' standby time could be less and less. To extend a smartphone's daily usage time, a hot topic is to reduce the energy consumption of a smartphone when gathering data from the servers over the Internet, especially through cellular network connections [1]. The cellular radio in the smartphone remains in a high energy state for up to 20s after each communication spurt [2]. To get a small piece of data from a server, the cellular radio needs to wake up from a low power idle state, switch to a high energy active state (DCH), and then stay in the DCH state for a certain inactive duration (e.g. 10s to 20s) before going back to the idle state. This inactive duration, also known as the radio-tail [3], is determined by the network operator in both the 3G and 4G LTE standards [4]. The high energy state of 20s is very energy-consuming and we should design efficient mechanism to minimize the energy cost whenever getting messages from Internet servers.

Many apps in smartphones frequently get messages from push servers. Push servers provide push service to notify their users of any new messages (or notifications, exchangeable in this paper) [5]. Though Google provides "Google Cloud Messaging (GCM)" as a solution for push services on one push server that aggregates multiple notifications, popular apps can still have their own push servers. This is a common scenario especially in some regions where Google services are blocked due to the local Internet censorship policy [6]. For instance, QQ uses XG push and Gaode map uses JPush. To get notifications from these app servers, a smartphone needs to establish multiple keep-alive connections with multiple push servers, as depicted in Fig. 1.

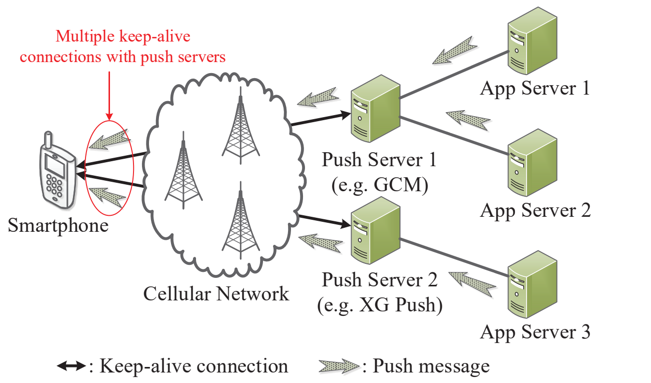


Fig. 1. Multiple push services on a smartphone. A smartphone has to maintain multiple keep-alive connections with different push servers.

In the current framework, a smartphone getting push messages involves two steps. First, the smartphone needs to maintain keep-alive connections with push servers by regularly sending out heartbeat packets. Second, a smartphone receives messages from push servers when a server finds that it is alive and a new message is available. Such keep-alive connections and message receiving could be both frequent and energy-intensive, which force the high-low energy state switches and drain a smartphone's battery quickly. A report from JPush shows that a single push service consumes 30mAh per day averagely [7]. Considering the energy cost of waking up the screen when receiving notifications, nearly 20% energy is consumed by push services for normal smartphone users.

No efficient solutions have been proposed yet to reduce the energy cost of smartphones in the scenario of multiple push services. How to merge multiple keep-alive connections into one remains to be open. Previous representative solutions, *TOP* algorithm [4] and fast dormancy [2], only focus on reducing radio-tails without considering the cost of multiple keep-alive connections. *TOP* algorithm enables cooperation between a smartphone and network to reduce radio-tails. For the scenario of multiple push services, however, it is hard to apply *TOP* algorithm on all push servers, because push servers can be set up independently by third-parties. Any push server violates *TOP* algorithm will increase a smartphone's energy consumption. Fast dormancy cuts down radio-tails by forcing the radio to transit from the DCH to the idle state quickly. However, the repeated transitions between idle and DCH states under bad network conditions may result in signal overloading (unpredictable increases in signaling load as the device repeatedly switches between idle and DCH states [2]) and increased energy cost.

Fog computing introduces a good solution for smartphone energy saving. It carries out a substantial amount of communication, control and management with near-user edge devices to benefit smartphones [8]. Fog computing has brought lots of benefits to today's networks, such as sensor, P2P and Wi-Fi networks [9]. In the push service scenario, fog computing can save the energy of smartphones by offloading the energy-intensive computation to the cloudlet, and optimizing radio-tails on the "fog" (cloudlet) side. This mechanism is practical in the Intranet with near-user edge devices on the cloudlet as "fog".

Contribution. Motivated by fog computing, we first introduce SCoP system, a framework to save energy for s-martphones in cellular communications. The SCoP system can successfully merge multiple push servers in one keep-alive connection by a two-proxy design: the local proxy and remote proxy. The local proxy is an application installed on a smartphone. The remote proxy is a near-user edge device built on the "fog" in company or home networks[1]. For instance, a user can set up the remote proxy in his home cloudlet that is always connected to the Internet. A company can also set up the remote proxy for all company and authorized users without any changes on today's push service framework.

SCoP saves a smartphone's energy in two ways: merging connections and message store-and-forward function. Originally, a smartphone needs to maintain multiple connections to multiple push servers. In contrast, the local proxy in SCoP merges push connections into one keep-alive connection with the remote proxy to save the energy for both real-time and delay-tolerant apps (e.g. merging the two keep-alive connections with GCM and XG Push into one in Fig. 1). Furthermore, the remote proxy identifies delay-tolerant apps, stores the push messages within a user-set delay time, and forwards them altogether to the local proxy to reduce radio-tails. The two-proxy design is transparent to both apps and push servers (supporting all apps with notification service, e.g. Facebook), and can efficiently minimize the smartphone energy cost in both keep-alive connections (i.e. via the local proxy) and radio-tails (i.e. via the remote proxy).

[1]Though the remote proxy could also be deployed in the cloud, we do not encourage this scenario since it may introduce some security problems.

We then build a model to evaluate the energy cost of a smartphone, including the local and Internet energy consumptions. We compare energy cost between the current push framework and our SCoP system based on the Poisson distribution of incoming messages. Under an empirical delay time constraint, SCoP can save 50% more energy than the current framework even when we consider the local proxy energy cost.

Finally, we implement a prototype of SCoP system and evaluate its performance. We design an application of the local proxy on a smartphone, and build the remote proxy on the Internet. We also evaluate the performance of SCoP system in real-world experiments. We install the local proxy app on an Android smartphone and use a power meter to measure the energy consumption. The experimental results show that SCoP consumes 30% less energy for real-time apps, and 60% less energy for delay-tolerant apps than the current framework. The store-and-forward function in the remote proxy can reduce nearly 4J energy per message under a given short delay time of 156 seconds.

The rest of the paper is organized as follows. Section II introduces the energy cost model and the key problem. In section III, we present the detailed designs of SCoP system. Section IV analyzes its performance theoretically and section V shows the implementation of our SCoP prototype. The experimental evaluation of SCoP's performance is shown in VI. We summarize the related work in section VII. Finally, we conclude the paper in section VIII.

II. Problem Statement

We give the energy cost model of performing push operations on smartphones, and state the problem of reducing a smartphone's energy consumption.

