Dynamic Adaptive Techniques for Learning Application Delay Tolerance for Mobile Data Offloading

Ozlem Bilgir Yetim and Margaret Martonosi
Princeton University
Email:{obilgir, mrm}@princeton.edu

***Abstract*—Today's worldwide mobile data traffic is roughly 18× larger than the full internet traffic in 2000, and continued large growth is expected. High mobile data usage has implications both for users and providers. For individual users, relying on cellular data connectivity incurs high cellular data fees. For cellular network providers, high mobile data usage requires expensive, ongoing infrastructure upgrades. Cellular data usage can be reduced by offloading to WiFi when available. If not available, prior work has considered delaying transmissions to wait for WiFi availability. While exploiting such *application delay tolerance* offers significant energy and performance leverage for data offloading and other techniques, a key question is: how long to wait? Prior work does not discuss how to estimate application delay tolerance without explicit help from programmers, nor how to adjust the estimate dynamically.**

This work proposes, implements, and evaluates four schemes to dynamically and adaptively deduce an application's delay tolerance. These schemes (*Adaptive*, *Decision Tree-Based*, *Hybrid* and *Lazy*) are low-overhead and effective. In our experiments, they cut cellular usage by 2× or more compared to non-delay-tolerant approaches. Furthermore, our dynamically adaptive decision schemes achieve up to 15% further cellular data reduction compared to fixed static delay tolerance values.

I. Introduction

Mobile data usage has been drastically increasing worldwide. According to the Cisco Visual Networking Index [9], global mobile data traffic grew by 81% in 2013 alone, and by 2018, it is projected to grow 12× and reach 15 exabytes per month. This heavy cellular network bandwidth usage forces cellular network providers to frequently upgrade their infrastructure to meet the demand. In addition to the burden on cellular network providers, user phone bills increase with high cellular data usage [2].

Fortunately, mobile devices are also equipped with WiFi connectivity as an alternative to cellular. Offloading some cellular traffic to WiFi is attractive to both users and network operators. For users, such offloads can decrease their monthly data usage bill or can improve smartphone battery life, since per-bit energy consumption of WiFi is one order of magnitude less than 3G [10]. For network operators, data offloading to WiFi reduces congestion in cellular networks and results in better network capacity management [18].

Exploiting application delay tolerance opens up additional opportunities for WiFi offloading. Delay tolerance refers to the amount of delay a mobile application or its user can tolerate in exchange for some benefit in bytes usage or energy consumption. Many applications have some amount of delay tolerance, and using this time window to optimize connectivity choices can greatly reduce reliance on cellular data connectivity. However, relying on application developers to select a delay tolerance value may not be acceptable to all users of the application.

Previous researchers recognized that applications have delay tolerance as well as studied mobile data offloading to WiFi networks during this time window [3, 5, 6, 12]. The focus in this prior work was on *using* fixed delay tolerance values to guide optimal or heuristic policies for connectivity choice, rather than on methods for how to estimate it dynamically. No previous work has discussed methods for estimating delay tolerance values or how to dynamically adjust them.

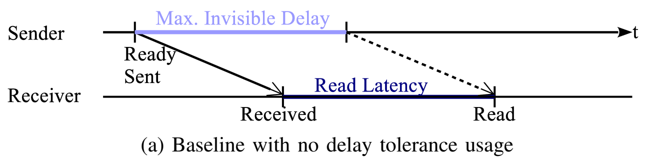
This work proposes dynamically, adaptively and automatically deducing and exploiting per-user application delay tolerance for each data transfer by using current and past request patterns for the application data. Our goal is to maximally utilize delay tolerance while minimizing degradation of user experiences (both for senders and receivers of the data). Thus, we propose heuristic and statistical delay tolerance decision schemes, which dynamically predict when to delay the data item more (if it is not going to be accessed immediately) and less (if other users want to retrieve the data). We show how time information regarding data access demand can be inferred using small amounts of metadata that is sent before transmitting the data item.

This paper's primary case study uses an email application, but our framework is general and gives insight on how to apply the proposed techniques to other applications as well. We evaluate our delay tolerance estimators using email usage traces. Our decision schemes, *Adaptive*, *Decision Tree-based*, *Hybrid* and *Lazy*, take the receiver's email reading request into account while making a delay tolerance decision. We show how offloading based on these delay tolerance estimates reduces sender cellular bytes usage and receiver user experience compared to a fixed delay tolerance decision scheme.

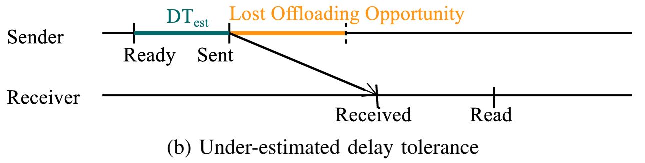
The primary contributions of our work are as follows. First, using application delay tolerance and postponing network transmissions to the future has significant leverage for cellular data savings, but must be applied intelligently. Assuming simple, fixed delay tolerance can have negative impact on the users waiting for that data transmission. We propose dynamic delay tolerance estimators that offer better trade-offs between cellular bytes saving and user impact.

Second, even within a single application, per-user adaptation of delay tolerance estimates is important. By using real email traces, we quantify how a fixed, identical delay tolerance can have varying impact depending on user data access behavior. Our trace-based simulations show that fast email readers can be affected 7× more than the slow email readers.

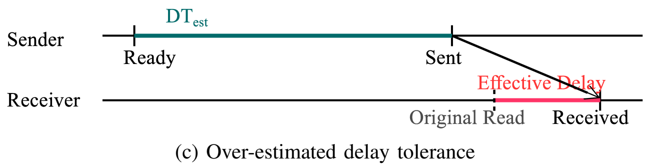
Material based upon work supported by NSF Grant CNS-1135953.



(a) Baseline with no delay tolerance usage



(b) Under-estimated delay tolerance



(c) Over-estimated delay tolerance

Fig. 1: Toy example timeline of an email delivery. For simplicity, we exclude the email servers in the middle.

