eTrain: Making Wasted Energy Useful by Utilizing Heartbeats for Mobile Data Transmissions

Tan Zhang[1] Xian Zhang[1] Fangming Liu[1]* Hongkun Leng[1] Qian Yu[1] Guanfeng Liang[†]
[1]Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China.

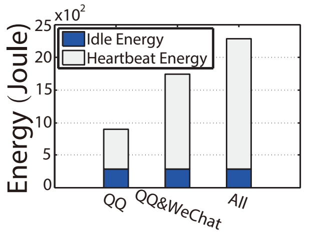
***Abstract*—With the rapid proliferation of smartphones, hundreds of millions of mobile users are attracted to Instant Messaging (IM) apps. While such apps have brought convenience to our life, it comes with the price of great energy consumption, as these apps keep sending heartbeat messages to the server periodically in order to maintain an always-online connection. These frequent and fragmented transmissions result in a considerable amount of energy waste.**

In this paper, we investigate the "cost and potential of heartbeats". We quantify power consumption of heartbeats of real-world IM apps through extensive measurements. The measurement results confirm that huge power consumption is induced by heartbeats. The goal of this paper is to save energy by turning the energy wastage of heartbeats into transmitting useful data. Thus, we develop *eTrain*, a transmission management system running on Android phones, which takes advantage of IM heartbeats (as trains) to piggyback aggregated delay-tolerant apps' data such as e-mail and Weibo (as cargoes) via an online transmission strategy, so as to minimize the cumulative tail energy without sacrificing user-specified deadlines. Compared to other existing works, *eTrain* can reduce more energy consumption under the same settings. Experiments conducted on smartphones show that *eTrain* can achieve 12%–33% energy saving in various application scenarios.

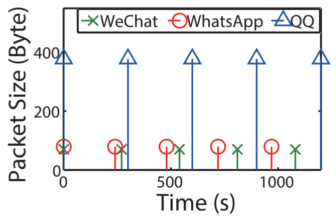
I. Introduction

Smartphones are ubiquitous today with a predicted number of 7,278 million users worldwide in 2015 [1], which become the main impetus of the booming mobile markets. Since the emerging mobile applications (apps) support real-time communication services (e.g., WeChat [2], WhatsApp [3]), they are often constantly running in background mode to receive status updates or messages from other parties. Thus, the applications continuously generate short signaling messages such as keep-alive requests to maintain the always-on connectivity. These requests are known as heartbeats, which consumes a large amount of energy on smartphones due to the tail energy. Previous studies [4], [5] have pointed out that when a wireless transmission completes, it will be followed by a "tail" during which the radio interface lingers in a high power state.

To make sure that heartbeat transmissions do consume a lot of energy, we conduct a measurement on a Samsung Galaxy S4 smartphonehe. Fig. 1(a) illustrates the overall power consumption of the smartphonehe for a 4-hour period with



(a) Energy consumption of different number of active IM apps



(b) Heartbeat activities of WeChat, WhatsApp and QQ

Fig. 1: Energy consumption and heartbeat activities of 3 IM apps

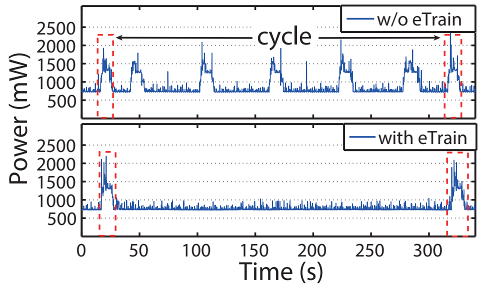


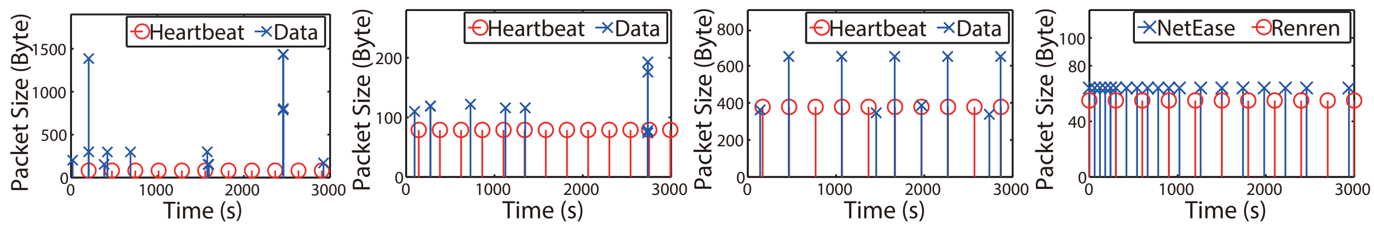
Fig. 2: Example of piggybacking implemented by *eTrain*

different number of active IM apps (Mobile QQ [6], WeChat and WhatsApp). To simulate the smartphone in standby mood, we turn the display off and shut down all other irrelevant tasks during the period. As we can see from the third bar in the figure, in 3G environment, the smartphone spends nearly 87% energy (2000 Joules) on transmitting their heartbeats with all these apps running, which corresponds to roughly 10 hours of standby time. Fig. 1(b) shows the size and timing of heartbeats generated by these three IM apps running simultaneously with no data communication. We can observe that heartbeat transmissions are frequent, once a minute on average. Despite the high energy cost, heartbeats are indispensable in many application scenarios to provide an always-connected user experience. Also, it is practically difficult to regulate mobile application developers in how they issue/use heartbeats. Hence, in this paper, instead of trying to regulate or reduce heartbeats, we focus on methods to better utilize heartbeats and turn their energy wastage into transmitting useful data.

An important observation in [5] is that the amount of tail energy is basically a constant and is independent of the preceding data transmission. As a result, the longer the preceding data transmission lasts, the smaller fraction of energy is wasted in the tail relative to that used for data transmissions. Thus, a natural idea to reduce tail energy wastage is to aggregate fragmented data transmissions (e.g., Binding several emails and sending them together). We can use heartbeats as a signal/indicator of communication opportunity

*The Corresponding Author is Fangming Liu (fmliu@hust.edu.cn). The research was supported by a grant from the National Basic Research Program (973 Program) under grant NO.2014CB347800.

[†]This work was done when Guanfeng Liang (guanfeng.liang@gmail.com) collaborated with HUST as an independent contributor not associated with his current affiliation.



(a) WeChat (b) WhatsApp (c) Mobile QQ (d) NetEase and RenRen

Fig. 3: Heartbeats timing and size of mobile applications on Android

and piggyback the aggregated data onto the tail of a heartbeat (the radio interface's high power state immediately following the heartbeat transmission). In this way, the tails of heartbeats are not wasting energy any more but transmitting actual data instead.

In this paper, we introduce *eTrain*, a transmission management system that implements the above idea of aggregating and piggybacking. We regard the three aforementioned IM apps' heartbeats as trains to piggyback delay-tolerant data (as cargoes) created by three apps we developed (Luna Weibo, *eTrain* mail and *eTrain* cloud). Heartbeats have two advantages to implement piggybacking. One is their short cycles. Different from other kinds of periodic transmissions like system and app updates which generally occurs once every few hours, heartbeat packets are transmitted every few minutes. More transmissions mean more trains to piggyback cargoes. The other advantage is monitoring heartbeat is costless and user-friendly. An naive implementation of the above idea is to monitor all network traffic and transmit aggregated traffic at an arbitrary traffic point. However, doing so requires high level system permissions and a lot of hardware resources (CPU cycles, Memory, etc.). It will affect user experience and consume a lot of energy. On the contrary, monitoring heartbeat requires negligible resources. We will discuss more details about implementation issue for monitoring heartbeats in the Sec.V. For delay tolerant applications such as Weibo, email and cloud storage apps, *eTrain* selectively defers data transmissions to sync with future heartbeats, so that the tail energy spent by heartbeats is reused. Fig. 2 illustrates the effectiveness of piggybacking in *eTrain*. The upper half of Fig. 2 shows the power trace of one heartbeat cycle of a smartphone without *eTrain*, where two consecutive heartbeats are highlighted with red dashed boxes. During the heartbeat cycle, 5 scattered data transmissions (five 5-KB emails) are issued. The bottom half of Fig. 2 shows the case with *eTrain*, which defers and aggregates the 5 data transmissions along with the second heartbeat. The power trace shows that we can save nearly 40% of transmission energy by using *eTrain* in this toy example. Our detailed contributions are as follows.