A. System Model

We consider a set of apps $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ that use the push technique, and a set of push servers $\mathcal{S} = \{s_1, s_2, ..., s_m\}$ that push messages to apps. For apps that use the same push server, they may not need to establish multiple keep-alive connections since developers can optimize push clients. We denote the cost of setting up and maintaining each network connection by $E_{conn}(s_i)$, and the cost of network communication by E_{com}. Suppose in a time period t, k messages are received, $\mathcal{M} = \{M_1, M_2, ..., M_k\}$. Let $E_{com}(M_i)$ be the energy cost of receiving message M_i, t_i be the arriving time of M_i, and $t_{t(i)}$ be the time when the smartphone starts to receive M_i.

In the push service, a smartphone first establishes keep-alive connections and registers on push servers. Second, the smartphone fetches tokens associated with the device from push servers, and forwards these tokens to app servers. Third, apps send heartbeat packets regularly to push servers to maintain these keep-alive connections. Finally, push servers push messages to the smartphone when they receive new messages from app servers. These energy-intensive connections and push messages can drain a smartphone's battery quickly. For normal

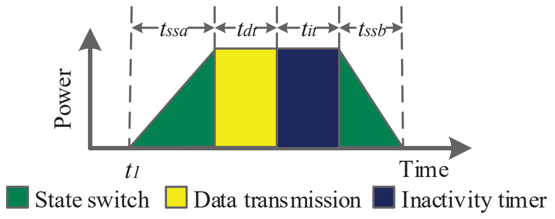


Fig. 2. Energy consumption of sending/receiving one message (M_1).

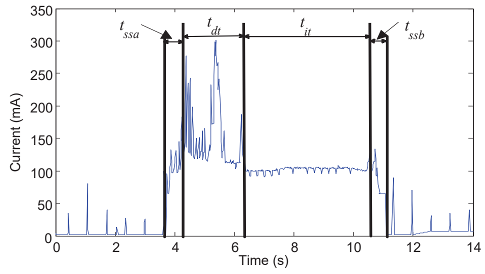


Fig. 3. A real test to show the energy cost of transmitting one message in cellular networks. The data are collected from our experiments.

smartphone users, nearly 20% energy is consumed by push services.

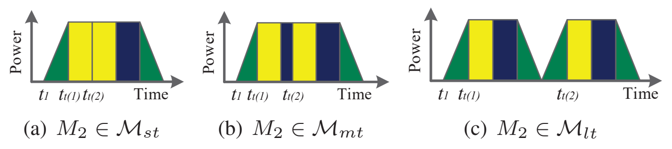
B. Energy Cost

There are three states during data transferring when sending or receiving a message: the state switch, data transmission, and inactivity timer [10] as depicted in Fig. 2. We denote the time of the state switch from idle state to active by t_{ssa}, the data transmission time by t_{dt}, the time of the inactivity timer by t_{it}, and the time of the state switch from active to idle state by t_{ssb}. Fig. 3 shows the real-time energy cost of a smartphone when transmitting one message in cellular networks.

The message state transition of M_i can be categorized into three types based on the value of $(t_{t(i)} - t_{t(i-1)})$. If $(t_{t(i)} - t_{t(i-1)}) = t_{dt}$, M_i arrives before M_{i-1} switches to its inactivity timer, a smartphone transmits these two messages without state changing as depicted in Fig. 4-a. We use $\mathcal{M}_{st} = \{M_i | (t_{t(i)} - t_{t(i-1)}) = t_{dt}\}$ to represent this subset of $\mathcal{M}$. If $t_{dt} < (t_{t(i)} - t_{t(i-1)}) \leqslant (t_{dt} + t_{it})$, M_i arrives during the inactivity timer of M_{i-1}, these two messages are still transmitted in the same active state period as depicted in Fig. 4-b. However, the energy cost in this scenario is greater than the previous one due to the inactivity timer of M_{i-1}. We use $\mathcal{M}_{mt} = \{M_i | t_{dt} < (t_{t(i)} - t_{t(i-1)}) \leqslant (t_{dt}+t_{it})\}$ to represent those messages. Otherwise, M_i is transmitted in another active state as depicted in Fig. 4-c. We use $\mathcal{M}_{lt} = \{M_i | (t_{t(i)} - t_{t(i-1)}) \in [t_{ssa} + t_{dt} + t_{it} + t_{ssb}, +\infty)\} \cup M_1$ to represent them. Therefore, the time interval between a smartphone receiving two messages $(t_{t(i)} - t_{t(i-1)})$ satisfies:

$$(t_{t(i)} - t_{t(i-1)}) \in \\ [t_{dt}, (t_{dt} + t_{it})] \cup [(t_{ssa} + t_{dt} + t_{it} + t_{ssb}), +\infty). \quad (1)$$

The energy cost $E_{com}(M_i)$ will be different when M_i belongs to different subset of $\mathcal{M}$. When $M_i \in \mathcal{M}_{st}$, we separate $E_{com}(M_i)$ into two parts as depicted in Fig. 5. The first part is the energy cost of sending/receiving one single message. The second part is the cost of data transmission of



Power
t1
tt(1)
tt(2)
Time
(a) $M_2 \in \mathcal{M}_{st}$
(b) $M_2 \in \mathcal{M}_{mt}$
(c) $M_2 \in \mathcal{M}_{lt}$

Fig. 4. Energy costs for messages in $\mathcal{M}_{st}$, $\mathcal{M}_{mt}$, and $\mathcal{M}_{lt}$ respectively.

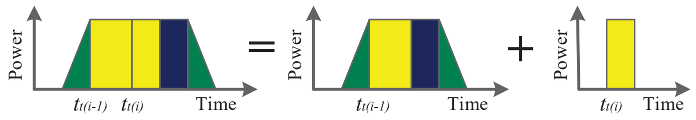


Fig. 5. Splitting messages in $\mathcal{M}_{st}$ to calculate energy cost.

M_i. For $M_i \in \mathcal{M}_{mt}$, we can also use this idea to separate them. The second part costs additional energy in the inactivity timer. Otherwise, M_{i-1} and M_i are sent in two individual active states, and we calculate them separately.

We denote the energy cost in data transmission by $E_{dt}(M_i)$, the energy cost in the state switch and inactivity timer by $E_{ssit}(M_i)$, and the energy cost in the inactivity timer by $E_{it}(M_i)$. $E_{com}(M_i)$ can be calculated as follows:

$$E_{com}(M_i) = \begin{cases} E_{dt}(M_i), & M_i \in \mathcal{M}_{st} \\ E_{dt}(M_i) + E_{it}(M_i), & M_i \in \mathcal{M}_{mt} \\ E_{dt}(M_i) + E_{ssit}(M_i), & M_i \in \mathcal{M}_{lt} \end{cases}. \quad (2)$$

The energy consumption of sending a message differs from that of receiving a message in cellular networks [10]. Since a smartphone receives data most of time in the push service scenario, we only calculate the energy cost of receiving messages in E_{com}. In a time period t, E_{com} can be calculated as follows when receiving k messages:

$$\begin{aligned} E_{com} &= \sum_{i=1}^{k} E_{com}(M_i) \\ &= \sum_{i=1}^{k} E_{dt}(M_i) + \sum_{M_i \in \mathcal{M}_{mt}} E_{it}(M_i) + \sum_{M_i \in \mathcal{M}_{lt}} E_{ssit}(M_i). \quad (3) \end{aligned}$$