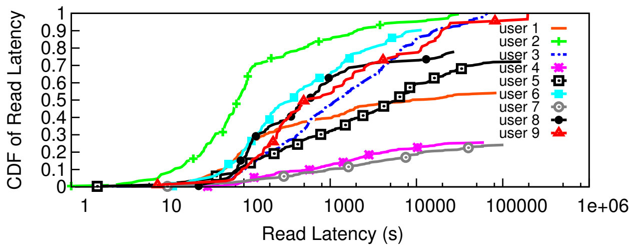


Fig. 2: CDF of email read latency for different users.

Third, dynamic delay tolerance estimation substantially decreases the delay impact on the users. Compared to a fixed policy, the *Adaptive* scheme which adapts based on email-reading delay patterns achieves the same cellular bytes usage with considerably less (80s) effective delay. In addition to email reading delay patterns, the email receive time and email size can be used in delay tolerance decisions. This offers a better Pareto curve. Our *Decision tree-based* scheme achieves the same cellular bytes usage with around 200s less effective delay. Moreover, the *Lazy* scheme tries to maximize the cellular data savings with the minimal impact on the receivers.

II. MOTIVATION: DYNAMIC DELAY TOLERANCE

As previous work suggested [3, 5, 6, 12], delaying transmissions until some delay tolerance deadline in order to offload them to WiFi can save cellular data usage. For intermittent WiFi, a 2 hour delay tolerance can result in 70% cellular data savings for large data transfer [12]. Such large delay tolerances may, however, degrade user experiences. Our goal is to automatically deduce and exploit delay tolerances that are typically much smaller, and that are tailored to individual users, applications, and usage scenarios.

A. Deducing Delay Tolerance

This paper discusses options in the context of an email application since it is commonly used [11] and intuitively quite delay tolerant. However, our framework also provides broader opportunities for other applications. Increasing use of dropbox, photo-gallery and other such mobile applications point to a broad set of workloads that can benefit from our dynamic delay tolerance decision techniques for better offloading.

To offer a simple motivating example and to define terminology, Fig. 1 shows a timeline of an email delivery without and with delay tolerance exploitation. Fig. 1a shows a case where the email is sent right away after becoming ready. Email delay tolerance is not being utilized here; if WiFi is not immediately available, the data will be sent via cellular. We define the difference between the read time and the receive time of the email at the receiver as *read latency*. This is the maximum invisible delay tolerance this email has without affecting the receiver's email read behavior.

In Fig. 1b, transmission is delayed until some delay tolerance estimation (DT_{est}) window with the hope to use WiFi for delivery. However, since DT_{est} is smaller than the true maximum delay tolerance for this case, some offloading opportunity is lost. If DT_{est} had been bigger, the probability of the sender finding WiFi connectivity would be greater.

In Fig. 1c, delay tolerance is over-estimated. As a result, the email is not read by the receiver at the original read time because it was not received yet. In this case, using the over-estimated delay tolerance introduces some delay, which we call *effective delay* on the receiver's read request. On the other hand, over-estimation can increase the offloading efficiency for the sender since the window to wait for WiFi is longer.

To summarize, there is a clear trade-off between cellular bytes saving and the impact on the users waiting for the data. Our goal is to better manage this trade-off by deducing delay tolerance of the application data and conservatively selecting DT_{est} for low effective delay.

B. User Study: Email Reading Behavior

There are many mobile applications where the data transmissions can benefit from our delay tolerance decision schemes. These include email, as well as widely-used cloud-based file storage and file transfer applications. This paper uses email as a the primary example, and here we discuss a user study whose data drives our experiments.

User objectives and network conditions affect the degree of application delay tolerance. For example, certain users read emails faster than others. For such users, postponing the network transmissions (to offload them to WiFi) can degrade their experience at the receiving end, even though it can results in better cellular data usage for the sender.

To understand the email reading behavior among different users, we collected the time stamps of email send, receive and read from 9 users for 10-15 days. Section IV explains our methodology in detail. Here, we will discuss the email reading latency as motivation for our per-user and adaptive strategies. Our main observations and their implications are;

Email reading latency varies highly among users, as expected. Thus, DT_{est} must be customized for each user. Some users tend to check emails very frequently, while other users read their emails very infrequently. Fig. 2 shows the CDF of email read latency for different users. The CDF curves vary considerable between users. For example, user 1 reads more than 50% of the email in less than 100s whereas user 9 reads less than 5% of the emails with the same latency.

In addition, there is a high variance of email read latency among different emails for the same user. Thus, DT_{est} might be

Parameter	Description
Acc_{body}	Data body access
D_{body}	Data body
D_{meta}	Metadata
D_{item}	Data item; D_{body} + D_{meta}
D_{head}	First item in Q
$DGraph$	Decision graph
H	History of previous observations
I	Increment value to increase DT_{est}
Q	Local queue, sorted by the deadlines
Q_{meta}	Metadata queue, not shown to outside world
R_{body}	Request from receiver to get D_{body}
$R_{DT_{est}}$	Request to receiver to learn DT_{est}
$scale$	Scaling factor for $DTree$
T_{data}	Data transmission request
WS_H	Window size of H

TABLE I: Common parameters used in the algorithms.

```
// Library call for application data send
Procedure send_data($T_{data}$):
    set_deadline($DT_{est}$);
    store $D_{item}$ in $Q$ wrt. its deadline;
end
// Offloading decision
Procedure make_transfer():
    while $True$ do
        if $Q$ not empty and WiFi available then
            send $D_{item}$ from $Q$ using WiFi;
        end
    end
end
Procedure meet_deadline():
    while $True$ do
        if the deadline of $D_{head}$ expires then
            send $D_{item}$ using available network;
        end
    end
end
```

Algorithm 1: $Fixed$ offloading algorithm. DT_{est} is same for all data items. Procedure set_deadline(DT_{est}) sets the deadline of the data items to the end of DT_{est} window.

customized for each individual email. For example, for some users, the CDF does not reach 1 because these users never read some of their emails (mostly mailing list emails). With very high read latency, these emails are very delay-tolerant. Furthermore, user 3 does not read 30% of his/her emails but he/she reads another 30% in under 200s. This also shows the high variability of delay tolerance of different emails for a single user.

These observations suggest that per-user, dynamic and adaptive delay tolerance decision is necessary.

