Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications

Lenin Ravindranath, *M.I.T. & Microsoft Research*
Jitendra Padhye, *Microsoft Research*
Ratul Mahajan, *Microsoft Research*
Hari Balakrishnan, *M.I.T.*

Abstract

Providing consistent response times to users of mobile applications is challenging because there are several variable delays between the start of a user's request and the completion of the response. These delays include location lookup, sensor data acquisition, radio wake-up, network transmissions, and processing on both the client and server. To allow applications to achieve consistent response times in the face of these variable delays, this paper presents the design, implementation, and evaluation of the Timecard system. Timecard provides two abstractions: the first returns the time elapsed since the user started the request, and the second returns an estimate of the time it would take to transmit the response from the server to the client and process the response at the client. With these abstractions, the server can adapt its processing time to control the end-to-end delay for the request. Implementing these abstractions requires Timecard to track delays across multiple asynchronous activities, handle time skew between client and server, and estimate network transfer times. Experiments with Timecard incorporated into two mobile applications show that the end-to-end delay is within 50 ms of the target delay of 1200 ms over 90% of the time.

1 Introduction

Interactive mobile applications, or "apps", are a large and rapidly growing fraction of software written today. Because users expect a timely response to their requests, app developers worry about responding to each request promptly. Responses that arrive within a predictable period of time improve the user experience, whereas the failure to provide consistent response times has adverse financial implications for even small degradations in response times [15, 5, 30].

This task is difficult enough for sophisticated developers and well-funded organizations, but for the legion of less-experienced developers with fewer resources at hand, the problem is acute. The problem is difficult because the end-to-end delay between a user's request and its response has several different components, each highly variable. For example, to service a user's action, the app may need to gather GPS or other sensor data on the mobile device, then form and transmit a request to one or more servers in the "cloud". The required network transmission may entail waking up the radio interface on the mobile device. After the server processes the request, the time required to transmit the response to the client is subject to numerous vagaries of the wireless network. Finally, even after the response reaches the mobile device, the time needed to render the response may vary depending on the client's hardware and OS.

In this paper, we focus on mobile apps that use servers in the cloud for some of their functions. Our goal is to develop a system for app developers to ensure that the end-to-end delay between the initiation of a request and the rendering of the response does not exceed a specified value. The system does not provide hard delay guarantees, but instead makes a best-effort attempt to achieve the delay goal.

Given the desired end-to-end delay, the idea is to allow the server to obtain answers to two questions:

1. *Elapsed time:* How much time has elapsed since the initiation of the request?
2. *Predicted remaining time:* How much time will it take for the client to receive an intended response over the network and then process it?

The server can use the difference between the desired delay bound and the sum of the elapsed time and predicted remaining time to determine the *work time* for the re-

LR and HB are with the Computer Science and Artificial Intelligence Lab (CSAIL) at M.I.T.
SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522717

quest. To control the end-to-end delay, the server should compute its response within the work time.

Although few services are designed with this flexibility today, many are amenable to such adaptation by striking a balance between response quality and work time. For example, speech-to-text services naturally produce results whose fidelity is roughly proportional to processing time [13, 7]. Similarly, search services spawn workers for different content types and aggregate results only from the workers that respond within a deadline [3]; different deadlines lead to different quality of results. Services can also adapt by changing the amount of resources used for request processing, the priority with which response is processed, or the scope of the work (e.g., radius for a location-based query). The adaptation mechanisms are service-specific and not the focus of our work; we focus on answering the two questions above.

Answering these questions poses several challenges. Tracking elapsed time requires accurate and lightweight accounting across multiple, overlapping asynchronous activities that constitute the processing of a request on both the mobile device and the server. When the request reaches the server, we must also factor in the clock skew between the client and the server. Inference of this skew is hindered by the high variability in the delay of cellular network links. Estimating remaining time is difficult because it depends on many factors such as device type, network type, network provider, response size, and prior transfers between the client and server (which dictate the TCP window size at the start of the current transfer).

We address these challenges by automatically instrumenting both the mobile app code, and the cloud service code. To this end, we extend the AppInsight instrumentation framework [29] to track the accumulated elapsed time, carrying this value across the stream of thread and function invocations on both the mobile client and server. We also develop a method to accurately infer clock skew, in which probes are sent only when the mobile network link is idle and stable. To predict the remaining time, we train and use a classifier that takes several relevant factors into account, including the intended response size, the round-trip time, the number of bytes already transferred on the connection prior to this response, and the network provider.

We have implemented these ideas in the Timecard system. To study its effectiveness, we have modified two mobile services to adapt their response quality using the Timecard API. Using these services and other data, we answer two questions. First, is Timecard useful in practice? We find that 80% of user interactions across 4000 popular Windows Phone apps that involved network communication could have benefited from Timecard. We also find that small reductions in work time for our two services lead to proportionally small reductions in the response quality, implying that Timecard would enable these services to effectively trade response quality for delay. Second, how often does Timecard meet the end-to-end delay bound for various network conditions and device types? We find that the response time is within 50 ms of the desired bound (1200 ms) 90% of the time. These results suggest that Timecard is a practical and useful way to build cloud-based mobile apps with predictable response times.

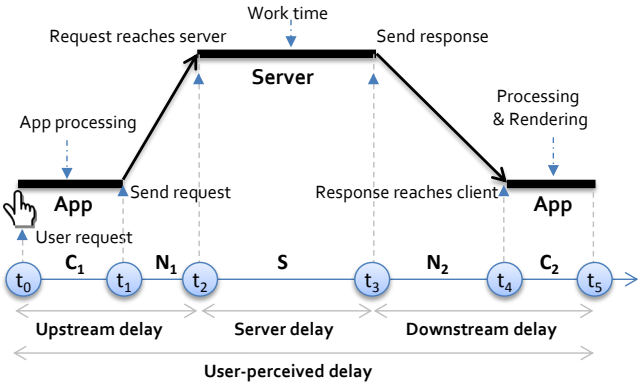


Figure 1: A typical transaction in an interactive app.

2 Timecard Architecture

Timecard uses the notion of a *user transaction* defined in AppInsight [29]. A user transaction in an app begins with a user request, expressed through a user interface (UI) action such as a button press, swipe, speech utterance, device shake, or gesture. The transaction ends with the completion of all synchronous and asynchronous tasks (threads) in the app that were triggered by the request.

Figure 1 shows the anatomy of a user transaction. The request starts at time t_0. The app does some initial processing, which entails local actions such as reading sensor data and possibly network operations like DNS requests. At time t_1 the app makes a request to the server, which reaches the server at time t_2. The server processes the request, and sends the response at time t_3, which reaches the client at time t_4. The app processes the response and renders the final results to the user at time t_5. In some cases, transactions have richer patterns that involve multiple calls sequential or parallel to the server. We focus on the single request-response pattern because, as we show in §6.1, it is dominant.