For one app a_i, we denote its local energy cost (e.g. calculating, local communication etc.) by $E_{local}(a_i)$, and the energy cost to maintain the connection with a base station in cellular networks by E_{hb}. The total energy cost E of a smartphone in the process of receiving k messages from push servers is the sum of the four parts: the local energy cost of n apps, the cost of establishing and maintaining m keep-alive connections with push servers, the cost of receiving messages and the cost of maintaining the connection with a base station.

$$E = \sum_{i=1}^{n} E_{local}(a_i) + \sum_{j=1}^{m} E_{conn}(s_j) + E_{com} + E_{hb} \quad (4)$$

In the delay-tolerant model, we also take the delay constraint into consideration. We denote the delay time of the i-th message M_i by $t_{delay}(M_i)$, and the tolerable delay time of a user by $t_{threshold}$. The delay time of any message should satisfy:

$$\max\{t_{delay}(M_i) | M_i \in \mathcal{M}\} \leqslant t_{threshold}. \quad (5)$$

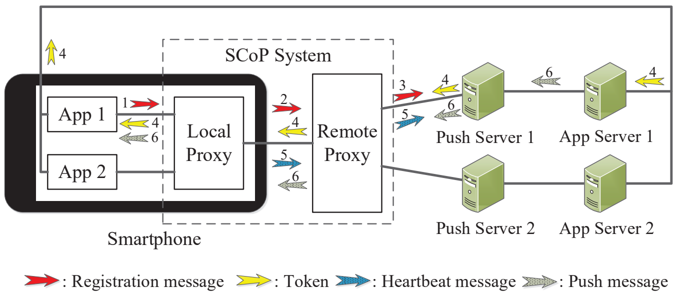


Fig. 6. SCoP architecture. The number on each arrow indicates the step number in the workflow.

C. Problem Formulation

The problem studied in this paper is how to reduce the energy cost of push services on smartphones. The energy cost of a smartphone in cellular networks can be calculated by Equation (4). It consists of four parts: the local energy cost of n apps $\sum_{i=1}^{n} E_{local}(a_i)$, the cost of establishing and maintaining m network connections $\sum_{j=1}^{m} E_{conn}(s_j)$, the communication cost E_{com}, and the cost of heartbeats with a base station E_{hb}. Based on Equation (4), we can find out that E increases with the growth of n and m. Thus, to reduce the energy cost of a smartphone, we should focus on part two and three to reduce the number of connected push servers m to minimize $\sum_{j=1}^{m} E_{conn}(s_j)$ and to reduce communication cost E_{com}.

Our goal is to design a framework to reduce the cost of keep-alive connections with push servers ($\sum E_{conn}(s_j)$) by merging them into one; and push messages in an energy-saving way to minimize radio-tails (E_{com}) under a delay constraint.

III. System Design

We design a system named SCoP, which can reduce the energy cost of a smartphone by merging multiple connections and pushing message in an energy-saving way. We describe the design of the SCoP system, including its architecture, detailed designs and challenges as follows.

A. SCoP Architecture

The framework of SCoP system consists of a local proxy and a remote proxy as depicted in Fig. 6. The local proxy is an application that runs on a smartphone, while the remote proxy is a proxy server that works on the Internet. SCoP reduces energy cost in two ways. First, it saves the cost of setting up and maintaining network connections by merging them into one on the local proxy for both real-time and delay-tolerant apps. Second, it reduces radio-tails by the store-and-forward function on the remote proxy for delay-tolerant apps. The local proxy behaves same for both real-time and delay-tolerant apps, but the remote proxy treats them differently.

For real-time apps, the workflow of SCoP system is as follows. First, the local proxy establishes a connection with the remote proxy, and redirects all push connections to it. Second, the local proxy merges these connections into one, and forwards registration messages to the remote proxy. Third, the remote proxy receives messages from the merged connection, it splits them and forwards messages to push servers accordingly. Fourth, after receiving tokens from push servers, the remote proxy forwards them to the smartphone, and the smartphone sends token messages to the app servers directly. Fifth, the local proxy blocks all heartbeat messages from the smartphone, and sends new heartbeat messages in a low rate to notify the remote proxy that it is alive. Meanwhile, the remote proxy generates and sends heartbeat messages to each push server when the smartphone is alive. Finally, the remote proxy forwards push notification messages from push servers to the local proxy, and the local proxy forwards them to apps accordingly.

For delay-tolerant apps, the remote proxy stores the push messages and waits for other messages within a tolerable time before forwarding them. SCoP is transparent to both smartphones and push servers, which does not need any changes on today's push service framework.

B. Local Proxy

The local proxy has three major functions, i.e. network connections redirection, blocking/splitting messages, and forwarding messages to the remote proxy/apps.

1) Redirection: Since some apps may not provide interfaces to use proxy, making apps send messages to the local proxy first is challenging. One solution is to build the local proxy as a transparent proxy. However, once the smartphone sets its gateway to 127.0.0.1, no message can be sent out. The connection with the remote proxy is unable to be established, and all network traffic will be regarded as push traffic. Therefore, the local proxy should be able to distinguish push traffic and regular traffic.

We first use an IP address table of the push servers to distinguish push traffic and regular traffic. The local proxy uses DNS protocol to automatically update the IP addresses of push servers every time it starts. It also provides interfaces for users to add more push servers. Then, we fulfill the redirection function based on iptables. It redirects all push connections to the local proxy. Meanwhile, it also allows users to set up bypass push servers which they do not wish to use SCoP.

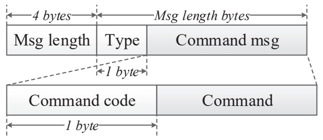
Suppose there are two records in the IP address table, push server A (the IP address is 1.1.1.1) and push server B (the IP address is 2.2.2.2), and the port of the local proxy is 8080. If we attempt to redirect all push connections except those to push server B, the local proxy first sends DNS requests to get the IP address of each record in the IP address table, i.e. 1.1.1.1 for push server A, and 2.2.2.2 for push server B. Second, the local proxy gets the IP addresses of the bypass push servers, and removes these bypass IP addresses from the IP set. In this case, the local proxy removes 2.2.2.2 from {1.1.1.1, 2.2.2.2}, and only leaves 1.1.1.1. Third, the local proxy redirects push connections by iptables with the following codes.

```
# iptables -t nat -A PREROUTING -s 1.1.1.1/32 -p tcp --
    dport 80 -j DNAT --to-destination 127.0.0.1:8080
```

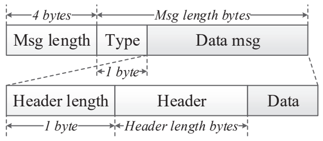
2) Blocking and splitting: In order to reduce the cost of heartbeat messages, the local proxy filters all heartbeat messages and sends its own heartbeats in a very low rate

TABLE I. Design of Command Code

Command	Code field	Data field
Shut down socket	0x00	null
Connection closed	0x01	connection's information
Suggest empirical $t_{threshold}$	0x02	empirical $t_{threshold}$
Set $t_{threshold}$	0x03	$t_{threshold}$
Update information (request)	0x04	NULL
Update information (response)	0x05	user's information
Extension	else	extension



(a) Data structure of control message.