III. Delay Tolerance Estimation Approaches

We propose and evaluate schemes for deducing and deciding delay tolerance for individual data transmissions: *Adaptive (Adapt)*, *Decision Tree-based (DTree)*, *Hybrid* and *Lazy*. The schemes then use this delay tolerance estimate to make offloading decisions. The schemes vary in approach, but all prioritize user experience over data savings. In addition, we also compare against *Greedy* and *Fixed decision (Fixed)* schemes, with the goal of having more cellular data savings than *Greedy* and better user experience for a given savings than *Fixed.*

```
// Library call for application data send
Procedure send_data($T_{data}$):
    if WiFi is available then
        send $D_{item}$;
    else
        send $D_{meta}$;
        send $R_{DT_{est}}$;
        store $D_{body}$ in $Q$;
    end
end
// Offloading decision
Procedure make_transfer():
    while $True$ do
        if $Q$ not empty and WiFi available then
            send $D_{body}$ from $Q$ using WiFi;
        end
    end
end
Procedure meet_deadline():
    while $True$ do
        if the deadline of $D_{head}$ expires then
            send $D_{body}$ using available network;
        end
    end
end
Procedure set_deadline($DT_{est}$):
    set the deadline for $D_{body}$ at the end of $DT_{est}$ window;
end
```

Algorithm 2: *Adapt* and *DTree* decision scheme on the sender device. *Adapt* uses Alg. 3 and *DTree* uses Alg. 4 to decide DT_{est} for each data item individually.

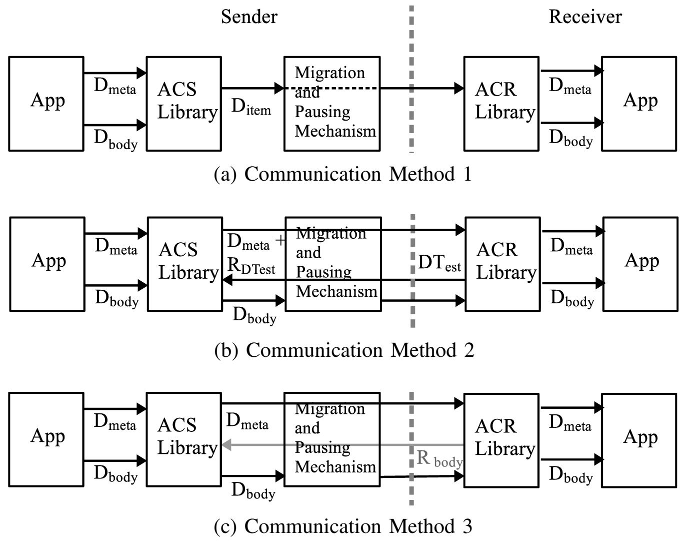


Fig. 3: Communication methods between the sender and the receiver required by the different decision methods. $Greedy$ and $Fixed$ use (a), $Adapt$, $DTree$ and $Hybrid$ use (b), and $Lazy$ uses (c).

A. System Overview

Fig. 3 shows the overall system diagram and the different communication methods. In this implementation, we introduce libraries both on the sender (application communicator in sender, ACS) and the receiver (application communicator in receiver, ACR). Different offloading schemes require different communication support from these libraries.

ACS and ACR are the main libraries with which applications communicate for their data transmissions. Applications

```
Procedure build_history():
    while true do
        if $D_{meta}$ is received then
            store it in $Q_{meta}$ ;
        end
        if $Acc_{body}$ then
            forall the $D_{meta}$ in $Q_{meta}$ do
                mark $D_{meta}$ late;
                update_DT(True);
            end
        end
        if $D_{body}$ is received then
            if its $D_{meta}$ is not marked as late then
                mark $D_{meta}$ early;
                update_DT(False);
            end
        end
        if $R_{DT_{est}}$ is received then
            send $DT_{est}$ to the sender;
        end
    end
end
Procedure update_DT(bool isBodyLate):
    append $isBodyLate$ to $H$;
    $median \leftarrow$ median of last $WS_H$ items in $H$;
    if $median == True$ then
        $DT_{est} \leftarrow DT_{est}/2$;
    else
        $DT_{est} \leftarrow DT_{est} + I$;
    end
end
```

Algorithm 3: Adaptive decision scheme on the receiver side, which continuously updates its DT_{est}

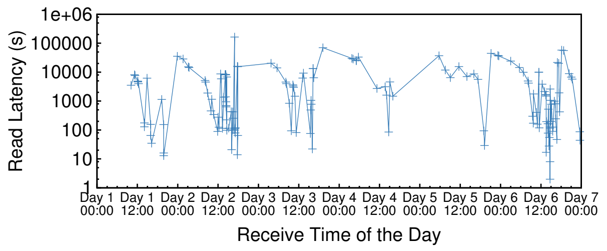


Fig. 4: Read latency of all emails received in the first 6 days for user 7. Relatively predictable variations in read latency motivate *Adapt* and *DTree* approaches.

do not make offloading decisions or use the delay tolerance directly. The only functionality required is to identify metadata and the data body and to use send/receive functions to forward them to these libraries. Both ACS and ACR are responsible for handling the required communication. ACS is also responsible for making the offloading decision.

ACS uses the migration and pausing modules proposed in [6]. The migration module uses Serval [15] which provides the necessary functionality to migrate flows seamlessly among different network interfaces without application support. Serval introduces a new layer to the network stack on top the of network layer. Instead of necessitating IP address and port number for network transmissions, it uses service IDs and flow IDs to identify services and the transmissions, respectively. In addition, this architecture also provides functionality to connect to the services and hosts without keeping track of their network address in the mobile environments.

The pausing module is responsible for delaying data transfers by pausing TCP window management and resuming later.

```
Procedure set_deadline($D_{item}$):
    $DT_{est} \leftarrow c$ x (look up from the $DGraph$);
    return $DT_{est}$;
end
```

Algorithm 4: Deciding DT_{est} by *DTree* on the sender.