- We have developed a formal optimization framework as well as an online transmission strategy to solve the tail energy minimization problem. Compared to other existing strategies in simulation experiments, our online algorithm has better performance on energy saving and delay cost.
- We have developed Weibo and Email apps with *eTrain* over Android and these energy-efficient apps have more than 100 users from different areas. We collect extensive user traces for our experiments.
- Extensive simulations and real-world experiments demonstrate that each component of *eTrain* can cooperate seamlessly with each other, and achieve significant energy saving in various scenarios.

The paper is organized as follows: Section II presents the research finding of heartbeat mechanism. Section III and Section IV introduce our problem formulation and online transmission strategy. Section V presents the implementation of *eTrain*. Section VI evaluates the performance of *eTrain* based on the collected traces. Section VII reviews related work and Section VIII concludes the paper.

II. Background

In this section, we will introduce heartbeat mechanism and our measurement results.

A. Heartbeat Transmission Mechanism

IM apps such as WhatsApp, WeChat and Mobile QQ adopt a store-and-forward message delivery mechanism. To send a message (text or multimedia) to one or more recipients, the message needs to be uploaded to a server, and then the server pushes a notification to the corresponding receiver(s). Once a receiver is notified, it downloads the message from the server [7]. Each IM app keeps a TCP connection between client device and the server to ensure near real-time delivery of messages, so that an interactive conversation can be conducted. To keep the TCP connection alive, these apps independently create a daemon thread which sends heartbeats periodically to refresh the TCP connection timeout counter. These daemon threads are kept alive in the background even when the main program (thread) of an app has been closed [8]. Other types of applications like News and SNS (social networking services) also apply this heartbeat mechanism to keep a constant connection for receiving in-time updates such as the latest breaking news and friends' messages, etc.

B. Heartbeat In Action

To demonstrate and verify that heartbeats widely exist in numerous apps that need to fetch messages, we conduct experiments on several popular applications (downloaded more than 100 million times): Mobile QQ, WeChat, WhatsApp, NetEase news [9], RenRen SNS (Social Networking Service) [10]. We perform our measurements on five smartphones running iOS and Android separately. In all experiments, the tested mobile devices are connected to a dedicated WiFi network, and we monitor network traffic while the applications are in use. We capture raw packets using Wireshark (A software used to capture packets) and analyze the captured traffic file offline

TABLE I: Heartbeat cycles in various apps

App (version)	WeChat (5.3.1.67)	WhatsApp (2.11.304)	QQ (4.7.2)	RenRen (7.5.6)	NetEase (3.9.3)
HTC Sensation Z710e	270s	240s	300s	300s	60–480s
Samsung Note II	270s	240s	300s	300s	60–480s
Samsung GALAXY S IV	270s	240s	300s	300s	60–480s
iPhone 4/iPhone 5	1800s	1800s	1800s	1800s	1800s

to determine the heartbeat cycle. Table I shows the heartbeat cycles of the aforementioned apps. As we can see, different applications generally employ different heartbeat cycles on the Android platform, while all apps share the same heartbeat cycle on iOS. This is because Apple forces all apps running on its iOS devices to use the Apple Push Notification Service (APNS) for notifications, which maintains only one TCP connection for heartbeat-sending apps. Google provides a similar cloud service for Android apps named GCM (Google Cloud Messaging), but it is optional. Actually, many commercial apps on Android adopt their own heartbeat services.

We measure the heartbeats with data communication presented. In particular, text messages and pictures are sent within the IM apps during the measurement. Fig. 3(a), Fig. 3(b) and Fig. 3(c) show the heartbeat cycles and the data transmissions of these apps. The result shows these data packet transmission has no impact on the timing for heartbeat transmissions. As we can see, other than NetEase news app, all tested apps have fixed heartbeat cycles. Fig. 3(d) shows that, the NetEase news app has an initial heartbeat cycle of 60s, and doubles the cycle after every 6 heartbeats are sent until it reaches 480s while RenRen SNS has a constant heartbeat cycle (300s).

Based on our measurement results, we find that heartbeat sending is deterministic with respect to their individual heartbeat cycle. We can take advantage of this knowledge for piggybacking data transmissions.

C. Energy Consumption of Different Stages for a Transmission

On smartphones, in addition to the inherent energy consumption of CPU processing and display, a significant portion of battery energy is consumed by wireless network interface, especially by mobile apps which require heavy data transmissions [11]. To investigate the energy consumption, we use Samsung Galaxy S4 as the test platform and conduct a number of measurements in a UTMS network in China (TD-SCDMA).

We measure the power state transitions for one heartbeat transmission over the 3G cellular interface with power monitor [12]. Fig. 4 shows that before the data transmission begins, the smartphone stays at a low power state, namely the IDLE (Idle Channel) state. Upon the initiation of a data transmission, the smartphone is promoted to a higher power state DCH (Dedicated Channel). When the transmission completes, the network interface lingers in DCH for a time duration δ_D and then demotes to a moderate power state FACH (Forward Access Channel). Then, the radio is further demoted to IDLE after staying in FACH for δ_F. We call the time period from the end of a data transmission to the time when the radio is demoted back to IDLE as the *tail period*, and the length of the tail period is denoted as $T_{tail} = \delta_D + \delta_F$, named as the *tail time*. The tail period is originally introduced to reduce radio state promotion delay and signaling overhead, as well as to mitigate frequent power state transitions and

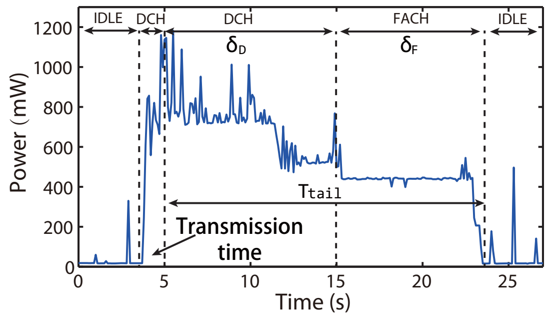


Fig. 4: Instantaneous power level at different power states

channel allocations/releases. But in reality, it is one of the major sources of energy consumption.

D. Energy Consumption of Heartbeats

Our measurement shows that mobile apps such as WeChat in general send more than 12 heartbeats per hour. Since heartbeats are small in size, most of their energy cost comes from the tail. From our measurement, in 3G environment a tail costs about 10.91 Joules. Given a battery capacity of 1700mAh with voltage 3.7V, if the battery life is 10 hours, the smartphone will spend at least 6% of its battery capacity on sending heartbeats of only one app. This is a huge waste of energy and should be made better use of.

E. eTrain in a Nutshell

As we shown in Fig. 5, *eTrain* mainly consists of three modules: the *Heartbeat Monitor* module is responsible for monitoring heartbeat activities of the train apps. It will inform *eTrain Scheduler* of heartbeat transmission times; the *eTrain Scheduler* module implements our online algorithm to schedule cargo apps' data transmissions; and the *eTrain Broadcast* module delivers the scheduler's decisions to the cargo apps. *eTrain* is transparent to the heartbeat-sending apps.

III. Models And Problem Formulation

In this section, we introduce the energy consumption model and the delay cost model, based on which we formulate the tail energy minimization problem.

A. Energy Consumption Model

Power consumption of a transmission over the cellular network consists of a transmission energy and a tail energy. The transmission energy corresponds to energy consumption incurred during actual data transmission and it is generally proportional to the data transmission time [13]. On the other hand, the tail energy represents the energy consumed within the tail period, during which data transmission is already finished. The goal of our work is to minimize the tail energy wastage.