The *user-perceived delay* for this user transaction is the duration $t_5 - t_0$. (In some cases a background task may continue past the final user-visible task without impacting user-perceived delay.) User-perceived delays for mobile apps vary widely, ranging from a few hundred milliseconds to several seconds (§6.1).

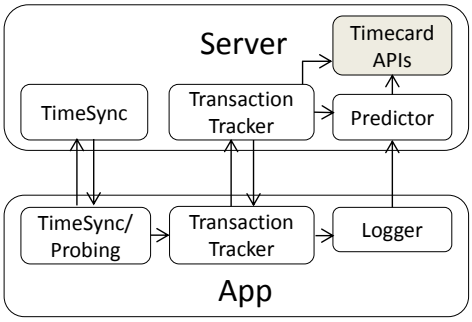


Figure 2: Timecard Architecture.

The *work time* at the server is $t_3 - t_2$. The client's processing is made up of two parts, $C_1 = t_1 - t_0$ and $C_2 = t_5 - t_4$, which correspond to the duration before the request is sent and the duration after the response is received. We denote the request ("uplink") and response ("downlink") network transfer times by N_1 and N_2, respectively: $N_1 = t_2 - t_1$ and $N_2 = t_4 - t_3$.

Timecard helps app developers control the user-perceived delay for user transactions. It provides an API with two functions for this purpose:

1. `GetElapsedTime()`: Any component on the processing path at the server can obtain the time elapsed since t_0.
2. `GetRemainingTime(bytesInResponse)`: At the server, a component can obtain an estimate of $N_2 + C_2$. Timecard provides this estimate as a function of the size of the intended response.

These two functions help control the user-perceived delay. Servers that generate fixed-size responses can infer how much time they have to compute the response by querying for elapsed time and for remaining time with the response size as input. Their work time should be less than the desired user-perceived delay minus the sum of times obtained from those API calls. Servers that can generate variable-sized responses can call this function multiple times to learn how much work time they have for different response sizes, to decide what response they should generate to stay within a given user-perceived delay. The desired user-perceived delay for a transaction is specified by the mobile app developer, based on the responsiveness needs of the app and other factors (e.g., how often the user is refreshing). The API may also be used for other purposes, as discussed in §7.

Determining the elapsed time requires tracking user transactions across multiple asynchronous threads and between the client and server, as well as synchronizing the time between the client and the server. Estimating the remaining time requires a robust way to predict N_2 and C_2. Figure 2 shows the high-level architecture of Timecard, depicting the information flow. Transaction tracking and time synchronization are described in detail in §3, while N_2 and C_2 prediction is covered in §4.

3 Tracking elapsed time

To track elapsed time, Timecard uniquely identifies each user transaction and tracks information about it, including its start time, in a *transaction context* object (§3.1). Timecard also synchronizes the time between the client and the server (§3.2). The transaction context is available to any client or server thread working on that transaction. The elapsed time is the difference between the thread's current time and the transaction's start time.

3.1 Transaction tracking

Transaction tracking is challenging because of the asynchronous programming model used by mobile apps and cloud services. Consider the execution trace of a simple app shown in Figure 3. On a user request, the app makes an asynchronous call to obtain its location. After getting the result on a background thread, the app contacts a server to get location-specific data (e.g., list of nearby restaurants). The server receives the request on a listening thread and hands it off to a worker thread. The worker thread sends the response, which is received by the app on a background thread. The background thread processes the response and updates the UI via a dispatcher call, completing the transaction.

To track the elapsed time for this transaction, Timecard passes the transaction's identity and start time across asynchronous calls, and across the client/server boundary.[1] Timecard instruments the client and the server code to collect the appropriate information, and stores it in a *transaction context* (TC) object (Table 1). The instrumentation techniques used by Timecard extend the AppInsight [29] framework in four key aspects: (i) Timecard's instrumentation tracks transactions on the client, on the server, and across the client-server boundary, whereas AppInsight tracks transactions only on the client; (ii) Timecard's instrumentation enables time synchronization between client and server (§3.2), unlike AppInsight; (iii) Timecard collects additional data to enable N_2 and C_2 prediction; and (iv) Timecard is an online system, while AppInsight collected user transaction data for offline analysis.

We now describe how TC is initialized and tracked (§3.1.1), how tracking TC enables Timecard to collect training data for predicting N_2 and C_2 (§3.1.2), and how TC is reclaimed upon transaction completion.

3.1.1 Transaction Context

Timecard identifies all UI event handlers in the app using techniques similar to AppInsight [29]. It instruments the handlers to create a new TC, assigning it a unique ID

[1] See (§7) for transactions that use multiple servers.

Tracked information	*Purpose*	*Set by*	*Used by*
Application Id	Unique application identifier	Client	Server and Predictor
Transaction Id	Unique transaction identifier	Client	Client and Server
Deadline	To calculate remaining time	Client	Server
t_3	To calculate N_2 for training data	Server	Predictor
t_4	To calculate N_2 and C_2 for training data	Client	Predictor
t_5	To calculate C_2 for training	Client	Predictor
Entry Point	To predict C_2, and to label training data	Client	Server and Predictor
RTT	To predict N_2, and to label training data	Client	Server and Predictor
Network type	To predict N_2 and to label training data	Client	Server and Predictor
Client type	To predict N_2 and to label training data	Client	Server and Predictor
Size of response from cloud service	To predict N_2 and to label training data	Server	Predictor
Pending threads and async calls	To determine when transaction ends	Client	Client

Table 1: Transaction context. The three timestamps are named as per Figure 1.

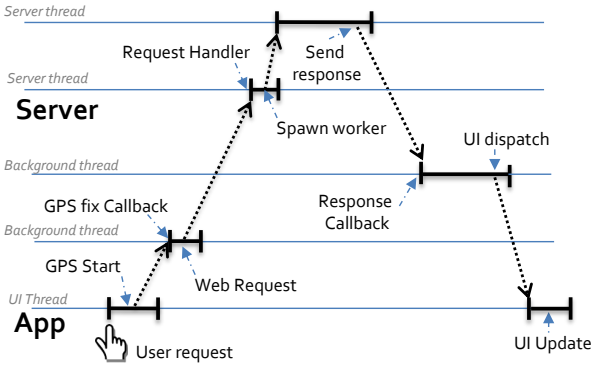


Figure 3: A location-based app that queries a server.

and timestamp t_0. It maintains a reference to the newly created object in the thread's local storage.

Tracking a transaction across asynchronous calls: To pass a reference to the TC from the thread that makes an asynchronous call to the resulting callback thread, Timecard builds upon AppInsight's "detouring" technique [29]. Before the asynchronous call is made, a unique tracking object is created at runtime. The object contains a method that encapsulates the original callback method, and has a signature that is identical to the callback method. Timecard rewrites the asynchronous call to include a reference to this newly created method. Thus, when the callback is made, we can match the callback to the right asynchronous call. Timecard includes a reference to the TC in the tracking object, which allows the thread that executes the callback to access the TC.