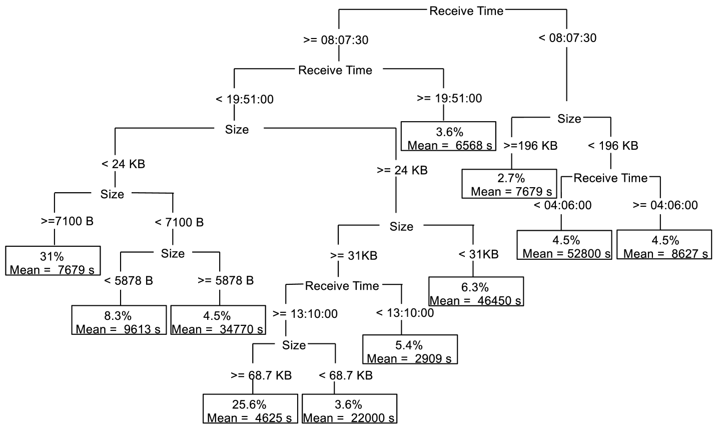


Fig. 5: Example decision tree composed by using the email receive times and size for user 7. Each leaf node shows the percentage of emails whose receive times and size follow the rules given at the parent nodes, and the resulting mean read latency.

For example, if ACS makes an offloading decision for the data item and some delay is necessary until the next WiFi region, the pausing mechanism is used to freeze the TCP window management. Thus, the migration and pausing modules together provide required seamless offloading mechanism.

ACS and ACR have three main communication methods, as seen from Fig. 3. Decision schemes use one of these communication methods depending on their implementation.

B. Greedy and Fixed Decision Schemes

The *Greedy* scheme is already supported on current phones, and is our simplest baseline. Application delay tolerance is not utilized in this scheme. If WiFi is available at the time of the transmission, then it is used. Otherwise, cellular connectivity is used for the transmission.

The *Fixed* decision scheme uses a constant DT_{est} that is the same for all of the data of the same application and for all users. As shown in Alg. 1, the algorithm works similar to the previously proposed *chunking heuristic* [6] offloading scheme, but does not use WiFi prediction. Table I summarizes terminology for this and other schemes. In *Fixed*, the data items, D_{item}, are stored in a pending queue, Q, sorted by predetermined fixed deadline. Whenever WiFi becomes available, *Fixed* sends data items from Q until the connection is lost and it continues the transfer at the next WiFi availability. If a data item's deadline is going to expire, it is sent immediately using available cellular connectivity.

Implementation: The Greedy and Fixed schemes require common communication support, where ACS and ACR send/receive data body and the metadata together, as shown in Fig. 3a. This is the traditional communication method used by applications. The other schemes send the body and metadata together only if WiFi is available at the time of the original send request from the application. Moreover, *Greedy* does not

```
// Library call for application data send
Procedure send_data($T_{data}$):
    if WiFi is available then
        send $D_{item}$;
    else
        send $D_{meta}$;
        store $D_{body}$ in $Q$;
    end
end
// Offloading decision
Procedure make_transfer():
    while $True$ do
        if $Q$ not empty and WiFi available then
            send $D_{body}$ from $Q$ using WiFi;
        end
    end
end
Procedure respond_request($R_{body}$):
    send $D_{body}$ using available;
end
```

Algorithm 5: *Lazy* offloading scheme on the sender device. The receiver sends R_{body} back if the user wants to access the data body of the D_{meta}.

need the migration and pausing mechanism since it does not delay transfers.

C. Adaptive Decision Scheme

Increasing in sophistication level, it is natural to consider dynamic adaptive schemes for deciding on DT_{est}. Fig. 4 motivates our *Adapt* scheme by showing user 7's read latency for each data item over the first 6 days of logging. Each point in the graph represents an email. From one email to the next, transitions in read latency are relatively smooth and predictable. Other users show similar data, and hence this supports the intuition that reading patterns can help estimate DT_{est}. *Adapt* updates its current DT_{est} continuously using the observations on dela from previously received data.

In order to obtain timing information regarding the data requests without sending the actual data item, we send small hints or metadata about the data item to the receiver before transmitting the data item [14, 24]. For the email application we focus on, the metadata contains the email header information [21], including the subject line, delivery timestamps, sender information, and total email size (the including attachments). In addition to email, other applications, like file transfer or online file storage, can also have separate metadata files which give information about the file content, but are much smaller in size compared to the whole file. In our system, metadata can be sent separately and some applications may choose to display it to the user. We think of each data item (i.e. email) as having two parts with different delay tolerance: the *delay intolerant* metadata and *delay tolerant* data body. By identifying the metadata to be delay intolerant and initiating its transmission immediately, information about the access request behavior can be caught or deduced at the recipient.

As Alg. 2 shows, when there is a new data transmission request, in addition to the metadata, each sender sends a $R_{DT_{est}}$ to the receiver to learn the DT_{est}. After learning the delay tolerance, the sender sets the deadline for the data body. Before the deadline, data bodies are only sent if WiFi becomes available. If the deadline is expiring, the data body is sent through the available network immediately.

Alg. 3 shows how DT_{est} is estimated. If the previous data items waited for some time without any accesses (e.g. emails are not read), then the receiver decides to increase DT_{est}. If it is suspected that the data would be accessed earlier, then this scheme decides that DT_{est} was too long, so it decreases it. When a metadata is received without a body, it is enqueued and hidden from the user. (The *Lazy* scheme below will handle this differently.) The reason for concealing the metadata is that it is present only as a hint to catch the user's access intention for this data item so that the delay tolerance decision can be updated. The supposition in *Adapt* is that a user will not be troubled by a few minutes of email delay if the user interface does not explicitly expose it to them.

The receiver makes a decision of whether a data body arrived early or late for each data item in two different situations. First, if one data item is accessed (ACC_{body}), then *Adapt* assumes any other email with metadata in the queue might have been accessed as well if received. Thus, this schemes decides that if DT_{est} was shorter, the user might have received and accessed them as well. Thus, *Adapt* calls its updating procedure to reduce DT_{est}. In the other case, when a D_{body} is received, if the metadata of this item is not marked as late previously, then this scheme thinks that DT_{est} is too short, and updates accordingly.

The DT_{est} updating procedure keeps a history (of size WS_H) of previous observations on the DT_{est} being short or long. Considering the median in the history, DT_{est} adjustments follow a *linear increase/multiplicative decrease* mechanism similar to TCP congestion control [17]. If the median says that DT_{est} was short, then DT_{est} is increased by I value. If the median says that DT_{est} was long, DT_{est} is decreased to half.

Implementation: The *Adapt*, *DTree*, and *Hybrid* methods use the approach shown in Fig. 3b. These schemes need to distinguish the metadata and body. Moreover, it is necessary to match each data body with its metadata. For this purpose, a 64-bit data ID is assigned to each data item. The data ID is appended to the beginning of the payload of metadata and the data body. Thus, both metadata and the body have the same ID. The data ID together with the Serval service ID (256 bits) is used to match the data body with its metadata.