We consider a "data packet" or simply a "packet" as an *application-layer data unit* which may constitute a group of transmission-layer packets that should be transmitted altogether. Examples for application-layer data packets are a text message, a piece of news update, etc. The actual amount of tail energy wasted for the transmission of a particular packet u depends on the length of the interval between the end of u's transmission and the start of the subsequent packet, say q. Suppose u is scheduled to be transmitted (either sent or received) at time $t_s(u)$ and the transmission lasts for $t_l(u)$, the next packet q is to be transmitted at $t_s(q)$. Then the gap between the two packets is $\Delta = t_s(q) - (t_s(u) + t_l(u))$. Denote p_I, p_D, and p_F as power levels of the IDLE, DCH and FACH states, respectively. Further define $\tilde{p}_D = p_D - p_I$ and $\tilde{p}_F = p_F - p_I$, then using p_I as the baseline, the tail energy wastage $E_{tail}(\Delta)$, i.e., the extra tail energy consumed during the gap, can be computed by

$$E_{tail}(\Delta) = \begin{cases} 0 & \text{if } \Delta \leq 0 \\ \tilde{p}_D \Delta & \text{if } 0 < \Delta \leq \delta_D \\ \tilde{p}_D \delta_D + \tilde{p}_F(\Delta - \delta_D) & \text{if } \delta_D < \Delta \leq T_{tail} \\ \tilde{p}_D \delta_D + \tilde{p}_F \delta_F & \text{otherwise.} \end{cases}$$

The four conditions in the above formulation represent the four cases when transmission of q starts: (1) before u finishes; (2) before demoting to FACH; (3) before demoting to IDLE; and (4) after demoting to IDLE.

B. Delay Cost Model

eTrain aggregates scattered data transmissions for energy saving, which inevitably incurs extra delays. However, many apps are delay-tolerant, for which transmissions can be scheduled quite flexibly with little impact on user experiences. For example, SNS (e.g., Twitter, Weibo, Facebook) apps are among the most popular categories of apps on smartphones and are checked periodically by users. On the other hand, cloud storage apps like iCloud and Dropbox, deferring the synchronization of newly generated mobile data for a short period, may not hurt user experiences significantly as well.

For the same amount of additional delay, it may be perceived as severe performance degradation for time-sensitive apps such as SNS ones whose (application-layer) packets are generally small. On the other hand, it is acceptable or even negligible for apps such as cloud storage ones which tend to have much larger (application-layer) packets, i.e., much longer transmission time. Intuitively, the larger a packet is in size the less sensitive to additional delays it should be. Suppose a packet u arrives at time $t_a(u)$, then the additional delay incurred is $delay = t_s(u) - t_a(u)$. Therefore we develop the following delay cost function for deferred transmission u as

$$\phi_u(delay) = w_i f_i(delay)/t_l(u) = w_i B(u) f_i(delay)/size(u),$$

where $size(u)$ denotes the size of packet u, $B(u)$ denotes the average bandwidth received by u and $t_l(u) = size(u)/B(u)$ is the corresponding transmission time. Weighting coefficient w_i reflects the user's preference on the application i that generates u. The profile function $f_i(delay)$ characterizes how the performance degrades with the increase of delay. And specific profile functions for several typical applications would be presented in the evaluation later.

C. Problem Formulation

Though heartbeats in various apps incur heavy power consumption, they are still necessary for their hosting apps to fetch updates. Any modification on the heartbeat cycle can bring unexpected side-effects to apps. So our *eTrain* solution for energy saving will not try to reduce heartbeats. Instead, we use heartbeats as opportunities for transmitting delay-tolerant data: the high power state triggered by heartbeats acts as carriages of a train for data transmissions to piggyback on. We consider a mobile user employs *eTrain* to manage N heterogeneous train apps which send heartbeats periodically and M cargo apps that generate data packets. A typical app may belong to both categories, e.g., SNS apps send heartbeats for getting periodic notifications (train) and can also generate cargo packets when the user browses friends' profiles.

Define $\mathbf{H}_i$ as the set of heartbeats generated by train app i and $h_{i,j}$ as the jth heartbeat in $\mathbf{H}_i$. As we have observed in our measurements, the heartbeat cycle of a train app, denoted as $cycle_i$, is quite stable. So as soon as *eTrain* observes one heartbeat of a train app, it can accurately predict when the subsequent heartbeats of the same train app will be transmitted according to $t_s(h_{i,j}) = t_{s,i,0} + cycle_i \times j$, where $t_{s,i,0}$ is the time when the first heartbeat is transmitted for train app i, assumed to be known. Define $\mathbf{H} = \bigcup_{i=1}^{N} \mathbf{H}_i$ as the union of all heartbeats generated by the N train apps. We call the transmission times $\{t_s(h)|h \in \mathbf{H}\}$ the "train departure times".

Let $\mathbf{U}_i$ be the set of all data packets generated by cargo app i and $u_{i,j}$ be the jth packet in $\mathbf{U}_i$. Without loss of generality, we assume $t_a(u_{i,j}) \leq t_a(u_{i,j+1})$ for all i and j. Also define $\mathbf{U} = \bigcup_{i=1}^{M} \mathbf{U}_i$ as the union of all packets from the M cargo apps. The job of *eTrain* is to design a transmission schedule $\mathbf{S} = \{t_s(u)|u \in \mathbf{U}\}$ for all data packets, given the knowledge of the train departure times of all heartbeats, so that the total tail energy wastage is reduced without significantly degrading user perceived delay performance.

Given a schedule $\mathbf{S}$, for any $x \in \mathbf{H} \cup \mathbf{U}$ (x can be either a heartbeat or a data packet), denote $x^- = \arg\max_q \{t_s(q)|q \in \mathbf{H} \cup \mathbf{U} \text{ and } t_s(q) < t_s(x)\}$ and $x^+ = \arg\min_q \{t_s(q)|q \in \mathbf{H} \cup \mathbf{U} \text{ and } t_s(q) > t_s(x)\}$ as the last heartbeat or packet transmitted prior to x and the first heartbeat or packet transmitted after x, respectively. Then the gap between x and x^+ is $\Delta(x) = t_s(x^+) - (t_s(x) + t_l(x))$, and the tail energy wastage of x is then $E(x) = E_{tail}(\Delta(x))$. Now we formulate the following tail energy minimization problem as

$$\min_{\mathbf{S}} \quad \sum_{h \in \mathbf{H}} E(h) + \sum_{u \in \mathbf{U}} E(u) \tag{1}$$

$$\text{s.t.} \quad t_s(u) \geq t_a(u) \qquad \forall u \in \mathbf{U} \tag{2}$$

$$t_s(x) \geq t_s(x^-) + t_l(x^-) \qquad \forall x \in \mathbf{H} \cup \mathbf{U} \tag{3}$$

$$\sum_{u \in \mathbf{U}} \phi_u(t_s(u) - t_a(u)) \leq \Theta \tag{4}$$

$$t_s(h_{i,j}) = t_{s,i,0} + cycle_i \times j \quad \forall h_{i,j} \in \mathbf{H}_i, i \leq N. \tag{5}$$

The first constraint (2) ensures causality, i.e., a packet cannot be transmitted until it has arrived. For simplicity of presentation, we assumed at most one heartbeat or data packet can be transmitted at any given time, which results in the second constraint (3). The third constraint (4) expresses a budget of the total delay cost acceptable to the client. The train departure times is captured by the last constraint (5).

Assuming perfect knowledge of the data packet arrival times $t_a(u)$ and the time-dependent bandwidth $B(u)$ (which determines $t_l(u)$), the above optimization problem is in fact a generalization of the classic *Knapsack problem* which is NP-hard. However, in reality, *eTrain* has little if not none knowledge of the future data packet arrivals as well as the time varying wireless channel condition. As a result, we focus on the design of an online transmission scheduling strategy.