Passing TC from client to server: When an app makes a request to the server, the client passes some fields in the TC to the server (Table 1) by encoding it in a special HTTP header called `x-timecard-request`. To add the header to the HTTP request, Timecard modifies all HTTP request calls in the application.

Tracking transaction at the server: Timecard instruments the service entry methods that handle client requests to create a new TC object using the information specified in the `x-timecard-request` header. Timecard then tracks the TC across server threads using the same techniques as for client threads.

Handling server response and UI updates: When the response arrives, the client OS invokes a callback method to handle the response. This method has access to the correct TC due to the detouring technique described earlier. The method processes the response and updates the UI via asynchronous calls to a dispatcher.

3.1.2 Collecting Data to Predict C_2 and N_2

Transaction tracking also enables Timecard to collect the data to train the N_2 and C_2 predictors for subsequent transactions. Figure 1 shows that N_2 and C_2 may be calculated from t_3, t_4, and t_5. Timecard instruments the server to log t_3 just before it sends the response to the client. Timecard also records the number of bytes sent in the response. This information, along with transaction id, the device type, client OS, and network provider (Table 1) are sent to the predictor.

Timecard instruments the client's callback handler to log t_4 as well as the time of the last UI update, t_5. Once the transaction is complete (§ 3.1.3), the values of t_4 and t_5 along with the transaction id are sent to the predictor. To reduce overhead, this data is sent using a background transfer service on the mobile that schedules the transfer after the app terminates [6].

3.1.3 Tracking Transaction Completion

When a transaction completes, Timecard can remove the TC on the client. On the server, Timecard can remove the TC as soon as t_3 is recorded and sent to the predictor.

A transaction is complete when none of the associated threads are active and no asynchronous calls associated with the transaction are pending. Thus, to track transaction completion on client, Timecard keeps track of active threads and pending asynchronous calls. Because Timecard instruments the start and end of all upcalls, and is able to match asynchronous calls to their callbacks, it can maintain an accurate list of pending threads and asynchronous calls in the TC.

Tracking transaction completion also allows Timecard to detect *idle time* on the client. When there are no active transactions on the client, it means that the

app is currently idle (most likely waiting for user interaction). Timecard maintains a list of currently active transactions. When the list is empty, it assumes that the application is "idle".[2] Timecard uses the application's idle time in two ways. First, Timecard garbage-collects some of the data structures it needs to maintain to take care of several corner cases of transaction tracking. Second, Timecard uses the start of an idle period to trigger and process time synchronization messages (§3.2).

3.2 Synchronizing time

The timestamps in the TC are meaningful across the client-server boundary only if the client and the server clocks are synchronized. Timecard treats the server's clock as the reference and implements mechanisms at the mobile client to map its local time to the server's. The *TimeSync* component code to synchronize the two times is added to the client and server using binary instrumentation. The transaction tracker queries TimeSync on the client for a timestamp, instead of the system time.

Before describing our method, we note that two obvious approaches do not work. The first is to run the Network Time Protocol (NTP) [20] on the clients and servers. The problem is that NTP does not handle the significant variability in delay that wireless clients experience; for example, the 3G or LTE interface in idle state takes a few seconds to wake up and transmit data, and in different power states, sending a packet takes different amounts of time. The second approach is to assume that the device can obtain the correct time from a cellular base station or from GPS. Both approaches are problematic: cellular base stations do not provide clients with time accurate to milliseconds, many mobile devices may not have a cellular service, GPS does not work indoors, and also consumes significant energy. For these reasons, Timecard adopts a different solution.

We conducted several measurements to conclude that the clocks on smartphones and servers usually have a linear drift relative to each other, and that the linearity is maintained over long periods of time (§6). We assume that the delay between the client and the server is symmetric[3]. Given the linearity of the drift and the symmetry assumption, client and server clocks can be synchronized using Paxson's algorithm [26, 21]. Briefly, the method works as follows:

1. At time τ_0 (client clock), send an RTT probe. The server responds, telling the client that it received the probe at time τ_1 (server clock). Suppose this response is received at time τ_2 (client clock).
2. Assuming symmetric delays, $\tau_1 = \tau_0 + (\tau_2 - \tau_0)/2 + \epsilon$, where ϵ is an error term consisting of a fixed offset, c, and a drift that increases at a constant rate, m.
3. Two or more probes produce information that allows the client to determine m and c. As probe results arrive, the client runs robust linear regression to estimate m and c.

However, in case of clients connecting over wireless networks, delays introduced by radio wake-up [17] and by the queuing of on-going network traffic confound this method. These delays are variable, and could be anywhere between a few tens of milliseconds to a few seconds. We develop a new probing technique that is aware of the state of the radio and traffic to produce accurate and robust results. We apply this technique to synchronize the client with each of its servers.

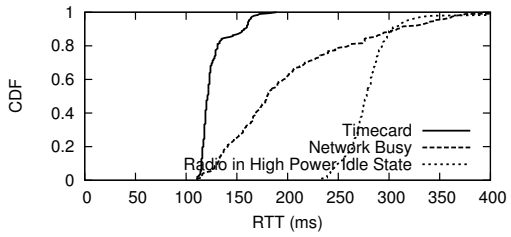
A useful insight is that the ideal time to send RTT probes is soon after a transaction's response completely arrives from the server, as long as no additional transfers are forthcoming. At this time, the radio will likely be in its high-power ("ready-to-transmit") state, ensuring that there is no wake-up delay and a lower marginal energy consumption relative to sending a probe when the radio is in any other state. Furthermore, the likelihood of the probe encountering queuing delay at either the client or the base station is also low because mobile devices typically run only one app in the foreground. Background apps are typically not scheduled when a foreground app is active. Base stations maintain per-device queues and implement fair schedulers, so queuing delays are likely to be low at this time. The methods used for client-side transaction tracking know when a transaction has ended and determine when an RTT probe should be sent.

Figure 4 shows the performance of our probing method. The graphs are based on data collected from an app that downloads between 1 and 50 Kbytes of data from a server over HSPA and LTE networks. The server and the app were instrumented with Timecard. Apart from the RTT probes sent by Timecard, the app sent its own RTT probes. These additional probes were carefully timed to ensure that they were sent either when the network was busy, or when the network was idle, and the radio was in an idle state (we used the Monsoon hardware power monitor to keep track of the power state of the radio interface). These results show that compared to the probes sent by Timecard, the additional probes experience highly variable round-trip delays, demonstrating the importance of sending probes only when the radio is in a high-power state and when the network is idle.