In these schemes, ACS sends only the metadata at the time of the original transmission request if WiFi is not available. In addition, ACS sends a request to learn DT_{est} for the data item. ACS stores data bodies in the local queue until WiFi becomes available and sets their transfer deadlines once DT_{est} information is received. ACS makes the offloading decision for the data body for the earliest deadline.

On the receiver side, ACR is responsible for storing the metadata locally until their bodies arrive. When a data body is received, the data ID and the service ID are used to match it with its metadata. ACR keeps track of the past observations and the resulting read latency to decide on DT_{est}.

D. Decision Tree-based Scheme

DTree uses Alg. 2 for offloading decision, similar to *Adapt*. However, this scheme uses a decision tree approach—common in machine learning [7]—to decide on the DT_{est} for each individual email using the past observations. We use this method to map the information about a data item to previous observations on the email size and receive time to correlate with the read latency. Fig. 4 also motivates some of the parameters used in *DTree*. As can be seen, email reading behavior of the same user is similar every day. Each day, around midnight, read latency increases and in the afternoon, it decreases. This behavior can also help estimate delay tolerance based on the trends on the previous days. Thus, *DTree* uses the email receive time of the day as one of the parameters for constructing its decision rules. In addition, total email size is also used as a parameter.

Fig. 5 gives an example decision tree obtained by using the receive times and email size of 3 days of observations for user 7. This shows the classification of these observations, ratio of emails in each class and the mean read latency. The tree acts as a set of rules to find which class the new data items fall into. The receiver multiplies this value by a constant $scale$ to estimate DT_{est} for a given email size and/or receive time. The purpose of using $scale$ is to able to have an impact on the decision and adjust how conservative the delay will be.

Implementation: *DTree* uses the implementation flow from Fig. 3b, but with additional metadata for training.

E. Hybrid Scheme

Because we wish to prioritize the user experience, we also implemented a hybrid scheme is more conservative than both the *Adapt* and *DTree*. For *Hybrid*, it calculates DT_{est} using both the *Adapt* and *DTree* methods, and then chooses the smaller of the two values. Naturally it follows that its implementation uses the flow in Fig. 3b.

F. Lazy Scheme

The core idea of *Lazy*, as the name suggests, is being lazy in the transmissions. Its goal is to utilize the maximum delay tolerance by deferring transmission of the data body until data is actually requested. An explicit DT_{est} is not kept in this scheme. Instead, data is assumed to be tolerant to delays until requested. Thus, in this scheme, data is only transmitted if (1) there is a WiFi connection, or (2) data is requested.

Alg. 5 explains how *Lazy* works. If there is no WiFi availability at the time of the data transmission, *Lazy* only sends the metadata and the body is stored in the local Q. Whenever WiFi becomes available at the sender, the data bodies are opportunistically sent from Q until the connection is lost. In the meantime, the metadata will be received at the destination. Unlike *Adapt*, here in *Lazy*, the metadata are actually exposed to the user, for example as email headers. If the user tries to access the data item before the body arrives, an R_{body} request is triggered from the receiver to the original sender to ask for the rest of the data item. When the sender receives this request, the data body is sent immediately.

Implementation: *Lazy* uses the communication method shown in Fig. 3c. Here, unlike Fig. 3b, ACR does not queue the metadata until their bodies arrive. Instead, metadata is given to the application as soon as it is received. At this point, if the user wants to access the data item (e.g. wants to read the email after they see the header), ACR triggers a request to ACS to fetch the body. Until the data body returns, users cannot access it and the delay due to the lazy-fetch is fully exposed to them. Thus, unlike other methods, in this scheme, users have an opportunity to access the email bodies whenever they want. On the other hand, they are explicitly aware of the delay until the body is fully received. Since our schemes rely on Serval for host discovery, the approach is robust to changes due to the sender mobility. ACR can easily communicate back to ACS when data body is requested, and ACS can return the matching data body to the requesting ACR.

IV. Methodology

We perform our evaluations using trace-based simulations, comparing the total cellular bytes usage, the average effective delay per email and the relative network energy consumption. The simulator, written in C++, uses gathered email usage traces to mimic data send and receives between different users at given times. We use email traces in our simulations, but other application traces could also be used.

A. Email Trace Collection

We collected our email traces from 9 gmail users for a time frame between 10-15 days. For each user, we collected the email size, send time (i.e. the time when the email originated from the sender), receive time (i.e. the time when the email reached receiver) and the read time (i.e. the time when the email is read at the receiver) for all the emails received to the gmail *All Mail* mailbox in our data collection period.

We use JavaMail 1.5.1 API to access the user mail boxes and collect traces [16, 25]. This API provides libraries to connect to an email account and access information about the emails received to an email folder. Gmail has an option to set filters to store emails to specific folders. However, by collecting traces for *All Mail* folder in gmail, we can access all of the emails received to all mailboxes (except the spam folder). For privacy reasons, we did not access sender information. Instead we assumed that each email originates from a different sender, but our schemes are general and do not rely on this.

Email send, receive time and size can easily be accessed using the methods provided by the JavaMail API. However, read time is not normally displayed at the headers and cannot be accessed using the JavaMail API. Instead, we use email read/unread status, which is provided. Thus, in order to collect email read time, we check the email read status every 2 mins and log the time when the status changes from unread to read.

Email read latency is the difference between read time and the receive time. Some users did not read certain emails in our data collection period. For those emails, we cap the delay tolerance at 10 days delay. To prevent miscalculation of the read latency for emails received on the last day of the data collection period, we take out the emails that are received in the last 24 hours but not read. (We picked the 24 hour threshold since for all of the users, most emails have $<$ 24 hour read latency if they were ever read at all.)

Transmission time is calculated as the time difference between the receive and the send time. However, queuing and processing delays at the email servers are also included in this time. Occasionally due to slow or overloaded email servers, they can dominate the transmission time. Thus, to be conservative and avoid overstating user delay tolerance, we exclude all emails whose calculated transmission time is more than 10 minutes, about 10% of emails per person.