IV. *eTrain* Online Transmission Strategy

To make the presentation uncluttered, in this section, we consider a slotted time model and assume all heartbeats are transmitted at the boundaries of slots. This will be a good approximation of the reality when the slot size is small enough compared to the heartbeat cycles. *eTrain* maintains one *waiting queue* Q_i for each cargo app i. All data packets generated by cargo app i is first enqueued into Q_i upon arrival. With a little abuse of notation, we use $Q_i(t)$ to denote the set of data packets in Q_i at the beginning of slot t, and correspondingly $\mathbf{Q}(t) = \bigcup_{i=1}^{M} Q_i(t)$. We further assume all data packets generated by cargo app i within slot t, denoted as $A_i(t)$, arrives by the end of slot t.

eTrain also maintains one FIFO (first-in-first-out) *transmission queue* Q_{TX}. Q_{TX} buffers packets that should be transmitted as soon as possible: whenever Q_{TX} is not empty and there is radio resource available, transmission of the HoL (head-of-line) packet of Q_{TX} starts immediately. At the beginning of every slot t, *eTrain* picks a subset of data packets $\mathbf{Q}^*(t) \subseteq \mathbf{Q}(t)$. $\mathbf{Q}^*(t)$ can also be written as $\bigcup_{i=1}^{M} Q_i^*(t)$ where each $Q_i^*(t) \subseteq Q_i(t)$ represents the subset of data packets picked from Q_i. These data packets are then removed from the waiting queues and injected into the transmission queue Q_{TX} for immediate transmission.

We then define the instantaneous cost of cargo app i at slot t as $P_i(t) = \sum_{u \in Q_i(t)} \phi_u(t - t_a(u))$, and the cost of slot t as

$$P(t) = \sum_{i=1}^{M} P_i(t). \tag{6}$$

Also define the *speculative cost* of a data packet $u \in \mathbf{Q}(t)$ as the cost of u in the subsequent slot $t+1$ if it is not included in $\mathbf{Q}^*(t)$: $\varphi_u(t) = \phi_u(t+1-t_a(u))$. Then according to our definitions of $P_i(t)$, $A_i(t)$ and $U_i^*(t)$, the queueing dynamics of the cost function can be characterized by:

$$P_i(t+1) = \sum_{u \in Q_i(t)} \varphi_u(t) - \sum_{u \in Q_i^*(t)} \varphi_u(t) + \sum_{u \in A_i(t)} \phi_u(0).$$

We have $\phi_u(0)$ in the last summation because we assume all packets of $A_i(t)$ arrive at the end of slot t, i.e., $t_a(u) = t+1 \quad \forall u \in A_i(u)$. Following the Lyapunov optimization framework [14], which has already been adopted to solve the power-performance tradeff problem not only in mobile devices such as [15], [16] but in Software-as-a-Service (SaaS) cloud platforms like [17], we define our Lyapunov function as: $L(t) = \frac{1}{2}\sum_{i=1}^{M} P_i^2(t)$. A small value of $L(t)$ implies that the delay cost of all pending data queues are small, while a larger value of $L(t)$ means that at least one application is suffering from severe performance degradation. Then we compute the one-step Lyapunov drift $\Delta(L(t)) = L(t+1) - L(t)$ as

$$\sum_{i=1}^{M} \left[\Phi_i(t) + \frac{\left(\sum_{u \in Q_i^*(t)} \varphi_u(t)\right)^2}{2} - \tilde{P}_i(t) \sum_{u \in Q_i^*(t)} \varphi_u(t) \right],$$

where $\Phi_i(t) = \frac{\left[\sum_{u \in Q_i(t)} \varphi_u(t) + \sum_{u \in A_i(t)} \phi_u(0)\right]^2 - P_i(t)^2}{2}$ and $\tilde{P}_i(t) = \sum_{u \in Q_i(t)} \varphi_u(t)$. Both $\Phi_i(t)$ and $\tilde{P}_i(t)$ are independent of the choice of $\mathbf{Q}^*(t)$.

Following the Lyapunov design principle, our online transmission strategy of *eTrain* targets in maximizing the negative drift of the Lyapunov function, i.e.,

$$\max_{\mathbf{Q}^*(t)} \sum_{i=1}^{M} \left[\tilde{P}_i(t) \sum_{u \in Q_i^*(t)} \varphi_u(t) - \frac{\left(\sum_{u \in Q_i^*(t)} \varphi_u(t)\right)^2}{2} \right] \tag{7}$$

Without additional constraints imposed, the above optimization is trivially solved by setting $\mathbf{Q}^*(t) = \mathbf{Q}(t)$, which means neither aggregation nor piggybacking is performed and as a result no energy is saved. Hence, we add the constraint

$$\mathbf{Q}^*(t)| \leq K(t) \tag{8}$$

to the above optimization problem. Here $K(t)$ is a time varying parameter that governs the maximum of the number of packets can be injected into Q_{TX} at slot t. Intuitively, $K(t)$ should be increased whenever some heartbeat(s) is/are transmitted at the beginning of slot t, i.e., $t = t_s(h)$ for some $h \in \mathbf{H}$, and it should be decreased otherwise. In particular, in our current implementation, $K(t)$ is modulated according to

$$K(t) = \begin{cases} k & \text{if } t = t_s(h)\ \exists h \in \mathbf{H} \\ 1 & \text{otherwise.} \end{cases}$$

Here $k > 1$ is a pre-defined parameter. Specifically, a larger k means that more packets *eTrain* can piggyback onto the tail of a heartbeat(if waiting queues Q_i have enough ones) and smaller delay. Actually, in our implementation, we set $k \leftarrow \infty$ to arbitrarily save energy meanwhile maintain the users' experience.

Noticing the optimization problem (1) is in fact a 0-1 integer programming problem, which is in general very difficult to solve optimally. We adopt an alternative greedy subgradient descend heuristic approach to find a near-optimal solution. The greedy heuristic involves at most $K(t)$ iterations, during each of which one new data packet is added into $\mathbf{Q}^*(t)$. Denote $\mathbf{Q}^*(t,r)$ and $Q_i^*(t,r)$ be the set of data packets selected after r iterations, both initialized to be empty for $r = 0$. Given $\mathbf{Q}^*(t,r)$, we find in the $(r+1)$th iteration a cargo app i and a data packet $u \in Q_i(t) \setminus Q_i^*(t,r)$ (i.e., u has not been selected yet) such that adding u into the current $\mathbf{Q}^*(t,r)$ will increase the objective function of (7) the most. In other words,

$$\max_{i, u \in Q_i(t) \setminus Q_i^*(t,r)} \tilde{P}_i(t) \sum_{q \in Q_i^*(t,r) \cup \{u\}} \varphi_q(t) - \frac{\left(\sum_{q \in Q_i^*(t,r) \cup \{u\}} \varphi_q(t)\right)^2}{2}.$$

With a bit of calculation, the overall optimization problem can be expressed as

$$\max_{i, u \in Q_i(t) \setminus Q_i^*(t,r)} \left(\tilde{P}_i(t) - \sum_{q \in Q_i^*(t,r)} \varphi_q(t) \right) \varphi_u(t) - \frac{\varphi_u(t)^2}{2} \tag{9}$$

$$\text{s.t.} \quad (2)(3)(4)(8),$$

which can be solved numerically very efficiently. Then we add the solution into the selected packets: $Q_i^*(t, r+1) = Q_i^*(t,r) \cup \{u\}$. The above is repeated until $K(t)$ packets are selected or there is no packet left. To further encourage energy saving, we force *eTrain* to not schedule any packet for transmission if the instantaneous total cost is lower than a certain threshold $\tilde{\Theta}$. In other words, $|\mathbf{Q}^*(t)| = 0$ if $P(t) < \tilde{\Theta}$. Overall, *eTrain*'s online transmission scheduling strategy can be summarized by the pseudo code in Algorithm 1.