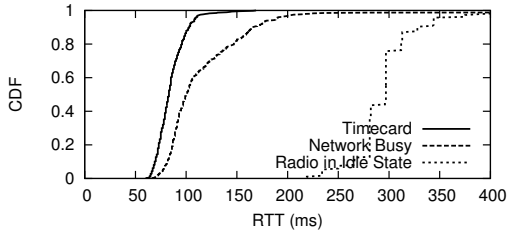
We conclude the discussion of TimeSync by noting a few additional features of this component. First, Time-

[2]This does not mean that the entire system is idle because other apps may be active in the background.

[3]NTP makes this assumption as well. Cellular links can have asymmetric delays, but the difference is typically small. See §6 for details.



(a) HSPA network. When the Radio is in Idle state, pings take 1800 ms to 2300 ms! (Not shown.)



(b) LTE network.

Figure 4: RTTs of probes from an app to a server with Timecard, when the network is busy, and when the radio is either idle or busy. (Note: There is no high-power idle state in LTE.) Timecard's probe transmissions strategy results in lower variability.

card includes an optimization not shown in the graphs above: it collects RTT samples only when the signal strength is above a threshold. The reason is that our data shows that uplink delays are highly variable when the signal strength is low. Second, to minimize the impact on app performance, Timecard computes the linear regression in a background process that runs only when no foreground app is running. Third, the TimeSync component of each app is independent because apps typically use different servers, which may each have a different notion of the current time.

4 Predicting Remaining Time

Timecard's `GetRemainingTime` function returns estimates of N_2 and C_2 for a specified response size. The sum of the two is the total amount of time required to receive and render the response at the client. The estimates are generated by decision tree algorithms that use models built from historical data.

4.1 Predicting N_2

N_2 is the amount of time required to transmit a specified amount of data from the server to the client. N_2 depends on a number of factors including the data size, the round-trip time (RTT) of the connection, the number of RTTs required to send the data, the bandwidth of the bottleneck link, and packet loss rate.

Our analysis of traces from over 4000 apps (§6.1), shows that (i) 99% of the data transfers are over HTTP (and hence TCP), and (ii) most are quite short – the 90th percentile of the response length is 37 KB, and median is just 3 KB. Hence our focus is to accurately predict duration of short HTTP transfers.

The duration of short TCP transfers over high-bandwidth, high-RTT, low-loss paths is determined primarily by the number of RTTs needed to deliver the data [24]. Modern cellular networks (3G, 4G, LTE) offer exactly such environment: bandwidths can high as 5Mbps, packet losses are rare [33]. However, RTTs can be as high as 200ms [33]. Thus, to predict N_2, we need to predict the RTT and estimate the number of RTTs required to transfer a given amount of data.

The number of RTTs required to download a given amount of data depends on the value of the TCP window at the sender when the response is sent. It would seem that the TCP window size and RTT can be easily queried at the server's networking stack. However, many cellular networks deploy middleboxes [32] that, *terminate and split* an end-to-end TCP connection into a server-to-middlebox connection and a middlebox-to-client connection. With such middleboxes, the server's window size or RTT estimate are not useful to predict N_2. Other factors that confound the prediction of N_2 include the TCP receiver window settings in the client OS, whether TCP SACK is used or not, and other TCP details. Under these circumstances, a method that measures the factors mentioned above and plug them into an analytic TCP throughput formula does not work well. Hence, we use an empirical data-driven model to predict N_2. After some experimentation, we settled on a model with the following features:

1. **The response size:** The size of the response, and TCP dynamics (see below), together determine the number of RTTs required.
2. **Recent RTT between the client and server:** We re-use the ping data collected by the TimeSync component (§3.2). We also keep track of TCP connection delay for these probes, to account for presence of middleboxes [32].
3. **Number of bytes transmitted on the same connection before the current transfer:** This feature is a proxy for the TCP window size at the sender, which can either be the server or the middlebox, if one is present. We are forced to use this metric because we have no way to measure the TCP window size at a middlebox. However, since TCP sender's window size generally grows with the number of bytes already sent over the connection, we can use the cumulative number of bytes that were previously transferred on the connection as a proxy for the sender's TCP window size.

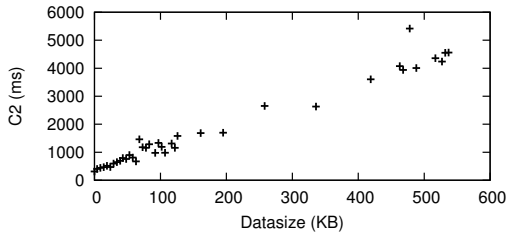


Figure 5: C_2 from one of the data-intensive user transactions in the Facebook app.

4. **Client OS version and client network provider:** This combined feature is a proxy for the TCP parameters of the client and the middlebox. The client OS version determines the maximum TCP receiver window size and other TCP details. The network provider is the combination of the cellular carrier (Verizon, AT&T, etc.) and network type (LTE, 4G, 3G, etc.). WiFi is a distinct network provider.

Each user transaction provides information about these features and the corresponding response time to the prediction module. The module buckets the observed response-time data into the features mentioned above. Multiple observed response time samples may map to the same bucket, creating a histogram of values for each bucket. The predictor is implemented as a decision tree on these features, It finds the best match among the buckets and returns the median[4] response time value for the bucket. The buckets used by the predictor are updated each time a Timecard-enabled app uploads the feature vector and response time information. Thus, this is an online predictor, with a constantly updating model.

The model used by the N_2 predictor is independent of the application or the service. Thus, we can combine data from multiple Timecard-enabled apps and services to build a more accurate model. We can also bootstrap the model by using offline measurements done by a dedicated measurement app (§6).

4.2 Predicting C_2

To understand the factors that affect the processing and rendering time on the client after the response is received (i.e. C_2), we analyzed thirty apps that had 1653 types of transactions. For most transactions, C_2 was highly correlated with the size of the response. Figure 5 plots C_2 for a popular transaction in the Facebook application, showing that C_2 is roughly linear in the response length.

C_2 typically includes two components: parsing delay and rendering delay. Many servers send data in the form of JSON, XML or binary (for images). On a mobile device, parsing or de-serializing such data takes a non-trivial amount of time. Our controlled experiments on popular off-the-shelf JSON, XML and image parsers show that, for a given data structure, this delay is linear in the data size. We also found that the rendering delay is linear in the data size consumed by the UI which is typically a subset of the response data.

Since the downstream processing is typically computation-bound, C_2 also depends on the device type and its processing speed. In general, it also depends on whether the current set of apps being run on the device is exhausting memory or CPU resources.