B. Scheme Parameters

For DTree, we use first 3 days of emails as the training set. Thus, for all of our schemes, we only use the rest of the trace as test trace. We use the rpart package from R as our statistical tool to generate the regression trees [13]. In addition, we vary the parameter $scale$ from 1/256 to 64 to cover a wide range of decisions. For Adapt, in order to see the effect of window size and the increment value, we vary the WS_H from 1 to 32 and I from 1 to 32 mins. Previous work studied delay tolerance values ranging from 1 min to 12 hours [3, 12]. In

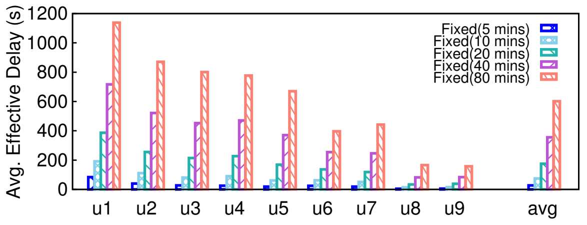


Fig. 6: If the delay tolerance is high, data offloading results in an increased effective delay for the email readers.

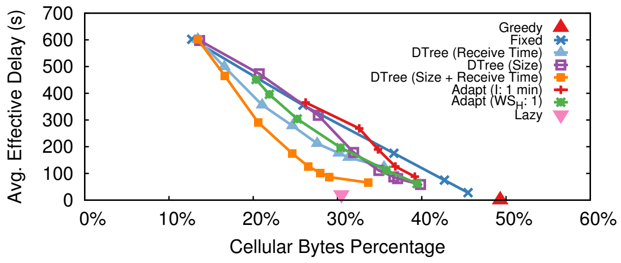


Fig. 7: Average effective delay - cellular bytes usage tradeoff where schemes use an upper bound in their decision. Dynamic schemes almost always better than fixed decision scheme and *DTree* forms the Pareto frontier among all schemes.

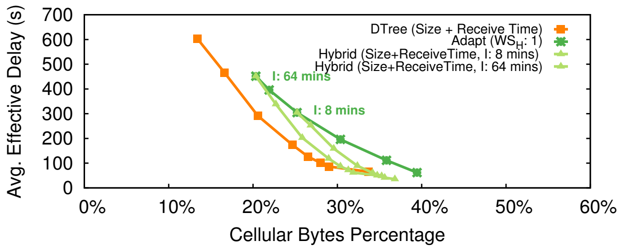


Fig. 8: The *Hybrid* scheme is more conservative than $Adapt$ and $DTree$ since it chooses the minimum DT_{est}.

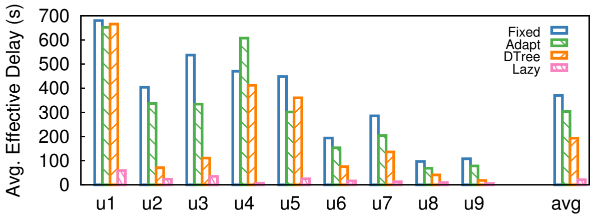


Fig. 9: Detailed analysis of effective delay for all the users with different schemes. Apart from *Lazy*, scheme parameters are chosen to achieve 25% cellular data usage.

this paper, we vary the fixed delay tolerance values from 5 to 80 minutes. We assume that extra transmissions in Adapt, DTree and Lazy for requesting/responding DT_{est} information are small transmissions sized 1KB.

All schemes cap DT_{est} to an upper bound, beyond which it never goes. The reason for the upper bound is to cap worst-case effective delay at receivers. We use 80 mins (i.e. the maximum delay used in *Fixed*) as an upper bound for all of the schemes.

C. Network Availability

We assume that 3G is always available during our simulations but this is not a requirement. In order to mimic different WiFi duration and inter-connection times, we use exponential distributions with different average values. Previous work showed that the average WiFi duration and inter-connection times can be a few seconds to a few hours [8, 12]. Thus, in order to cover a different range of WiFi scenarios, we vary the average duration and inter-connection times from 450s to 2 hours. We repeat each experiments 10 times since the WiFi behavior is generated from a random distribution.

D. Energy Calculation

We also compare the network energy usage of different schemes. Our focus is the transfer energy as well as the promotion and tail energy of WiFi and 3G networks. Promotion energy is spent when the interface goes from idle to high power active state, whereas tail energy is spent in high power states at the end of the transfer [4]. We use 0.29 uJ/bit for WiFi transfers and 5.86 uJ/bit for 3G transfers [10]. For WiFi, we use 0.01J for promotion energy (0.08s) and 0.028J for tail energy (0.238s). For 3G, we use 0.383J for promotion energy (0.582s), 6.5J for DCH tail energy (8.0882s) and 0.5J for FACH tail energy (0.824s) [10].

V. Results

This section presents our quantitative results, applying dynamic delay tolerance schemes in data offloading and comparing them in terms of resulting cellular bytes usage, average effective delay and network energy usage. Unless otherwise stated, graphs are averaged over all users and the average WiFi duration and inter-connection times are 1 hour.

A. Fixed Delay Decision for Different Users

Fig. 6 compares the average effective delay results using *Fixed* for different DT_{est}. Using a high DT_{est} decreases the cellular data usage greatly. On average, 80 mins DT_{est} can reduce cellular data usage 35% more than *Greedy*, which already results in ∼50% cellular data reduction compared to all-cellular. On the other hand, with *Fixed*, higher DT_{est} also increases the effective delay at the receiver. Even though *Greedy* (non-delay-tolerant) does not reduce cellular usage as much as delay-tolerant schemes do, it incurs zero effective delay since it does not delay transmissions. For *Fixed*, 80 mins of delay tolerance causes the effective delay to be ∼22× larger than the value at 5 mins delay tolerance. Moreover, this effect is even worse for users with faster email reading behavior. For example, user 1 (our dataset's fastest email reader) experiences ∼1150s effective delay on average. This is ∼7× more than the slowest reader's experience. These results show that a fixed delay tolerance across all users is unlikely to work well.

B. Tradeoffs of Effective Delay and Cellular Usage

Fig. 7 shows the average effective delay-cellular bytes usage trade-off curves for all schemes. Plotting to show Pareto tradeoffs, better solutions (i.e. lower cellular bytes usage with lower effective delay) are near the origin in the lower-left corner. Each point on the curves represent different values for key parameters used in schemes (i.e. I and WS_H for *Adapt*, *scale* for $DTree$, DT_{est} for $Fixed$). Increases in I and *scale* lead to more aggressive delay tolerance decisions. *Greedy* and

Scheme	Transfer	Promotion + Tail	Total
Fixed	83 %	82 %	83 %
Adapt	74 %	240 %	78 %
Lazy	66 %	147 %	69 %
DTree	57 %	218 %	61 %

TABLE II: Average energy consumption of different schemes normalized to *Greedy*.