Algorithm 1 *eTrain* online transmission strategy.

```
Input:
    $\tilde{\Theta}$, $\mathbf{Q}(t)$, $k$, $\mathbf{H}$, $t$ //the current time slot
Output:
    $\mathbf{Q}^*(t)$
 1: Calculate $P(t)$ from $\mathbf{Q}(t)$ according to Eq.(6)
 2: $\mathbf{Q}^*(t) \leftarrow \{\}, K(t) \leftarrow 0$
 3: if $P(t) \geq \Theta$ or $t = t_s(h)$ for some $h \in \mathbf{H}$ then
 4:     if $t = t_s(h)$ for some $h \in \mathbf{H}$ then
 5:         $K(t) \leftarrow k$
 6:     else
 7:         $K(t) \leftarrow 1$
 8:     end if
 9:     while $|\mathbf{Q}^*(t)| \leq K(t)$ and $|\mathbf{Q}(t)| > 0$ do
10:         Find $i$ and $u \in Q_i(t)$ that solves (9)
11:         $Q_i^*(t) \leftarrow Q_i^*(t) \cup \{u\}$, $Q_i(t) \leftarrow Q_i(t) \setminus \{u\}$
12:     end while
13: end if
14: Return $\mathbf{Q}^*(t)$
```

Notice here that, unlike our offline formulation (1), the above online formulation does not require any knowledge of the transmission bandwidth for the transmissions. There have been many attempts [18]–[20] to exploit the benefit of timing the transmissions according to the prediction of future network conditions. However, according to our previous experience, from application layer perspective, accurately predicting instantaneous wireless network condition is very difficult in reality. Such prediction can be quite expensive since it usually require channel scanning or probing, consuming a non-trivial amount of extra energy. Moreover, cellular service providers usually employ various traffic engineering techniques in their infrastructure such as congestion control, load balancing, pacing, flow rate control, etc., which add another layer of difficulty for instantaneous rate prediction. For these reasons, we consider the channel obliviousness an advantage of *eTrain*. Finding efficient ways for accurate channel prediction and making use of it is part of our future work.

V. IMPLEMENTATION OF *eTrain*

Based on our theory research shown in preceding sections, we implement *eTrain* as a broadcast system which monitors the train apps' heartbeat transmission activities and informs cargo apps the time to transmit aggregated data.

1) System Overview: There are four levels of Android system (Application, Framework, Libraries and Linux Kernel). As shown in Fig. 5, *eTrain* only runs at the application and framework level of Android system, and it does not have to alter the lower levels. Though both train apps and cargo apps are running at application level, the interaction between trains and cargoes is indirect. They are only able to interact through *eTrain*. Train apps trigger *heartbeat monitor* when they transmit heartbeats, and *heartbeat monitor* then informs *eTrain scheduler* the heartbeat transmission times. *eTrain scheduler* executes Algorithm 1 to determine which packets should be

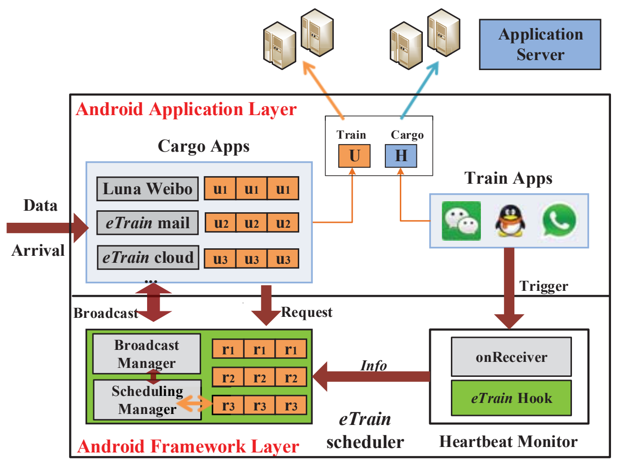


Fig. 5: System Overview of *eTrain*

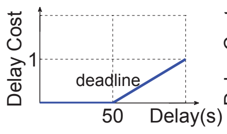
transmitted after next heartbeat and informs cargo apps this decision through *Broadcast*. *eTrain* communicates with cargo apps with Android broadcast method. Actually, there are two methods in Android for process communication, IPC (Inter-Process Communication) and Broadcast. IPC is efficient when a continuous communication has to be maintained between two processes, while broadcast is more efficient for one-to-many communications, which is the case for *eTrain*.

2) ***Heartbeat Monitor:*** The most common way train apps use to schedule periodic transmissions of heartbeats on Android is to use the system classes **AlarmManager** and **BroadcastReceiver**. The former is designed to generate a system signal at any specific time, while the latter is used to pick up the system signals and trigger the train apps to send a heartbeat. The APIs of these two classes help us to precisely locate the heartbeat sending code in the decompiled train apps.

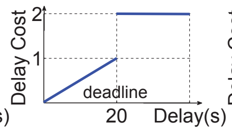
To add the trigger sending method to the end of the train apps' heartbeat sending code, we develope a **Xposed** module based on **Xposed** framework. The **Xposed** framework extends the `system/bin/app_ process` executable and allows developers to extend or modify any method in a given class without modifying the original source code. Whenever a train app sends out a heartbeat, the **Xposed** module will send a trigger to the heartbeat monitor. After heartbeat monitor receives the trigger, it will inform *eTrain Scheduler* immediately, so *eTrain Scheduler* will know the exact time of transmitting heartbeat and schedule cargo apps' data transmissions accurately.

3) ***eTrain Scheduler:*** This module implements the online transmission strategy presented in the previous section. It internally maintains one (virtual) queue for every cargo app that has subscribed to *eTrain*'s service, corresponding to Q_i in previous section's discussion. The requests submitted by a cargo app is stored in the corresponding queue in *eTrain*. Based on the meta-data information contained in the requests and the cargo app's profile, which is obtained when the cargo app registers for *eTrain*'s services, the scheduler module executes Algorithm 1 to determine when and which transmission should take place. In case when no train app is running, *eTrain* will stop its scheduler to avoid cargo apps' indefinite waiting .

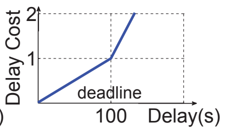
4) ***eTrain Broadcast:*** This is the interface through which *eTrain* interacts with the cargo apps. When a cargo app has data packets to send or want to download some data (mainly



(a) f_1: *eTrain* Mail



(b) f_2: Luna Weibo



(c) f_3: *eTrain* cloud

Fig. 6: Profile functions

for prefetching purpose), it submits a request to *eTrain*, which contains meta-data about the transmission, e.g., size of the data packet and its deadline for delivery, etc., through the broadcast module. *eTrain* also delivers the transmission decisions (about when and which packet should be transmitted) made by the scheduler module using the broadcast module. Then the cargo app will perform data transmission according to *eTrain*'s decision. All communications between *eTrain* and the cargo apps are currently implemented with **BroadcastReceiver** provided by Android. Developers only need to add some predefined subclasses of **BroadcastReceiver** provided by *eTrain* system, and let other logic unchanged.

*5) **Cargo apps for evaluation**:* We have developed three cargo apps based on *eTrain* for our experiments: A full-featured third party Weibo (counter part of Twitter) client Luna [21], which is the representation of SNS applications; an Email client (*eTrain* Mail), which is one of the most widely used type of mobile applications; a cloud storage application (*eTrain* Cloud), which stands for applications that need to transmit large amount of delay-tolerant data. The Weibo client Luna as an experimental product has over 100 users in different areas, including (Wuhan, ShangHai, HongKong and California). We record user behaviors, store them in our server, and use them as the controlled experiment traces. There are four elements in the trace (User ID, Behavior type, Time, Packet Size). This information is used to replay user behavior in our controlled experiments, as will be described later in the evaluation.