To predict C_2, we build a decision tree model similar to N_2 with app id, transaction type, device type, and response data size as the features[5]. The C_2 predictor continuously learns from previously completed transactions. After each transaction, the Timecard client logs the above specified features with a measured value of C_2 and sends it to the predictor. Thus, like the N_2 predictor, the C_2 predictor is also an online predictor. However, unlike the N_2 predictor, the C_2 predictor uses numerous models, one per transaction type (which includes the app id), making this predictor difficult to bootstrap. Currently, Timecard requires the app developer to provide rough models for the transaction types in the app, and refines them as more data becomes available. Without developer-provided models, Timecard can simply disable predictions until enough data is available.

5 Implementation

Timecard is implemented in C# with 18467 lines of code. It is currently targeted for Windows Phone Apps and .NET services. We do binary instrumentation of both the client- and server-side code. Our instrumentation framework is currently designed for .NET. Over 80% of the apps in the Windows Phone app store is written in Silverlight. Many web services are powered by .NET (for e.g. ASP.NET) and hosted through IIS. With the popularity of cloud providers such as Amazon Web Services and Azure, developers are able to easily host their services with minimal infrastructure support.

Incorporating Timecard into an app or a service requires little developer effort. We provide Timecard as a Visual Studio package, which can be added into a service or a app project workspace. Once added, it automatically includes a library into the project that exposes the Timecard APIs to the developer. It also modifies the project metadata to include a post-build step where it runs a tool to automatically instrument the built binary. When the instrumented server and the app are deployed, they jointly track transactions, synchronize time, estimate elapsed time, and predict remaining time.

[4] In future, we plan to experiment with other statistics such as the mean or the 90th percentile.

[5] We currently do not consider memory and CPU utilization.

Timecard does not require any modification to Silverlight, the Phone OS, IIS, or the cloud framework.

6 Evaluation

In §6.1 we demonstrate that network and client delays are highly variable, motivating the potential benefits of Timecard. In §6.2 we show that Timecard can successfully control the end-to-end delays for mobile apps. In §6.3 we measure the accuracy of the methods to predict N_2 and C_2. In §6.4 we validate the two key assumptions in the TimeSync component. Finally, we evaluate the overhead of Timecard in §6.5.

6.1 Is Timecard Useful?

The usefulness of Timecard depends on the answers to three questions. First, how common is the single request-response transaction (Figure 1) in mobile apps? This question is important because Timecard is designed specifically for such transactions. Second, how variable are user-perceived delays? Using Timecard, app developers can reduce variability and maintain the end-to-end delay close to a desired value. Third, how variable are the four components (C_1, C_2, N_1, N_2) of the user-perceived delay that Timecard must measure or predict? If these delays are not highly variable, a sophisticated system like Timecard may not be needed.

6.1.1 Common Communication Patterns

We study common communication patterns in mobile apps using the AppInsight and PhoneMonkey datasets (Table 2). The AppInsight dataset is based on 30 popular Windows Phone apps instrumented with AppInsight. We only instrument the clients because we have no control over the servers that these apps use. We persuaded 30 users to use these instrumented apps on their personal phones for over 6 months. Our dataset set contains over 24,000 user transactions that contact a server and 1,653 transaction types. This data set is an extended version of the dataset used in previous work [29].

Over 99% of the transactions in the dataset use HTTP-based request-response communication. Moreover, 62% of these transactions involve exactly one request-response communication of the form shown in Figure 1.

The dominance of this pattern is further confirmed by our study of 4000 top Windows Phone apps. We instrumented these apps with AppInsight and ran them using an UI automation tool called PhoneMonkey. PhoneMonkey runs the apps in an emulator. It starts the app, selects a random UI control on the current screen, and acts on it to navigate to the next screen. Across all apps, we obtained over 10,000 unique user transactions with at least one request to a server. We call these traces the PhoneMonkey dataset. Over 80% of PhoneMonkey transactions have the single request-response pattern.

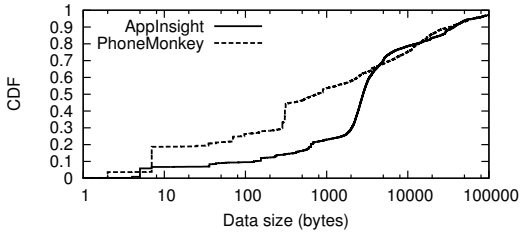


Figure 6: Size of data downloaded by apps in the AppInsight and PhoneMonkey datasets.

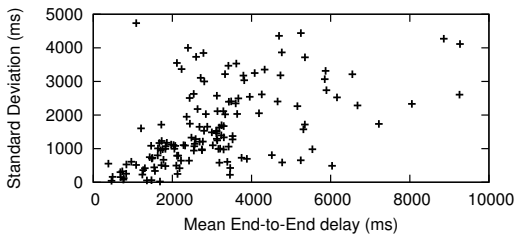


Figure 7: Mean user-perceived response time and its standard deviation for different transactions.

Recall that our network prediction model is geared towards *short* HTTP transfers (§4.1). Figure 6 shows the amount of data downloaded in the AppInsight and PhoneMonkey data. The median is only about 3 KBytes, and the 99th percentile is less than 40 KBytes.

To summarize: Timecard addresses the dominant communication pattern in today's mobile apps.

6.1.2 Variability of User-perceived Delay

Figure 7 shows a scatter plot of user-perceived delay and its standard deviation for different types of user transactions in the AppInsight dataset. Each point corresponds to a unique transaction type. We see that the user-perceived delays for a transaction are high—the mean delay is more than 2 seconds for half of the transactions—and also highly variable. This finding highlights the need for a system like Timecard that can control the variability.

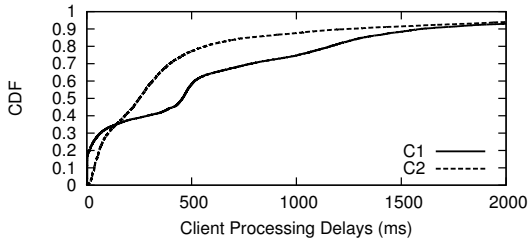
6.1.3 Variability of Individual Components

We now show that client-side processing (C_1 and C_2) and network transfer times (N_1 and N_2) both contribute to this variability.