(b) Data structure of data message.

Fig. 7. Message data structure.

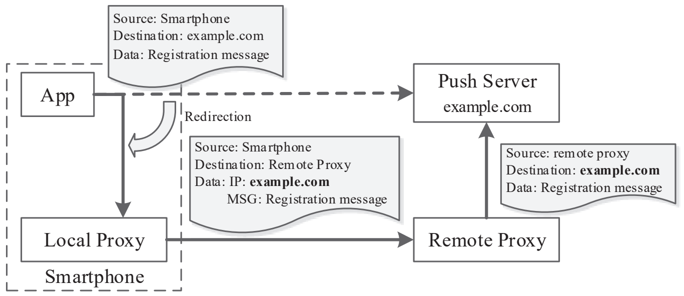


Fig. 8. SCoP forwards registration request.

to notify the remote proxy that it is alive. One challenging problem is that it is hard for the local proxy to distinguish heartbeat messages and registration messages, since these messages may be encrypted. In push service scenario, there won't be any traffic from apps to push servers except heartbeat messages after registration. Therefore, we block all messages from apps once the apps register successfully. In downstream messages, the remote proxy adds an 1-byte "Type" field to denote whether this message is a "token" message on the second most significant bit before sending it to the local proxy (1 for "token" messages and 0 for push messages). After the local proxy receives and forwards a "token" message, it regards this app registers successfully, and blocks all messages from this app. The local proxy sends a heartbeat message every 20 minutes to notify the remote proxy.

We introduce two types of messages, i.e. control message and data message, in the communication between the two proxies. They are distinguished by the most significant bit on the "Type" field (0 for control messages and 1 for data messages). Besides, we add a "Length" field for each message to identify its length.

Control message is to coordinate the communication between two proxies. In SCoP, it consists of four kinds of commands: shutting down socket, informing the local/remote proxy that one connection is closed, setting $t_{threshold}$, and updating a user's information. We also reserve some command codes for extension. The length of "Command code" field is 1 byte. The structure of control message is depicted in Fig. 7-a, and the detail of command code is depicted in Table I.

Data message carries the data which are transmitted between apps and push servers. The client's information is denoted on each message as a header. We use 1 byte to represent its length since the header is not long (30 bytes to 40 bytes). The structure of data message is depicted in Fig. 7-b.

The local proxy splits the merged messages based on the "Length" field, and forwards them to each app according to the header information.

3) Forwarding: As the local proxy uses only one connection with the remote proxy, it is important to maintain this connection even when apps or push servers shut down. The local proxy connects to the remote proxy by a client before its server starts. Once disconnected accidentally, the client tries to reconnect to the remote proxy. If it finds that the remote proxy is unreachable (e.g. the remote proxy shuts down), the local proxy will exit automatically and switch to current network connection.

Another challenge is the push server's information in registration messages may be lost because the destination IP address is changed to the remote proxy's after forwarding. The remote proxy is unable to get the IP address of push servers unless the local proxy adds them into the registration request. As depicted in Fig. 8, the local proxy adds the push server's IP address to the data field of data message.

C. Remote Proxy

Similar to the local proxy, the remote proxy also has splitting function and forwarding to the local proxy/servers function. Besides, the remote proxy applies the store-and-forward function to reduce radio-tails of for delay-tolerant apps. Since the procedures of splitting on the remote proxy are similar to those on the local proxy, we skip this part due to space constraints.

1) Forwarding: After receiving a modified registration request from the local proxy, the remote proxy changes the request back based on the IP address in the data field before forwarding to a push server, as depicted in Fig. 8. To notify push servers that the smartphone is alive, the remote proxy generates multiple heartbeat messages and spreads them to each push server.

2) Store-and-forward: After receiving push messages from a push server, the remote proxy first evokes the store-and-forward function, and then forwards messages to the local proxy. The remote proxy distinguishes real-time and delay-tolerant apps based on $t_{threshold}$. When a user sets $t_{threshold} = 0$, the remote proxy regards those apps as real-time apps. Otherwise, the remote proxy treats them as delay-tolerant apps.

The store-and-forward function is designed to reduce E_{com} by sharing radio-tails. Generally speaking, it stores messages for $t_{threshold}$ time and forwards them altogether to apps. Since a smartphone transmits messages continuously ($t_{t(1)} = t_{threshold} + t_{ssa}, t_{t(2)} = t_{threshold} + t_{ssa} + t_{dt}, ...$), there is only one radio-rail when receiving these messages, as depicted in Fig. 9. It is obvious that E_{com} is less with larger $t_{threshold}$ (more messages share one radio-tail), and there is no store-and-forward function when $t_{threshold} = 0$. We set $t_{threshold} = 0$ for real-time apps.

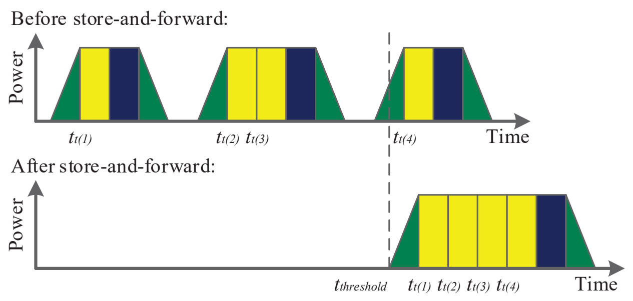


Fig. 9. Using the store-and-forward function to reduce radio-tails.

One challenge is that the secure protocols must be designed to resist replay attacks. One message is valid only under current connection or within a time period. Most of today's push services use SSL protocol to establish secure connections. SSL uses sequence number to resist replay attacks. This sequence number may cause serious problem to the store-and-forward function. If an app sets up a new connection with a push server (e.g. the app loses connection temporarily), forwarding messages received in the previous connection directly is invalid.

To solve this problem, we build the remote proxy as a man-in-the-middle proxy [11], which is transparent and does not need to modify the socket interfaces of both client and server sides. The basic idea is pretending to be the server to the client, and pretending the client to the server [11]. Therefore, we can avoid messages become invalid since the sequence number is always based on the current connection.

Man-in-the-middle proxy brings another challenge. Certificate authority system (CA) is designed to resist man-in-the-middle attack by allowing a trusted third-party to cryptographically sign a server's SSL certificates to verify that they are legit [11]. Apps can reject SSL handshake when the remote proxy fails to provide trusted certificates. The solution is to make the remote proxy a trusted certificate authority which includes a full CA implementation that generates interception certificates [11]. This solution is practical in the home or company networks. For instance, network administrators can place the remote proxy in the company network as "fog", and distribute its certificate in the Intranet. In home networks, we can place the remote proxy on the cloudlet and register it as a trusted CA on the smartphone manually.