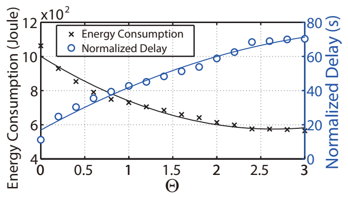
VI. Evaluation

In this section, we evaluate *eTrain* through both extensive simulations and real measurements on our implementation.

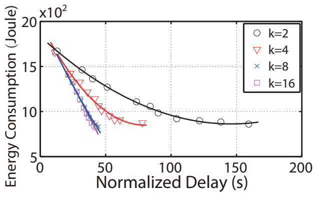
A. Simulation Setup and Methodologies

Profile functions: We choose the delay cost functions illustrated Fig. 6 for the three tested cargo apps in both experiments and simulations. The choice of the functions are inspired by [15]. For *eTrain* Mail, the cost function f_1 stays at 0 before deadline is violated, and increase linearly afterwards. In particular, $f_1(d) = d/deadline - 1$ for $d \geq deadline$, where d denotes delay. For Luna Weibo, we choose f_2 whose initial cost is proportional to d before deadline, and stays at a large constant afterwards. For the presented results, we set $f_2(d) = d/deadline$ when $d \leq deadline$ and $f_2(d) = 2$ otherwise. For *eTrain* Cloud, the delay cost is also proportional to d before deadline. But it increases much faster after deadline is violated. We set $f_3(d) = d/deadline$ for $d \leq deadline$ and $f_3(d) = 3 \times d/deadline - 2$ otherwise. The choice of delay cost profile functions is somewhat arbitrary and is for experimental purpose only. However, we do believe they are representative in various application scenarios. Finding more sophisticated delay cost models is beyond the scope of this paper.

Synthesized packet trace: According to measurement results presented in Sec.II, we synthesize heartbeats for 3 train apps (QQ, WeChat, and WhatsApp) with cycles of 300 seconds, 270 seconds and 240 seconds and heartbeat packet sizes of 378B, 74B and 66B, respectively. For the 3 *cargo* apps (Mail, Weibo and Cloud), we generate packet arrivals according to independent Poisson processes. We fixed the proportion of their mean inter-arrival times as 5:2:10, which roughly captures the different network access frequencies of the three types of applications. Besides, the size of data packets are drawn from truncated *Normal Distribution* with mean and minimum 5KB and 1KB for *eTrain* Mail, 2KB and 100B for Luna Weibo and 100KB and 10KB for *eTrain* Cloud.



(a) Impact of $\tilde{\Theta}$



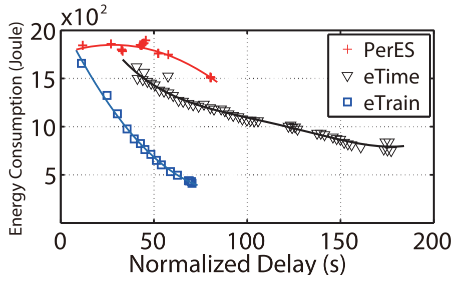
(b) Impact of k

Fig. 7: Performance of *eTrain* under different values of $\tilde{\Theta}$ and k

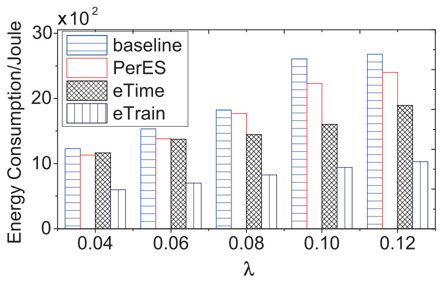
Real-world bandwidth trace: We use real-world bandwidth trace in our simulations in order to capture the intrinsic stochasticity of realistic cellular networks. During a 2-hour period (8:00AM - 10:00AM, December 8th, 2014), carrying an Android smartphone equipped with 3G interface, we took a bus to travel around downtown area of Wuhan, China, and then walked around in a university campus. During the period, a trace collecting app persistently uploaded data to the Amazon EC2 datacenter located in East Asia. The app measured and recorded the average uplink bandwidth every second.

Benchmark: We compare *eTrain* with a default baseline strategy and two recent online scheduling algorithms, *PerES* [15] and *eTime* [16]. In baseline, no energy-saving scheduling intelligence is imposed and all data is scheduled for transmission immediately after arrival. Similar to *eTrain*, *PerES* [15] and *eTime* [16] are designed under the Lyapunov optimization framework. However, both solutions heavily rely on accurate estimation of instantaneous wireless bandwidth or channel state quality, and try to time transmissions when bandwidth is high or channel is good. As discussed before, such estimations are hard to obtain with high accuracy in practice. Besides, *PerES* is deadline-aware as *eTrain* dose, while *eTime* is not. In simulation, we strictly follow the algorithm description from [15], [16]. All three scheduling algorithms only make scheduling decisions for data packets and do not interfere original heartbeat transmission. So heartbeats packets are always transmitted immediately after arrival. Moreover, multi-interface selection (e.g., WiFi, cellular) in [16] is limited to the cellular network interface only.

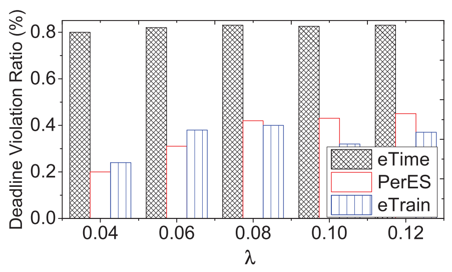
Other simulation settings: According to Fig. 4, we set the power cofficient with the screen off $\tilde{p_D} = 700mW$, $\tilde{p_F} = 450mW$, the duration of tail time $\delta_D = 10s$, and $\delta_F = 7.5s$. We run simulation of 7200 seconds (the length of bandwidth trace) for all algorithms. Since all three scheduling algorithms execute in a time-slotted fashion, we set the length of a time slot in *eTime* to be 60 seconds as suggested in [16], and 1 second for *PerES* and *eTrain*. We investigated the following three performance metrics: (1) total energy consumption, (2) normalized delay: average delay per data packet, and (3) deadline violation ratio: the percentage of packets which violate their deadlines.



(a) Performance of all algorithms in E-D panel when $\lambda = 0.08$



(b) Energy consumption of all algorithms under different λ when normalized delay is 55 seconds



(c) Deadline violation ratio of all algorithms under different λ when normalized delay is 55 seconds

Fig. 8: Performance metrics of all the algorithms under different λ

B. Parameter Analysis

We first investigate how parameters of *eTrain*'s online scheduling algorithm affect its performance. we first set the mean inter-arrival time of the three cargo apps as 50s, 20s and 100s, i.e., total arrival rate $\lambda = 0.08$ packets/second.

1) Impact of the cost bound $\tilde{\Theta}$: $\tilde{\Theta}$ is a tunable parameter in our prototype implementation that user can set at will. We first set k (a pre-defined parameter) as 20. As shown in Fig. 7(a), by changing $\tilde{\Theta}$ from 0 to 3 with a step of 0.2, the total energy consumption for 2-hour period reduces from over 1000 Joules to about 600 Joules (about 40% reduction), and the average delay increases from 18 seconds to 70 seconds. We can observe that a larger delay usually implies more energy saving.

2) Impact of k: Fig. 7(b) is the E-D panel plot we used to evaluate the performance of our algorithm, for k ranging from 2 to 16. For a given k, each data point which represents the value pair of the total energy consumption (E) and normalized delay (D) in E-D panel is achieved at a different value of $\tilde{\Theta}$. It is obvious that as k increases, *eTrain* is able to achieve the same level of energy saving with lower delay, or saves more energy at the same delay. Specifically, as k increases from 2 to 8, the energy consumption decreases by about 460 Joules when normalized delay is 40 seconds. However, when k further increases from 8 to 16, only over 30 Joules are saved. Given larger k always improves performance, we set $k = \infty$ for the following simulations.