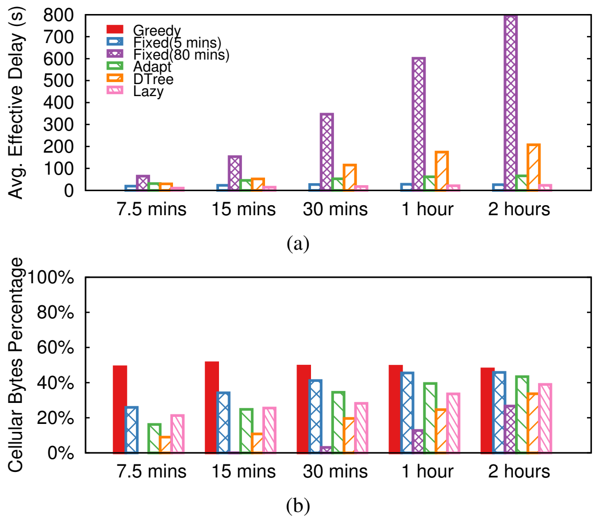


Fig. 10: Effective delay and cellular data usage at different WiFi settings. WS_H, I and $scale$ are set to one.

Lazy are shown as single points since they do not have any parameter variability.

Comparing Different Schemes: *Greedy* saves around 50% cellular bytes usage with zero effective delay on the receiver. Since *Lazy* postpones the transmissions until either there is WiFi or the user explicitly requests the data body, it achieves the maximum cellular bytes saving for the resulting effective delay. The average effective delay is ~20s while the cellular bytes usage is reduced to 30%. However, even though this is a small delay, this delay is actually explicitly exposed in the user interface, unlike the other schemes.

All the dynamic decision schemes have better trade-off curves than *Fixed*, because they better track DT_{est}. For example, in order to get ~75% total savings on cellular bytes, *Fixed* introduces 355s effective delay on average. On the other hand, *Adapt* (where window size is 1) and *DTree* (size + receive time) can achieve the same cellular bytes saving with much smaller effective delay. Similarly, if the goal is to introduce no more than 100s effective delay, *Fixed* can only save 60% of cellular bytes whereas *Adapt* (where window size is 1) and *DTree* (size + receive time) can save around 65% and 75%, respectively. When *DTree* uses both size and receive time information, accuracy improves. Both play an important role in gauging delay tolerance.

If we focus on *Adapt* where we fix I to 1 min and increase window size, the tradeoff curve gets closer to the *Fixed* curve and sometime gets even a little worse. This shows that using a longer history does not improve DT_{est}. If WS_H is set to 1 and I is increased, the curve gets closer to *Fixed*. This shows that if the increment value is high, then since DT_{est} changes more drastically, sometimes with higher inaccuracy.

Hybrid Scheme: Fig. 8 shows the trade-off curve for *Hybrid* when *DTree* and *Adapt* (8 and 64 mins increment values, window size 1) are used. Since *Hybrid* conservatively chooses the minimum DT_{est} decided by *Adapt* and *DTree*, it provides a trade-off curve which is between these two schemes. Even though *DTree* generally performs better than *Adapt*, there are some cases where *Adapt* achieves better results. *Hybrid* can provide a better trade-off between the two.

C. *Effective Delay on Different Users*

Fig. 9 shows per-user average effective delay for four schemes. For *Fixed*, *Adapt* and *DTree*, the operating point is chosen to keep the the cellular byte usage at 25%. For *Lazy*, which has no controllable parameters, cellular bytes usage is between 20% and 35% for different users. For almost all users, dynamic decision schemes result in better (lower) effective delay than *Fixed*, with *DTree* typically inducing less delay than *Adapt*. *Lazy* can achieve ~20s effective delay on average which is the lowest among all of the schemes. However, unlike the other schemes, this delay is actually experienced by the users since the metadata has exposed the existence of this email to them. We have also measured how DT_{est} tracks read latency, and have found that on average, only 25% of *DTree* DT_{est} estimations have more than 1000s over-estimation which is almost half of *Fixed*. To sum up, dynamic schemes can better estimate delay tolerance. For the same cellular bytes usage, dynamic decision schemes offer much better effective delays.

D. *Energy Usage*

Table II shows the ratio network energy usage of different schemes compared to *Greedy* when the effective delay is less than 100s. Overall, all decision schemes reduce the total energy compared to *Greedy* and *Fixed*. For example, compared to *Fixed* which already reduces *Greedy*'s network energy by 17%, *Adapt* and *DTree* reduce another 5% and 21%, respectively.

Table II also shows the ratio of transfer and tail + promotion energy separately. *Adapt*, *Lazy* and *DTree* have higher promotion + tail energy compared to other schemes. This is because these schemes make multiple transmissions for a single data request if WiFi is not available at the original transmission time. For example, if WiFi is not available, *Adapt* sends the metadata first and send the body later depending on DT_{est}. Moreover, the incoming receiver DT_{est} response can also trigger a change in network state. Especially for 3G, these different transmissions incur extra high promotion and tail energy. (If transmission occurs in the tail time before the network goes idle, it incurs less tail energy). Even though the promotion + tail energy can be very high compared to *Greedy* and *Fixed*, total energy is still much lower.

E. *Varying WiFi Availability*

Fig. 10 shows the average effective delay and cellular bytes usage at different WiFi duration and inter-connection times. WS_H, I and $scale$ are set to one as default. As the WiFi duration and inter-connection time decrease, cellular usage decreases for all schemes except for *Greedy*. Our schemes have better savings and lower effective delay at more frequent WiFi connections since frequent WiFi availability increases the probability of finding WiFi for every transmission.

Even though *Fixed* with 80 mins DT_{est} achieves the best cellular data usage, it can have up to 4× more effective delay compared to *DTree*. Even though *DTree* achieves the best tradeoff as before, *Adapt* can result in very low effective delay at all different WiFi availability. This is because *Adapt* estimates DT_{est} online using the immediate observations, hence it does not increase DT_{est} drastically even when WiFi is scarcely available.