IV. Numerical Analysis

To evaluate the performance of SCoP system, we compare the energy cost of a smartphone between using SCoP and the current framework theoretically. We first present the distribution of incoming push messages. Then we calculate and compare the expected energy cost under SCoP and the current framework. Finally, we give our empirical tolerable delay time.

A. Message Distribution

We build energy cost model based on the distribution of incoming messages from push servers. Previous studies [12], [13] have pointed out that the number of messages (X) received in a unit interval follows a Poisson distribution, and the distribution of a time interval ($\Delta t = t_i - t_{i-1}$) between receiving two messages M_i and M_{i-1} is exponential [12], [13]. Its cumulative distribution function is:

$$F(\Delta t;\lambda) = Pr(\Delta t \leqslant t_{int}) = \begin{cases} 1 - e^{-\lambda t_{int}}, & t_{int} \geqslant 0 \\ 0, & t_{int} < 0 \end{cases}. \quad (6)$$

The positive real number λ equals to the expected value of X and also to the variance of X.

B. Current Framework

We calculate E_{com} under the current framework. Recall Equation (3), for each message M_i, $E_{ssit}(M_i)$ is fixed ($E_{ssit}(M)$) for each smartphone. $E_{dt}(M_i)$ can be also regarded as a fixed value ($E_{dt}(M)$) in the push service scenario since the sizes of messages are almost the same. $E_{it}(M_i)$ is not fixed, but we can use the average value $\overline{E_{it}(M)}$ to calculate the expectation of E_{com}. Based on Equation (3), E_{com} can be calculated as follows:

$$\begin{aligned} E_{com} &= \sum_{i=1}^{k} E_{dt}(M_i) + \sum_{M_i \in \mathcal{M}_{mt}} E_{it}(M_i) + \sum_{M_i \in \mathcal{M}_{lt}} E_{ssit}(M_i) \\ &= kE_{dt}(M) + |\mathcal{M}_{mt}|\overline{E_{it}(M)} + |\mathcal{M}_{lt}|E_{ssit}(M). \quad (7) \end{aligned}$$

M_i belongs to $\mathcal{M}_{st}$, $\mathcal{M}_{mt}$, or $\mathcal{M}_{lt}$ is related to all messages transmitted in the same state with M_{i-1}. To simplify calculation, we only consider M_i is related to M_{i-1}, since a time interval is normally longer than the time of transmitting a message under the push service scenario. M_i is related to messages other than M_{i-1} only occurs when too many messages are received in a short period.

We divide the arriving time interval Δt into two subsets to calculate the probability of $M_i \in \mathcal{M}_{mt}$: $t_{dt} < \Delta t \leqslant (t_{dt} + t_{it})$ when M_{i-1} is transmitted without state changing; and $(t_{ssa} + t_{dt}) < \Delta t \leqslant (t_{ssa} + t_{dt} + t_{it})$ for other scenarios. Since t_{ssa} is a very small value for most smartphones, we regard $t_{dt} < \Delta t \leqslant (t_{dt} + t_{it})$ when $M_i \in \mathcal{M}_{mt}$. Therefore, the probability of $M_i \in \mathcal{M}_{mt}$ can be calculated as follows:

$$\begin{aligned} Pr(M_i \in \mathcal{M}_{mt}) &= Pr(t_{dt} < \Delta t \leqslant (t_{dt} + t_{it})) \\ &= e^{-\lambda t_{dt}} - e^{-\lambda(t_{dt}+t_{it})}. \quad (8) \end{aligned}$$

It is similar for the probability of $M_i \in \mathcal{M}_{lt}$. The arriving time interval Δt can be divided into two subsets: $(t_{dt} + t_{it}) < \Delta t < \infty$ when M_{i-1} is transmitted without state changing; and $(t_{ssa} + t_{dt} + t_{it}) < \Delta t < \infty$ for other scenarios. Here we regard $(t_{dt} + t_{it}) < \Delta t < \infty$ when $M_i \in \mathcal{M}_{lt}$. The probability of $M_i \in \mathcal{M}_{lt}$ can be calculated as follows:

$$Pr(M_i \in \mathcal{M}_{lt}) = Pr(\Delta t > (t_{dt} + t_{it})) = e^{-\lambda(t_{dt}+t_{it})}. \quad (9)$$

Based on Equation (7), Equation (8), and Equation (9), in a time period t, the expectation of E_{com} can be calculated as follows:

$$\begin{aligned} &\exp(E_{com}) = \\ &\quad \lambda t E_{dt}(M) + \lambda t(e^{-\lambda t_{dt}} - e^{-\lambda(t_{dt}+t_{it})})\overline{E_{it}(M)} + \\ &\quad \lambda t e^{-\lambda(t_{dt}+t_{it})} E_{ssit}(M). \quad (10) \end{aligned}$$

The expected energy cost under the current framework (E_{CF}) can be calculated as follows:

$$\exp(E_{CF}) = \sum_{i=1}^{n} E_{local}(a_i) + \sum_{j=1}^{m} E_{conn}(s_j) + \exp(E_{com}) + E_{hb}. \quad (11)$$

C. SCoP System

We denote the local energy cost of the local proxy by $E_{local}(a_{lp})$, and the connection cost to the remote proxy by $E_{conn}(s_{rp})$. Based on Equation (4), the energy cost under SCoP (E_{SCoP}) can be calculated as follows:

$$E_{SCoP} = \sum_{i=1}^{n} E_{local}(a_i) + E_{local}(a_{lp}) + E_{conn}(s_{rp}) + E_{com} + E_{hb}. \quad (12)$$

In the SCoP system, radio-tails are reduced by the store-and-forward function. For most cases, messages are transmitted in one active state when they are received in a time interval $t_{threshold}$. But for other cases, too many messages are received in the current interval, messages in the next interval won't be sent in another active state. The good thing is that this scenario seldom happens in the push service system. Therefore, we only consider $t_{threshold}$ is longer than the total transmission time in this interval, $(t_{ssa} + kt_{dt} + t_{it} + t_{ssb}) < t_{threshold}$. In a time interval $t_{threshold}$, since messages are transmitted in one same state, there will be no messages in $\mathcal{M}_{mt}$ and $\mathcal{M}_{lt}$ except the first message. Therefore, E_{com} under SCoP system can be calculated as follows:

$$E_{com} = kE_{dt}(M) + rE_{ssit}(M), \quad when\ (t_{ssa} + kt_{dt} + t_{it} + t_{ssb}) < t_{threshold}. \quad (13)$$

r represents the number of intervals with data transmission in a time period t. To get the value of r, we calculate the probability of receiving zero message in $t_{threshold}$ first. The probability of receiving zero message in $t_{threshold}$ is:

$$f(0; \lambda t_{threshold}) = e^{-\lambda t_{threshold}}. \quad (14)$$

We calculate the expectation of E_{com} under SCoP system in a time period t:

$$\exp(E_{com}) = \lambda t E_{dt}(M) + ((1 - e^{-\lambda t_{threshold}})t/t_{threshold})E_{ssit}(M), \quad when\ (t_{ssa} + kt_{dt} + t_{it} + t_{ssb}) < t_{threshold}. \quad (15)$$