C. Comparative Analysis

We compare *eTrain* with the selected benchmarks under different packet traces with different λ. Based on the packet trace with $\lambda = 0.08$, we generate other four packet traces with λ as 0.04, 0.06, 0.10, 0.12 packets/second respectively by scaling up or down their mean inter-arrival time from 50s, 20s and 100s ($\lambda = 0.08$) as λ, e.g., the mean inter-arrival time of the three cargo apps is 100s, 40s, 200s when $\lambda = 0.04$. For each trace, we conduct all the scheduling algorithms individually and evaluate them in E-D panel. In the E-D panel for *eTime* which does not have delay cost bound, each data point is required by tuning the control parameter V (the tradeoff parameter between energy saving and user experience in Lyapunov framework). For eTrain and *PerES* (*PerES* is designed with a dynamic V which would converge dynamically according to users' performance cost bound Ω), we just adjust cost bound to achieve different value pairs of E and D as *eTime*.

eTrain outperforms *PerES* and *eTime* under different arrival rates. Due to limitation of space, we only show the performance of all the scheduling algorithms in E-D panel with $\lambda = 0.08$ as shown in Fig 8(a). Then, we quantify the total energy consumption and deadline violation ratio of all algorithms with the same normalized delay as 55 seconds under different λ by picking the right value of Ω, V and $\tilde{\Theta}$.

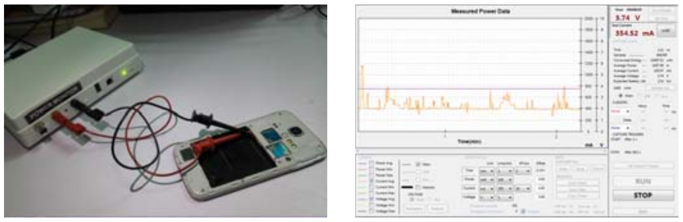


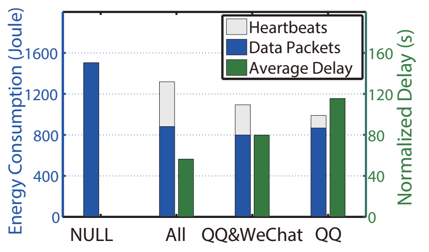
Fig. 9: Experimental setup

As shown in Fig. 8(b), as λ increases with a granularity of 0.02, the total energy consumption of baseline increases fast at first then flattens around 2600 Joules when λ gets to 0.1. Under such heavy workload, a significant number of tails of transmissions already overlap with the following data transmissions. Obviously, for energy saving, our *eTrain* performs best among them under all traces with the energy saving from 628 Joules to almost 1650 Joules compared with the baseline. Besides, *eTime* outperforms *PerES* by about 320 Joules in energy saving when λ get to 0.08. This is mainly because in *PerES*, the benefit of seizing the right time for energy-efficient transmission is counteracted under such highly fluctuant bandwidth and such heavily fragmented transmissions (about 288 arrivals per hour when $\lambda = 0.08$). While in *eTime*, the effect of aggregation is enhanced under such scenario by its coarse-grained scheduling time slot, which is set 60 seconds compared with 1 second in *PerES*. Therefore, it is more significant to aggregate fragmented transmissions to save the tail energy than to time data transmission. Compared with *eTime*, *eTrain* performs better since *eTrain* can manage the interfere of other packets which can not scheduled flexibly.

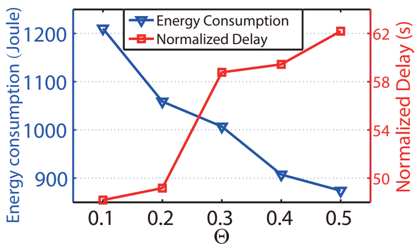
As shown in Fig. 8(c), *eTrain* and *PerES* perform better than *eTime* under different λ. Specifically, *eTime* has a high deadline violation ratio of over 0.8 while it ranges from 0.2 to 0.45 for *eTrain* and *PerES*. This is mainly because *eTime* is not deadline-aware. Besides, in *eTime*, small packets may be blocked and incur large delay since big packets have a high priority when scheduled to transmit. However, the packets in trace are mostly small, accounting for about 87.5%.

D. Controlled Experiment

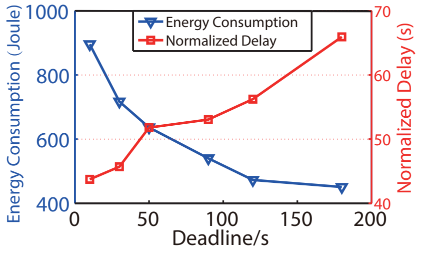
To evaluate *eTrain*'s power saving performance in real world, we performed controlled experiments by running *eTrain*



(a) Energy consumption with different train apps



(b) Energy saving characteristics of $\tilde{\Theta}$



(c) Energy saving characteristics of deadline

Fig. 10: Results from controlled experiment

and cargo apps we developed. Fig. 9 shows the experimental setup. We use power monitor [12] to provide the current with constant voltage 3.7V to the smartphone instead of using the battery. We install power tool software [22] on the laptop, and use it to capture the current of the smartphone every 0.1 second. Then, we can accurately calculate the energy consumption of the smartphone from the current trace. We implemented workload generating functionality that replays the user traces for our controlled experiment. Our experiments are conducted on state-of-the-art devices, Samsung Galaxy S4 and Google Nexus IV smartphones running Android OS.

1) Impact of train apps: We first investigate the impact of the number of train apps on the test devices. Fig. 10(a) shows the total energy consumption and average delay for 3 cargo apps' data transmissions with 0, 1, 2 and 3 train apps. The red bars show the energy consumed with only transmissions for heartbeats from the train apps for the same amount of time with no cargo app. The blue bar for NULL shows the total energy consumption with only the three cargo apps and no train app; and bars for the other three settings show the **additional** energy consumption incurred by the transmission of the three cargo apps' data packets scheduled by *eTrain*. The green bars show the average delay for data packets. Since all data packets will be transmitted upon arrival when there is no train app, the average delay is 0 for NULL.

As we can see, if only the additional energy consumption for cargo apps (the blue bars) is considered, *eTrain* can save about 45% of energy, and the amount of energy (for data packet transmissions) saved varies little when the number of train apps changes. i.e., *eTrain* is very effective even if there is very few heartbeats to piggyback on with only 1 train app. When the energy cost of heartbeats are also considered, *eTrain* can still save about 12-33% of total energy (the blue and red bars). The total energy cost increases as there are more train apps, for there will be more heartbeats. On the other hand, increasing the number of train apps significantly reduces delay, as there are more opportunities for piggybacking. With 3 train apps, the delay is reduced by 50% compared to that with only 1.

2) Impact of the cost bound $\tilde{\Theta}$: By changing $\tilde{\Theta}$, the user can enjoy a wide range of tradeoff between energy saving and delay performance, as illustrated in Fig. 10(b). By changing $\tilde{\Theta}$ from 0.1 to 0.5, the total energy consumption for the 2-hour period reduces from over 1200 Joules to about 850 Joules (about 30% reduction) with 3 cargo apps and 3 train apps running on the device, and the average delay increases from 48 seconds to 62 seconds (about 30% increase). For users that value delay performance a lot, choosing a small $\tilde{\Theta}$ will achieve better delay performance at the cost of larger energy consumption. Meanwhile, a more patient user that can accept longer delays can set a larger $\tilde{\Theta}$ for better power saving.

3) Impact of delay cost function: While $\tilde{\Theta}$ directly imposes an upper bound on the power consumption, its effect on the delay performance is indirect and quite convoluted. For users that desire more predictable delay, a better and more direct way to control the delay performance is to adjust the delay cost function. Here we examine how the $deadline$ parameter of the delay cost functions affects *eTrain*'s performance. For this experiment, all three cargo apps share the same $deadline$, which varies from 10 to 180 seconds. Fig. 10(c) shows that adapting $deadline$ can achieve a similar energy-delay that is achieved by $\tilde{\Theta}$. A larger $deadline$ allows a data packet to stay in the system longer to have more opportunities for piggybacking, hence achieve more energy saving.