F. Results Summary

Exploiting WiFi using application delay tolerance has great potential in reducing cellular data usage, but delay tolerance needs to be estimated properly to reduce the effective delay at receivers. Even though *Fixed* can save up to 85% cellular usage on average for our data trace when DT_{est} is high, it can incur a high $\sim$1150s effective delay on the fast email readers.

Dynamic decision schemes provide better tradeoffs between cellular bytes usage and effective delay. *DTree* can estimate delay tolerance the best and decrease the effective delay by 50% compared to *Fixed*. For individual users, this ratio can go up to 80% for the same cellular usage of 25%. *Adapt* is the most robust to scarce WiFi availability since it adjusts its decision online depending on the observed delay. *Lazy* provides the lowest effective delay of average $\sim$20s for 30% cellular data usage, but this delay is actually experienced by the users who are waiting for the email to read. All are easily implemented within real systems.

VI. Related Work

Network discovery and selection problems are defined in RFC 5113 [1], however this definition does not discuss application delay tolerance. Qualcomm introduced a framework for operators to improve network selection and benefit from the WiFi offloading [18]. However, the framework does not use application delay tolerance for offloading and it only focuses on network provider policies. Power/performance effects of using different wireless channels have also been investigated by previous authors [20, 22]. However, these do not exploit delay tolerance either.

Utilizing delay tolerance for lower 3G usage has been studied as a 3G/WiFi network selection problem [3, 12]. [3] develops a heuristic method for delaying and offloading transfers. In [12], the authors studied the potential cellular bytes savings from a delay-tolerant offloading scheduler. However, both these works assume that the application delay tolerance is already provided. They do not explore how delay tolerance could be estimated.

In our previous work [5], we formulate the WiFi/cellular scheduling problem as an optimization problem for cellular data usage. We also proposed simpler heuristics which are easier to build, provide close-to-optimal performance when WiFi prediction is accurate, and also more robust to real-world prediction errors compared to power- and compute-intensive optimal schedulers [6]. Moreover, we built a real-system implementation which support delaying transmissions. However, previous work does not propose solutions for estimating delay tolerance, instead relies either on the programmers or users themselves to provide it.

Previous researchers have extensively studied decision tree learning for data classification, data mining and machine learning in many different areas [7, 19, 23]. Our work relies on the basic fundamentals of such methods. More advanced decision tree algorithms can be used to improve the performance of our DTree method.

VII. Conclusion

Delay-tolerant WiFi offloading has great potential in reducing cellular data usage, but simple fixed schemes for estimating delay tolerance can expose too much delay to users. In our work, we first defined our impact metric, *effective delay* which can be used as performance metric for delay tolerance decision. We defined dynamic adaptive estimators that *learn* delay tolerance during execution, and that can be used to improve cellular bytes usage with much less impact on user experience. While our work focused here on email as a case study, our approach is more broadly applicable across important application classes including file transfer or online photo/file sharing.

References

[1] J. Arkko *et al.*, "Network discovery and selection problem," in *RFC 5113*. IETF, 2008.

[2] AT&T. (2012) Data plans from AT&T. [Online]. Available: http://www.att.com/shop/wireless/plans/data-plans.jsp?WT.srch=1&fbid=qJecLd0u-5

[3] A. Balasubramanian *et al.*, "Augmenting mobile 3G using WiFi," in *ACM MobiSys*, 2010.

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

[5] O. Bilgir Yetim and M. Martonosi, "Adaptive usage of cellular and WiFi bandwidth: An optimal scheduling formulation," in *Proc. 7th ACM Intl. Workshop on Challenged Networks (CHANTS)*, 2012.

[6] O. Bilgir Yetim and M. Martonosi, "Adaptive Delay-Tolerant Scheduling for Efficient Cellular and WiFi Usage," in *Proc. IEEE WoWMoM*, 2014.

[7] L. Breiman *et al.*, *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[8] V. Bychkovsky *et al.*, "A measurement study of vehicular internet access using in situ Wi-Fi networks," in *ACM MobiCom*, 2006.

[9] Cisco White Paper, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013-2018," 2014.

[10] J. Huang *et al.*, "A close examination of performance and power characteristics of 4G LTE networks," in *ACM MobiSys*, 2012.

[11] IDC, "Always Connected: How smartphones and social media keep us engaged," IDC Research, October 2013.

[12] K. Lee *et al.*, "Mobile data offloading: How much can WiFi deliver?" in *ACM Co-NEXT*, 2010.

[13] J. Maindonald and J. Braun, *Data Analysis and Graphics Using R*. Cambridge University Press, 2007.

[14] NISO, *Understanding metadata*, National Information Standards Organization, 2004. [Online]. Available: http://www.niso.org/standards/resources/UnderstandingMetadata.pdf

[15] E. Nordström *et al.*, "Serval: An end-host stack for service-centric networking," in *NSDI*, 2012.

[16] Oracle. JavaMail. [Online]. Available: http://www.oracle.com/technetwork/java/javamail/index.html

[17] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*. Morgan Kaufmann, 2007.

[18] Qualcomm Incorporated White Paper, "A 3G/LTE Wi-Fi Offload Framework: Connectivity Engine (CnE) to Manage Inter-System Radio Connections and Applications," 2011.

[19] J. R. Quinlan, "Induction of Decision Trees," in *Machine Learning*, 1986.

[20] A. Rahmati and L. Zhong, "Context-based network estimation for energy-efficient ubiquitous wireless connectivity," *IEEE Trans. Mobile Computing*, vol. 10, no. 1, 2011.

[21] P. W. Resnick, "Internet Message Format," Internet RFC 5322, 2008.

[22] P. Rodriguez *et al.*, "MAR: a commuter router infrastructure for the mobile internet," in *ACM MobiSys*, 2004.

[23] S. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Systems, Man and Cybernetics*, 1991.

[24] J. Smith and P. Schirling, "Metadata standards roundup," *MultiMedia, IEEE*, vol. 13, no. 2, pp. 84–88, 2006.

[25] Sun Microsystems, Inc. (2007) Package javax.mail. [Online]. Available: http://docs.oracle.com/javaee/5/api/javax/mail/package-summary.html