The expected energy cost under SCoP system can be calculated as follows:

$$\exp(E_{SCoP}) = \sum_{i=1}^{n} E_{local}(a_i) + E_{local}(a_{lp}) + E_{conn}(s_{rp}) + \exp(E_{com}) + E_{hb}. \quad (16)$$

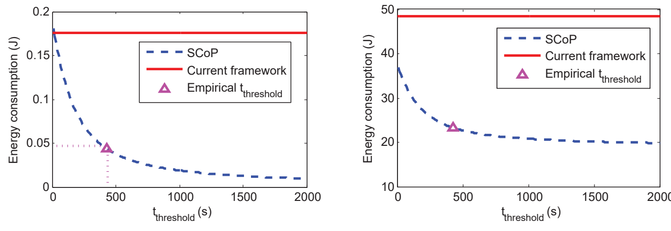
D. Performance Analysis

Recall Equation (11) and Equation (16), we only need to compare $(\sum_{j=1}^{m} E_{conn}(s_j) + \exp(E_{com}))$ and $(E_{local}(a_{lp}) + E_{conn}(s_{rp}) + \exp(E_{com}))$ to evaluate the performance of SCoP. First, for both real-time and delay-tolerant apps, the number of connections has been reduced in SCoP system, $E_{conn}(s_{rp}) < \sum_{j=1}^{m} E_{conn}(s_j)$ when $m > 1$. Second, for delay-tolerant apps, we compare $\exp(E_{com})$ based on Equation (10) and Equation (15). Clearly, there is no $\overline{E_{it}(M)}$ part in SCoP system since $\mathcal{M}_{mt} = \Phi$, and $E_{dt}(M)$ part is the same. For $E_{ssit}(M)$ part, we can simply compare $(1 - e^{-\lambda t_{threshold}})/t_{threshold}$ and $\lambda e^{-\lambda(t_{dt}+t_{it})}$. Since $E_{ssit}(M)$ and t are positive real numbers, if we want SCoP performs better in $E_{ssit}(M)$ part, $t_{threshold}$ follows:

$$(1 - e^{-\lambda t_{threshold}})/t_{threshold} < \lambda e^{-\lambda(t_{dt}+t_{it})}. \quad (17)$$

We give an empirical $t_{threshold}$ to reduce the energy cost of a smartphone without causing too much delay. Using the empirical values, $\lambda = 1/100$, $E_{ssit}(M) = 19J$ and $t_{dt} + t_{it} = 8s$, we can get the energy cost of $E_{ssit}(M)$ part under SCoP and the current framework (these empirical values are proved to be acceptable in our later experiments). The result is depicted in Fig. 10-a. We can infer that SCoP saves more energy with larger $t_{threshold}$. The energy cost of SCoP goes down quickly when $t_{threshold}$ increases at the beginning. Then it decreases slowly and comes close to the asymptote $x = 0$. We give an empirical $t_{threshold}$, which makes SCoP cost 75% less than the current framework in $E_{ssit}(M)$ part without causing too much delay.

Moreover, we compare the expected energy cost between SCoP ($\exp(E_{SCoP})$) and the current framework ($\exp(E_{CF})$). Using the empirical values, $E_{local}(a_i) = 0.1J$, $E_{conn}(s_j) = 4J$, $E_{hb} = 2.3J$ in 100 seconds, and $n = 10$, $m = 4$, $E_{dt}(M) = 12J$, $\overline{E_{it}(M)} = 9J$, we plot the expected energy cost-$t_{threshold}$ curves under SCoP and the current framework by setting $t = 100s$ and $\lambda = 1/100$. The result is depicted in Fig. 10-b. Generally speaking, SCoP costs 50% energy less than the current framework under $\lambda = 1/100$ theoretically.



(a) $E_{ssit}(M)$ in a time unit. (b) Total energy cost in 100s.

Fig. 10. Theoretical comparison when $\lambda = 1/100$.

V. Prototype Implementation

We implement a prototype of our proposed SCoP system. We describe detailed implementations of the local and remote proxies.

A. Local Proxy Implementation

The local proxy is implemented on Android OS 6.0 based on JAVA. Since starting up a server requires `INTERNET` permission, we add it into `AndroidManifest.xml` file of the local proxy.

The communication between the two proxies is implemented by JAVA socket. In the local proxy, we use the `InputStream` and `OutputStream` to handle incoming and outgoing messages, and use serialized stream information as header.

Since the structure of the message is designed by us, we cannot use high level functions in JAVA for sending and receiving. We use `OutputStream.write` function to send messages. For receiving messages, we first use `InputStream.read` to get the first four bytes of the message as its length, and then use a while loop to keep receiving data till the length bytes of data are received.

B. Remote Proxy Implementation

The remote proxy is implemented on Ubuntu 14.04 based on C. After the server starts up, a while loop is used to keep accepting new clients. A thread is started to handle incoming and outgoing messages once a client is accepted. The steps of receiving message are similar to those on the local proxy. But for sending message, we use a while loop to send till the length bytes of data are sent.

To work as a man-in-the-middle proxy, the remote proxy responds with `200 Connection Established` once it receives `CONNECT` request. The client initiates SSL handshake with server name indication (SNI). Then the remote proxy establishes an SSL connection with the server using the SNI hostname, and the server responds with the original certificate. Finally, the remote proxy generates the interception certificate using `CN` field and `SAN` field from the original certificate and establishes SSL connection with the client.

For the store-and-forward function on the remote proxy, we use a queue for each client to store the messages to be forwarded. First, we set the timer to $t_{threshold}$. Once the timer counts down to zero and the queue is not empty, the remote proxy keeps sending the message which is the head of the queue till the queue is empty. Then the timer is set to $t_{threshold}$ again. The remote proxy waits for $t_{threshold}$ to send next time. If the queue remains empty when timer becomes zero, the remote proxy also waits for $t_{threshold}$ to send next time.

VI. Experiment

To evaluate the performance of SCoP system precisely, we test it under real-world experiments. We first describe our experiment setup and then discuss the experimental results.

A. Setup

We evaluate the performance of SCoP under 3G network on a smartphone equipped with Android OS 6.0. We use artificial traffic to test SCoP system, and the inactivity timer of the smartphone is 4s. To get a fine-grained measurement of the smartphone's energy consumption, we use a power meter, Monsoon Power Monitor [14], to measure the real-time current of the smartphone, as depicted in Fig. 11.

First, for real-time apps, we evaluate the real-time performance of merging connections. We show a real-time current

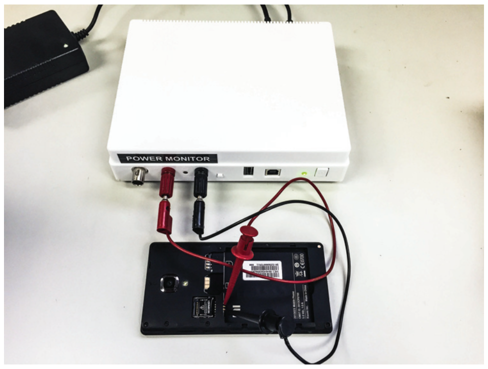


Fig. 11. Energy consumption measurement using Monsoon Power Monitor.