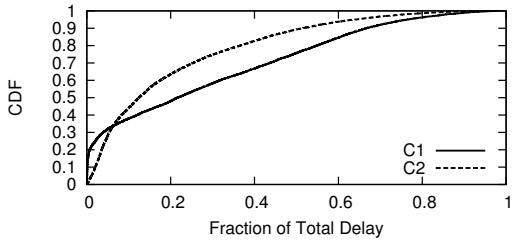
Client processing delays (C_1 and C_2): Figures 8(a) and 8(b) show the absolute values of C_1 and C_2 and the fraction they contribute to the user-perceived response delay seen in the AppInsight data set. The median delays

Name	Summary	Used in
AppInsight	30 instrumented apps, 30 users, 6 months. Over 24K network transactions.	6.1, 6.3
PhoneMonkey	4000 instrumented apps driven by UI automation tool.	6.1
NetMeasure	250K downloads over WiFi/3G/HSPA/LTE over ATT/Sprint/Verizon/TMobile. 20 users + lab. 1 month.	6.1, 6.3
EndToEnd	2 instrumented apps on 20 user phones sending requests to 2 instrumented services. Over 300K transactions.	6.2

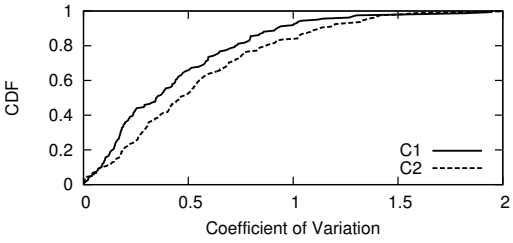
Table 2: Data sets used for evaluation of Timecard.



(a) Absolute values



(b) Fraction of total delay



(c) Coefficient of variation

Figure 8: Client processing delays.

are around 500 and 300 ms for C_1 and C_2, while the median fractions are 0.3 and 0.15, respectively. Figure 8(c) shows the Coefficient of Variation (CoV) (σ/μ) for each unique transaction type. The median values of CoV for C_1 and C_2 are 0.4 and 0.5, suggesting high variability. We discussed the reasons for the variability of C_1 and C_2 in §3 and §4.

Networking delays (N_1 and N_2): The AppInsight data cannot be used to analyze N_1 and N_2 because it does not have server-side instrumentation. Thus, we built a custom background app for Windows Phone and Android. The app periodically wakes up and repeatedly downloads random amounts of data from a server. Between successive downloads, the app waits for a random amount of time (mean of 10 seconds, distributed uniformly). The download size is drawn from the AppInsight distribution (Figure 6). The app and the server run the TimeSync method and log N_1 and N_2.

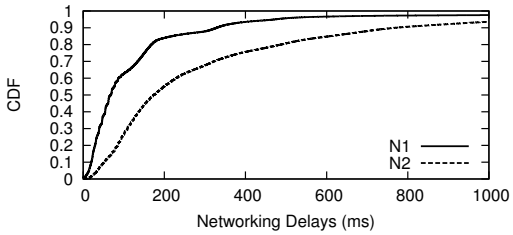


Figure 9: Network transfer delays.

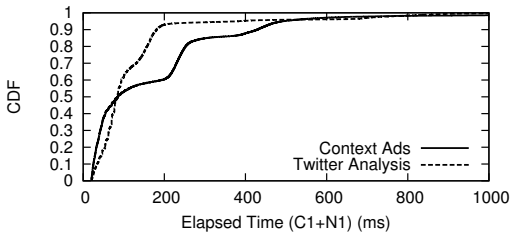


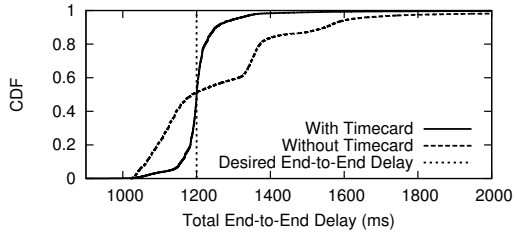
Figure 11: Elapsed time ($C_1 + N_1$) from two apps.

We ran the app on the personal phones of 20 users, all in the same U.S. city, as well as four Android test devices in two different cities. These phones used a variety of wireless networks and providers such as 3G, 4G (HSPA+), and LTE, on AT&T, T-Mobile, Verizon, and Sprint. The users went about their day normally, providing data across different locations and mobility patterns (indoors and outdoors, static, walking, driving, etc.). The interval between successive wake-ups of the apps was set to anywhere between 1 and 30 minutes depending on user's preference. In all, we collected data from over 250K downloads over a period of one month. We term this the NetMeasure dataset (Table 2).

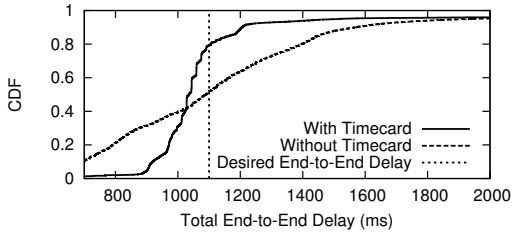
Figure 9 shows the CDF of N_1 and N_2. We see that the delays are both high and highly variable. The median delays are 75 and 175 ms, respectively. 30% of the N_2 samples are over 400 ms. Given the values of user-perceived response times in mobile apps (Figure 7), these delays represent a substantial fraction of the total.

6.2 End-to-End Evaluation

To conduct an end-to-end evaluation of Timecard, we incorporated Timecard into two mobile services and their associated apps. The apps were installed on the primary mobile phones of twenty users, configured to run in the background to collect detailed traces. We term these traces the EndToEnd data set (Table 2).

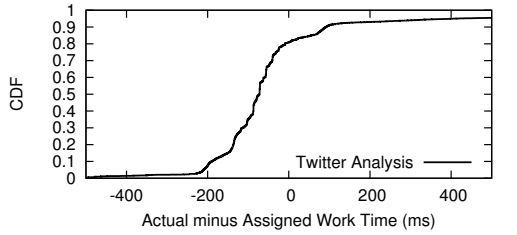


(a) Context ads delay.

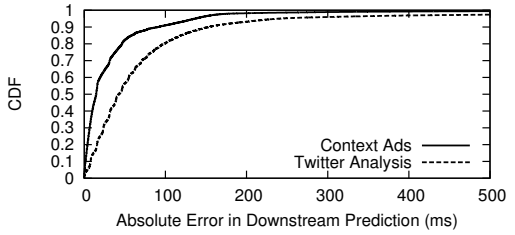


(b) Twitter analysis delay.

Figure 10: User-perceived delays for two apps. With Timecard, delays are tightly controlled around the desired value.



(a) Error due to discrete work times



(b) Downstream prediction error

Figure 12: Deadline control errors: discrete work time in one app and N_2 prediction errors.

We first describe the services and the associated apps. Next, we show that Timecard helps apps not exceed a desired end-to-end delay. Finally, we discuss the quality vs. timeliness trade-off for these two services.

The first service is an ad server that delivers contextual ads to apps [22]. The ad server is coupled with a mobile ad control, which is a small DLL that the app developer incorporates into the app. At run time, the mobile ad control scrapes the page displayed by the app for keywords and forwards them to the ad server. The server spawns multiple requests to an ad provider using these keywords. It sorts the received ads according to their relevance and returns the top ad to the app. The returned ad is a small text string, less than 1KB in size. The ad provider needs at least 500 ms to generate one response. By waiting longer, the server can receive additional responses from the ad provider, which can improve the relevance of the returned ad. Hence, there is a trade-off between server work time and the ad quality. The ad server uses the API described in §2 to determine how long the service should wait before sending an ad to the client. Note that the specified delay is not a hard deadline; Timecard tries to keep the actual delay around the specified value, seeking to reduce the delay variance around that value.