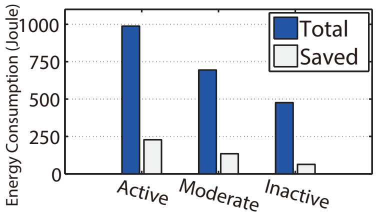


Fig. 11: Energy consumption of different user categories

4) Impact of the user activeness: Different user activeness will influence the energy saving performance of *eTrain*. We define an "app use" as a period the user opens the app and stay actively using. According to the traces we collected from our Weibo app, we divide the users into 3 categories: *active users* whose upload event records are more than 20 in each "app use" on average; *moderate users* whose upload event records are between 10 and 20 in each "app use", and *inactive users* whose upload event records are less than 10. Most users only spend 5 to 10 minutes on browsing Weibo each time. For traces longer than 10 minutes, we truncate them and use the first 10 minutes. For traces shorter than 10 minutes, we extend them to 10 minutes and insert synthetic heartbeat for the period beyond the original trace length. Other default settings are as follows: delay cost bound $\tilde{\Theta}$ = 0.2; $k = 20$ (maximum number of packets allowed to piggyback); and the deadline for Weibo is 30 seconds. We replay user traces with or without *eTrain* running on the background and record the energy consumption to figure out the energy *eTrain* saved. As shown in Fig 11, blue bars represent the total energy consumption without *eTrain*. Green bars represent the energy *eTrain* saved. For active users, *eTrain* saved 227.92 Joules (account for nearly 23.1% of energy spending on network activities) with 3 train apps running. Under the same experimental environment, *eTrain* reduce 134.47 Joules (19.4%) for moderate users and 63.23 Joules (13.3%) for inactive users. Active users generate more uploading events, which means more cargoes for *eTrain* to piggyback. Hence, *eTrain* saves more energy for active users.

VII. Related Work

Energy consumption of networking activity in mobile phones has seen a large body of works recently. Measurements [4], [5], [11], [23] reveal the serious impact of cellular tail time on energy consumption in popular mobile apps. However, to the best of our knowledge, there has been no universal software platform like *eTrain* that application developers can utilize with ease to reduce tail energy wastage. Instead, developers need to carefully restructure each application to schedule and aggregate I/O requests [5], [24]. Besides, compared to other cloud computing based works in [25] and AppATP [26] which have to build a extra middleware in the cloud, *eTrain* is a more lightweight system which only works in mobile end.

Another popular technique to reduce tail energy is named fast dormancy, which puts the radio interface into IDLE state immediately after a data transmission to shorten or eliminate the tail [4], [5], [11], [27]. However, Changing the tail mechanism may lead to frequent radio interface state transitions and significantly increase the chance for a data transmission to experience additional state promotion delay [28]. In contrast, while keeping the tail mechanism, *eTrain* saves energy by turning heartbeat tails into useful data transmission opportunity.

Existing works on scheduling delay-tolerant or perfetch-friendly data transmissions usually utilize the time when mobile devices are in high power state such as phone calls or user actively using other applications. PCS [29] exploits smartphone app opportunities to piggyback mobile sensor data. However, its success requires accurately predicting user activities, which is difficult in reality. On the other hand, *eTrain* makes much better scheduling decisions because it uses heartbeat activities which can be easily predicted with high accuracy.

VIII. Conclusion

Through measuring the characteristics of heartbeats of popular smartphone apps, this paper first quantitatively reveals not only their incurred cumulative tail energy consumption, but opportunities to turn such energy wastage into more useful piggyback of delay-tolerant data. This practical insight further inspires us to design and implement *eTrain*, an energy-efficient transmission management system running on Android smartphones, which schedules aggregated delay-tolerant data of multiple apps to piggyback along with heartbeats, so as to significantly reduce the tail energy wastage in fragmented data or heartbeat transmissions. Extensive trace-driven simulations and real-world experiments demonstrate that each component of *eTrain* can cooperate seamlessly with each other, and achieve notable energy saving in various scenarios.

References

[1] Mobile Users Worldwide. [Online]. Available: http://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/

[2] WeChat. [Online]. Available: https://play.google.com/store/apps/details?id=com.tencent.mm

[3] WhatsApp. [Online]. Available: https://play.google.com/store/apps/details?id=com.whatsapp

[4] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Characterizing radio resource allocation for 3g networks," in *ACM IMC*, 2010.

[5] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *ACM IMC*, 2009.

[6] Mobile QQ. [Online]. Available: https://play.google.com/store/apps/details?id=com.tencent.mobileqq

[7] Y. Liu and L. Guo, "An empirical study of video messaging services on smartphones," in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*. ACM, 2014, p. 79.

[8] Android Developers. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html

[9] Netease News. [Online]. Available: https://play.google.com/store/apps/details?id=com.netease.newsreader.activity

[10] RenRen. [Online]. Available: https://play.google.com/store/apps/details?id=com.renren.xiaonei.android

[11] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li, "Optimizing background email sync on smartphones," in *ACM MobiSys*, 2013.

[12] Monsoon Solutions Inc. Power Monitor. [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/.

[13] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *IEEE/ACM/IFIP CODES+ISSS*, 2010.

[14] M. J. Neely, "Stochastic network optimization with application to communication and queueing systems," *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1–211, 2010.

[15] Y. Cui, S. Xiao, X. Wang, M. Li, H. Wang, and Z. Lai, "Performance-aware energy optimization on mobile devices in cellular network," in *IEEE INFOCOM*, 2014.

[16] P. Shu, F. Liu, H. Jin, M. Chen, F. Wen, Y. Qu, and B. Li, "etime: energy-efficient transmission between cloud and mobile devices," in *IEEE INFOCOM*, 2013.

[17] Z. Zhou, F. Liu, H. Jin, B. Li, B. Li, and H. Jiang, "On arbitrating the power-performance tradeoff in saas clouds," in *IEEEINFOCOM*, 2013.

[18] Y. Wang, X. Liu, A. Nicoara, T.-A. Lin, and C.-H. Hsu, "Smarttransfer: transferring your mobile multimedia contents at the right time," in *ACM NOSSDAV*, 2012.

[19] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan, "Bartendr: a practical approach to energy-aware cellular data scheduling," in *ACM MobiCom*, 2010.

[20] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely, "Energy-delay tradeoffs in smartphone applications," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 255–270.

[21] Luna Weibo. [Online]. Available: https://play.google.com/store/apps/details?id=com.datacenter.luna&hl=zh_CN

[22] Power tool. [Online]. Available: http://msoon.github.io/powermonitor/.

[23] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Periodic transfers in mobile applications: network-wide origin, impact, and optimization," in *ACM Proceedings of the 21st international conference on World Wide Web*, 2012.

[24] H. Liu, Y. Zhang, and Y. Zhou, "Tailtheft: leveraging the wasted time for saving energy in cellular communications," in *ACM MobiArch*, 2011.

[25] F. Liu, P. Shu, H. Jin, L. Ding, J. Yu, D. Niu, and B. Li, "Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications," *Wireless Communications, IEEE*, vol. 20, no. 3, pp. 14–22, 2013.

[26] F. Liu, P. Shu, and J. Lui, "Appatp: An energy conserving adaptive mobile-cloud transmission protocol," *Transactions on Computers*, 2015.

[27] B. Zhao, Q. Zheng, G. Cao, and S. Addepalli, "Energy-aware web browsing in 3g based smartphones," in *IEEE ICDCS*, 2013.

[28] W. Hu and G. Cao, "Energy optimization through traffic aggregation in wireless networks," in *IEEE INFOCOM*, 2014.

[29] N. D. Lane, Y. Chon, L. Zhou, Y. Zhang, F. Li, D. Kim, G. Ding, F. Zhao, and H. Cha, "Piggyback crowdsensing (pcs): energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities," in *ACM SenSys*, 2013.