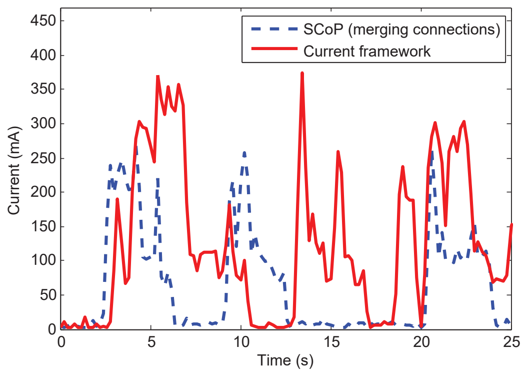


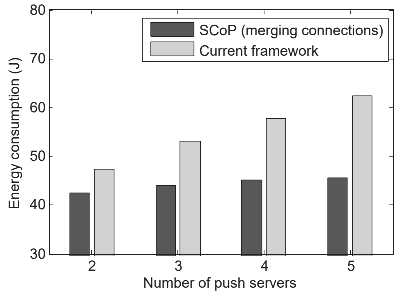
Fig. 12. Real-time energy cost comparison between SCoP (merging connections) and the current framework.

comparison between SCoP and the current framework under four push servers (Facebook uses GCM, QQ uses XG push, Sina Weibo uses Getui push and inbilin uses JPush). Here we set $t_{threshold} = 0$ because we only want to evaluate the performance of merging connections.

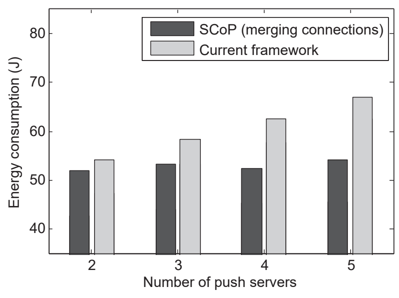
Second, for real-time apps, we evaluate the performance of merging connections in more details. A unit interval is one second. We adopt two to five apps which use different push services on the smartphone, and compare the cumulative energy cost of receiving ten push messages in total under different rates ($\lambda = 1/6$, $1/12$ and $1/30$) using SCoP and the current framework.

Third, for delay-tolerant apps, we evaluate the performance of the store-and-forward function. We compare the cumulative energy cost of one app (QQ) when receiving ten messages. We show the energy cost-$t_{threshold}$ curves under SCoP and the current framework with different λ, from 1/120 to 1/30. The value of λ is acceptable under push service scenario based on our experience.

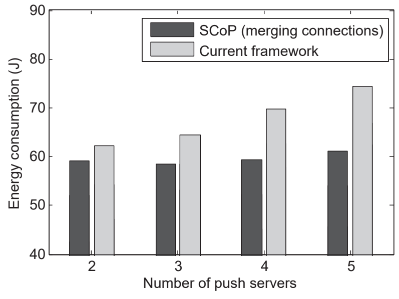
Finally, for delay-tolerant apps, we combine merging connections and store-and-forward function to show overall energy cost comparisons between SCoP and the current framework under four push servers when $\lambda = 1/30s$. We use our empirical delay time as $t_{threshold}$ ($t_{threshold} = 156s$). We show the energy consumption-number of messages curves of SCoP and the current framework.



(a) $\lambda = 1/6$.

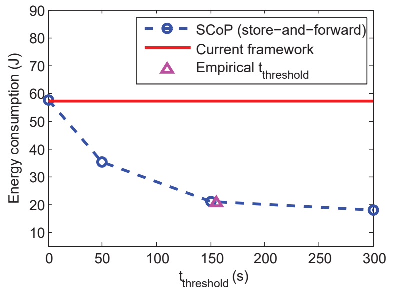


(b) $\lambda = 1/12$.

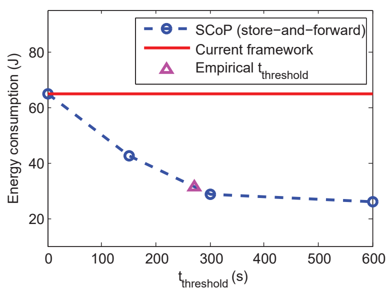


(c) $\lambda = 1/30$.

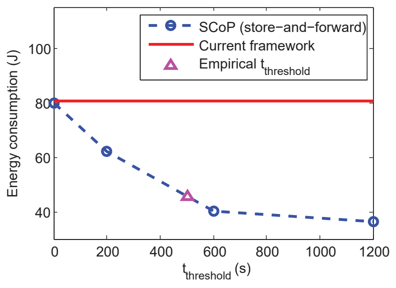
Fig. 13. The cumulative energy cost comparisons between SCoP (merging connections) and the current framework when receiving ten messages.



(a) $\lambda = 1/30$. Empirical $t_{threshold} \approx 156$.



(b) $\lambda = 1/60$. Empirical $t_{threshold} \approx 271$.



(c) $\lambda = 1/120$. Empirical $t_{threshold} \approx 505$.

Fig. 14. Energy cost comparisons between SCoP (store-and-forward) and the current framework when receiving ten messages under different λ.

B. *Experimental Result*

Real-time apps: real-time energy cost of merging connections. Fig. 12 shows a real-time comparison between SCoP (merging connections) and the current framework. At the beginning (0 to 2s), there is almost no energy consumption when no message needs to be transmitted. We can infer that $E_{local}(a_{lp})$ is quite small, approximate to zero. Generally speaking, the energy cost of real-time apps under SCoP is lower than that under the current framework. The peak current under SCoP is about 260mA, while the peak current under the current framework is nearly 400mA.

Real-time apps: cumulative energy cost of merging connections. Fig. 13 shows that the cumulative energy cost of SCoP does not increase with the number of push servers. The performance of SCoP can be regarded as unchanged with an increasing number of push servers. Specifically speaking, it only costs 70% energy under the current framework in five push servers' scenario when λ=1/6 for real-time apps. Another interesting thing is that the performance of SCoP degrades with smaller λ. It is because E_{hb} becomes not negligible when receiving the same number of messages under smaller λ. Generally speaking, SCoP performs better in multiple push servers' scenario.

Delay-tolerant apps: cumulative energy cost of store-and-forward. Since the performance of SCoP remains unchanged with an increasing number of push servers, it is acceptable to evaluate the performance of the store-and-forward function under one push server. From Fig. 14, we can find that the trend of energy cost is as same as we expected in Fig. 10, and the empirical $t_{threshold}$ is acceptable. Generally speaking, the store-and-forward function can reduce nearly 4J energy per message for delay-tolerant apps under the empirical delay time.

Delay-tolerant apps: overall comparison. Fig. 15 shows overall comparisons between SCoP and the current framework. SCoP reduces about 60% energy cost of the current framework for delay-tolerant apps under four push servers when $\lambda = 1/30$.

Discussion. The experimental results show that SCoP system can reduce the energy cost in two ways: merging connections to reduce the cost of keep-alive connections; and the store-and-forward function to reduce radio-tails. SCoP is much more energy saving than the current framework. It minimizes a smartphone's energy consumption within the constraint of a predefined message delay requirement.