We built a simple app and added the ad control to it. The app wakes up at random times and feeds randomly selected keywords based on data in [22] to the ad control. We set the desired end-to-end delay for fetching ads to be 1.2 seconds.

The second service is a Twitter analysis service, with an associated mobile app that has been in the Windows Phone store for over 2 years. The app lets the user specify a keyword, which it sends to the analysis service. The service fetches recent tweets for the keyword, categorizes them into positive and negative tweets (sentiment), and finally sends an aggregated sentiment score to the app.

We modified the app to specify a desired delay of 1.1 seconds in addition to specifying the keyword. The server uses Timecard to decide how many tweets to fetch and analyze, given the desired delay. The quality of the response (sentiment analysis and the aggregated score) improves with the number of tweets, but fetching and analyzing more tweets takes more time. If more tweets are fetched, the size of the response sent to the app increases as well. The service sends between 8 KB to 40 KB of data per request. The app simply parses and renders the response.

Because of restrictions imposed by Twitter's web API, the service can only fetch and process tweets in multiples of 100, so the work time can be adjusted only in steps of roughly 150 ms. As a result, we cannot always meet the deadline precisely, but the server attempts to ensure that the user-perceived delay is smaller than the 1.1-second deadline. We pre-computed the expected

work times for fetching and analyzing different numbers of tweets by separately profiling the service.

We bootstrapped the N_2 predictor for both services using the NetMeasure data set. We bootstrapped the C_2 predictor using offline measurements.

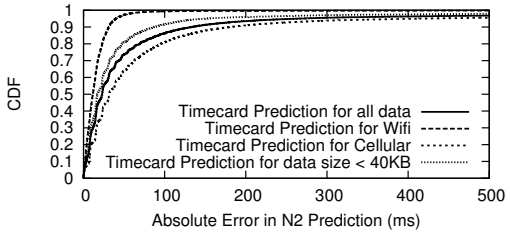
Figure 10 shows that with Timecard these two apps achieve user-perceived delays that are tightly distributed around the desired value. This result is significant because the upstream elapsed time when the request hits the server is highly variable, as shown in Figure 11. Over 90% of the transactions are completed within 50 ms of the specified deadline for ad control.

The difference between the observed and the desired delay is due to two main factors. For the Twitter analysis service, the work time is limited to be a multiple of 150 ms. Figure 12(a) shows that this causes 80% of the transactions to finish before the deadline, and over half the transactions to finish 50 ms early. The error in N_2 and C_2 prediction is the other main reason for the observed delay being different than the desired delay. Figure 12(b) shows that the median error in $N_2 + C_2$ is only 15 ms for the ad control app, because the service returns a small amount of data for each request. The median error is higher (42.5 ms) for the Twitter analysis app. TimeSync error also likely contributes to the downstream prediction error; unfortunately we have no way of measuring its precise impact.

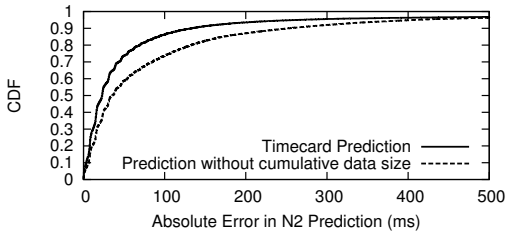
As the two services described above try to meet the end to end deadline, they trade-off quality of results for timeliness of response.

Figure 13(a) shows the trade-off between the ad server work time and probability of fetching the best ad. Recall that we had set the total deadline to be 1.2 seconds. Thus, the best ad is the ad that would have been top rated if the server had spent the entire 1.2 seconds on the job. If the server spends less time, it may not always find the best ad. Using trace data from 353 apps, 5000 ad keywords [22], and about 1 million queries to the ad server, we calculate the probability that the best ad is found for various work times. As one might expect, the probability increases as the server spends more time. Similarly, Figure 13(b) shows the trade-off between fetching and analyzing different number of tweets (mapped to average server work time) and the quality of sentiment analysis. The data is based on over 150,000 tweets for 100 popular keywords. We see that the as the server spends more time to fetch and analyze more tweets, the error in aggregated sentiment score compared to fetching and analyzing the maximum of tweets (1500) from twitter is reduced.

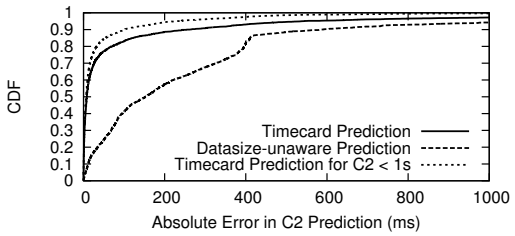
We stress that these trade-off curves are specific to each service; we do not claim that the above curves are representative in any manner.



(a) N_2 Prediction error



(b) N_2 Prediction with and without cumulative data



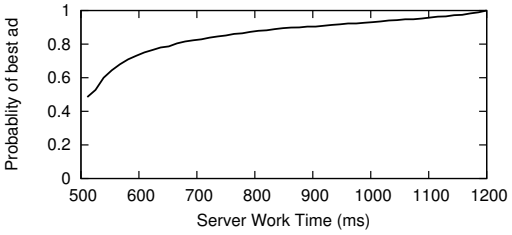
(c) C_2 Prediction error

Figure 14: Accuracy of N_2 and C_2 prediction

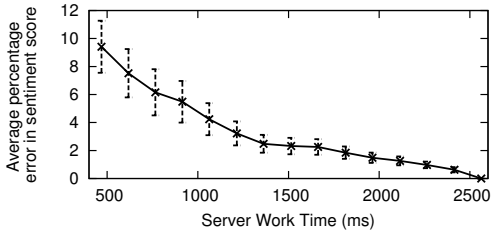
6.3 Prediction Accuracy

Accuracy of N_2 Prediction: We evaluate accuracy of N_2 prediction using the NetMeasure dataset (Table 2). We randomly split the data into two halves for training and testing. Figure 14(a) shows the CDF of absolute errors in the prediction. The median error is 23 ms; the 90th percentile error is 139 ms. To dig deeper, we look at WiFi and cellular links separately. We find that our prediction is more accurate for WiFi (median error 11.5 ms median, 90th percentile 31 ms) than it is for cellular networks (median 31 ms, 90th percentile 179 ms). Some of the longer tail errors (>100 ms) for cellular networks are due to radio wake-up delays on the downlink. In certain device models and carriers, the radio does not go to highest power state during upload, since upload transfers (i.e. client requests) are assumed to be small. Full wake-up happens only when the download begins.