VII. Related Work

Cutting down radio-tails. Radio-tails consume a great part of energy in both 3G and 4G cellular networks [15]. The methods to reduce radio-rails have been discussed by many papers (e.g. [3], [4], [10], [16], [17], [18], [19], [20] etc.). Generally speaking, they can be separated into three categories: inactivity timer reconfiguration, tail cutting, and tail sharing [10].

Inactivity timer reconfiguration is to configure the inactivity timers. Jin et al. [21] point out that the inactivity timers should be configured dynamically according to the user's behaviors and the battery restriction. Falaki et al. plot the CDF of packet inter-arrival times, and find that the inter-arrival time values for most messages (95%) are relatively small (smaller than 4.5s).

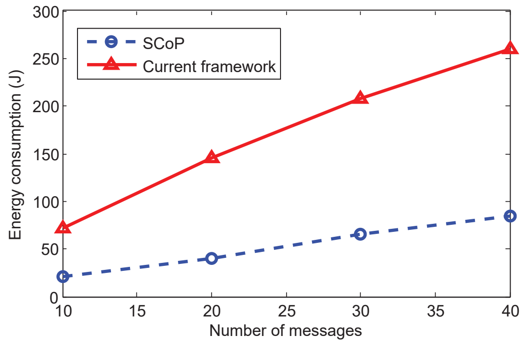


Fig. 15. Overall energy cost comparisons between SCoP and the current framework. $\lambda = 1/30$, $t_{threshold} = 156$.

Therefore, they set the inactivity timer to a fixed value (4.5s). Deng et al. [10] configure the inactivity timers dynamically using traffic pattern information within a short period without the prediction of app's traffic.

Tail cutting is forcing a smartphone to change to idle state right after data are transmitted using fast dormancy. However, the energy cost of state changing is not negligible. Qian et al. [4] propose *TOP* algorithm which enables cooperation between the smartphone and the radio access network to eliminate radio-tails. *TOP* decides whether to trigger fast dormancy based on the traffic information provided by apps on the smartphone.

Tail sharing is to coalesce separate data transfers by delaying some of them. Balasubramanian et al. [3] propose *TailEnder* protocol to share radio-tails for delay-tolerant apps such as email. *TailEnder* schedules transfers to minimize the energy consumption while meeting user-specified deadlines. They also suggest 10 minutes of delay for such apps. This solution receives a good result, but is not practical for real-time apps such as browser. Qian et al. [22] investigate the optimality of UMTS state machine configurations and propose an algorithm to reduce radio-tails between transfers of video pieces for YouTube using the idea of prefetching.

Merging multiple connections. The energy cost of establishing and maintaining network connections is not negligible, especially for keep-alive connections when push services become more and more popular. To the best of our knowledge, this is the first paper to save energy by merging different connections in the push service scenario.

VIII. Conclusion

Communication apps require keep-alive connections to notify users of new messages, which can drain a smartphone's battery quickly in cellular networks. We propose SCoP system based on fog computing to reduce the energy cost of multiple push services on a smartphone in home and company networks. SCoP can successfully reduce multiple keep-alive connections and minimize radio-tails without too much additional energy cost. The novel architecture of SCoP is practical in the real world, and needs no changes in today's push service framework. To show the performance of SCoP, we build an energy cost model and compare the energy cost between SCoP and the current framework. We give an empirical delay time based on our theoretical analysis. We also implement a prototype of SCoP system and evaluate the performance of SCoP with today's apps. The experimental results show that SCoP is much more energy saving than the current framework and the empirical delay time is acceptable.

Acknowledgement

This paper is partially supported by HK PolyU G-YBAD and G-YBJV.

References

[1] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-Efficient Computation Offloading in Cellular Networks," in *Proc. of IEEE ICNP*, 2015.
[2] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese, "Radiojockey: Mining Program Execution to Optimize Cellular Radio Usage," in *Proc. of ACM MOBICOM*, 2012.
[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," in *Proc. of ACM SIGCOMM*, 2009.
[4] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation," in *Proc. of IEEE ICNP*, 2010.
[5] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, "Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services," in *Proc. of ACM CCS*, 2014.
[6] "Websites blocked in mainland china." https://en.wikipedia.org/wiki/Websites_blocked_in_mainland_China.
[7] JPush, "JPush Energy Consumption." https://docs.jiguang.cn/jpush/guideline/faq/.
[8] "Fog computing." https://en.wikipedia.org/wiki/Fog_computing.
[9] L. M. Vaquero and L. Rodero-Merino, "Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing," *ACM SIGCOMM Computer Communication Review*, 2014.
[10] S. Deng and H. Balakrishnan, "Traffic-aware Techniques to Reduce 3G/LTE Wireless Energy Consumption," in *Proc. of ACM CoNEXT*, 2012.
[11] A. Cortesi, "mitmproxy." https://mitmproxy.org/.
[12] C.-h. Lee and M. Haenggi, "Interference and Outage in Poisson Cognitive Networks," *IEEE Trans. on Wireless Communications*, vol. 11, no. 4, pp. 1392–1401, 2012.
[13] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile Data Offloading: How Much Can WiFi Deliver?," *IEEE Trans. on Networking*, vol. 21, no. 2, pp. 536–550, 2013.
[14] "Monsoon power monitor." http://www.msoon.com/LabEquipment/PowerMonitor/.
[15] F. Yu, G. Xue, H. Zhu, Z. Hu, M. Li, and G. Zhang, "Cutting without Pain: Mitigating 3G Radio Tail Effect on Smartphones," in *Proc. of IEEE INFOCOM*, 2013.
[16] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li, "Optimizing Background Email Sync on Smartphones," in *Proc. of the ACM MOBISYS*, 2013.
[17] J. Li, K. Bu, X. Liu, and B. Xiao, "ENDA: Embracing Network Inconsistency for Dynamic Application Offloading in Mobile Cloud Computing," in *Proc. of the ACM SIGCOMM workshop on Mobile Cloud Computing (MCC)*, 2013.
[18] J. Li, Z. Peng, B. Xiao, and Y. Hua, "Make Smartphones Last A Day: Pre-processing Based Computer Vision Application Offloading," in *Proc. of the IEEE International Conference on Sensing, Communication and Networking (SECON)*, 2015.
[19] J. Li, Z. Peng, S. Gao, B. Xiao, and H. Chan, "Smartphone-Assisted Energy Efficient Data Communication for Wearable Devices," *Computer Communications*, 2016.
[20] S. Gao, Z. Peng, B. Xiao, and Y. Song, "Secure and Energy Efficient Prefetching Design for Smartphones," in *Proc. of the IEEE International Conference on Communications (ICC)*, 2016.
[21] S. Jin and D. Qiao, "Numerical Analysis of the Power Saving with a Bursty Traffic Model in LTE-Advanced Networks," *Elsevier Trans. on Computer Networks*, vol. 73, pp. 72–83, 2014.
[22] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Characterizing Radio Resource Allocation for 3G Networks," in *Proc. of ACM SIGCOMM*, 2010.