The data size also has an impact on prediction delay, due to complex interactions between server TCP state, middlebox TCP state, and client TCP parameters. For smaller data sizes, these interactions do not matter as much, so the prediction error is low when we download less than 37 KBytes (median 17 ms, 90th percentile 86



(a) Context ads server trade-off.



(b) Twitter analysis server trade-off.

Figure 13: Trade-off between server work time and quality of results.

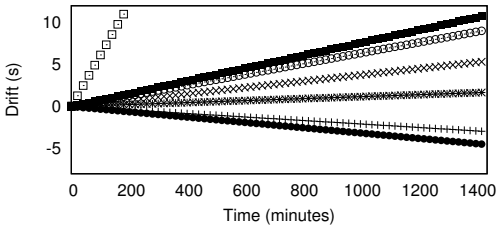


Figure 15: Clock drift in smartphones.

ms). Recall that in the AppInsight data set, 37 KBytes represents the 90th percentile download size (Figure 6).

Recall from §4.1 that we use the amount of data already transferred on the connection as a coarse way of modeling the TCP window behavior at the middlebox or the server. Figure 14(b) shows that it is important to include this feature in the model. Without the cumulative data sent, the median error in N_2 prediction is 54% higher, and almost double for the 90th percentile.

Accuracy of C_2 prediction: We use the AppInsight data set to evaluate the accuracy of C_2 predictor. In 30 apps, we identity 100 transaction types that have at least 20 transactions each from different users or different sessions. We use half the data for training and the other half for testing. Figure 14(c) plots the absolute error in C_2 prediction. The median error is 8 ms, but the 90th percentile error is 261 ms. When normalized for transaction duration, the median error is 4.6%, while 90th percentile is 22%. Both the percentage and absolute errors are low for shorter transactions. The graph shows that for transactions with $C_2 < 1$ second, the 90th percentile C_2 prediction error is 100 ms (10%). It also shows that C_2 predictor must take the size of the downloaded data into account. Without it, the median error is over 150 ms.

6.4 TimeSync

Our TimeSync method assumes that the clock drift is linear and that the uplink and downlink delays are symmetric. We now test these hypotheses.

We connected a smartphone to a desktop machine and sent TimeSync RTT probes from the smartphone to the desktop over the low delay USB link. We found that the combined drift between the smartphone clock and desktop clock is linear, and stayed linear over several days. We repeated this experiment on many different smartphone models and obtained similar results. Figure 15 shows the clock drift on seven different smartphones over a day. A simple linear regression fits the data and the mean error is 0.8 ms.

Cellular networks can have asymmetric uplink and downlink delays [17]. To estimate the asymmetry, we connected a smartphone to a desktop and sent probes from the desktop through the phone's cellular connection (tethering), back to the desktop's Ethernet connection. By using a single clock to measure uplink and downlink delays, we can measure the difference between the two (i.e., the asymmetry). We find that for three LTE networks, the difference between the uplink and downlink delay is less than 5 ms. But on 3G networks, the difference can be as high as 30 ms. The error in time synchronization can be as high as this difference, which impacts the accuracy of the elapsed time estimation. Thus, highly asymmetric links can cause the Timecard to miss the overall deadline. We also find that low signal strength greatly impacts the cellular uplink, making the probe delays asymmetric. Thus, we do not collect probes samples when the signal strength is low.

6.5 Overhead

To quantify the overhead of Timecard, we use an HTC Mazaa running Windows Phone 7.1 as client and an HP Z400 2.8 GHz dual-core with 16 GB RAM as server.

App run time: The impact of Timecard on app's run time is negligible. The average overhead of tracking an edge in the transaction graph is 50 μs. For the apps in the AppInsight data set, we estimate that the average total increase in app's run time would be 2 ms, which is less than 0.1% of the average transaction length. Overhead of sending and processing of RTT probes is minimal, due to various optimizations described in (§3.2). Timecard increases app launch time slightly (2 ms), since it needs to initialize various data structures. Regular garbage collection and bookkeeping of various data structures

is done during app idle time. The AppInsight data set shows that all apps have more than 10% idle time, which is sufficient for our needs.

Service run time: The overhead of Timecard at the server is small. The average time required for tracking an edge is less 10 μs. Overall, for the two services we instrumented, Timecard adds less than 0.1 ms to processing of each request.

Memory: Timecard consumes between 20 KB to 200 KB of additional memory to keep track of various data structures. Since the average memory consumption of apps in the AppInsight data set is 25 MB, the memory overhead of Timecard is less than 1%. On the server, the memory overhead of Timecard is negligible.

Network: Timecard consumes network bandwidth during app execution to send transaction context to server (§3.1) and to send RTT probes for TimeSync (§3.2). It also sends log data to the predictor to improve the prediction models. The size of the extra header is only 50–100 bytes. In rare cases, however, adding extra bytes can increase the request size just enough so that TCP incurs an extra round trip to send the request. TimeSync probes are small packets and transfer only a few bytes of data. The amount of data sent to predictor per transaction is just 20 bytes. Furthermore the training data is uploaded using background transfer. The total network overhead is less than 1% for the apps we instrumented.

The server incurs roughly the same network overhead. Most cloud services are deployed in well-provisioned data centers, and the marginal overhead is insignificant.

Battery: The battery overhead of Timecard that results from additional network usage is worth discussing; the CPU overhead is small. We time our RTT probes to avoid a radio wake-up power surge (§3.2). The battery impact of the few additional bytes sent in each request header is small. Thus, although we have not actually measured the marginal battery consumption, we see no reason why it would be significant.

7 Discussion and Limitations

Limitations of the N_2 predictor: Our approach for predicting N_2 has several limitations. First, for large transfer sizes, the number of RTTs matters less than the bottleneck rate of the connection. This limitation does not matter much for our purposes, because our focus is on request-response interactions (the common case for mobile apps). Second, a cellular provider could arbitrarily alter middlebox parameters, so the learning has to be continuous and may require retraining. In our experiments we observed consistent middlebox behavior for over a month, but that behavior may not always hold. Third, our model does not use the client's location as a feature. A network provider could deploy differently-behaving middleboxes in different areas, reducing the predictor's effectiveness. If that is observed, we would need to include the location as a feature. Fourth, our predictor depends on recent RTT samples. For geo-replicated servers, we could end up measuring RTT to a different server than the one client eventually downloads data from. If that happens, our prediction can be erroneous. In practice, we believe that this situation is uncommon because of the nature of replica selection algorithms.

Complex transactions: We focused on user transactions that included a single request to a cloud service. However, Timecard can be extended to more complex patterns (parallel or sequential requests to servers, complex dependencies between server requests, etc.) as well. For the `GetElapsedTime()` call, Timecard needs to ensure that the right timestamp is used with the right server. Extending `GetRemainingTime()` is more complex, and may require the developer to apportion budgets among multiple servers.

Privacy and security: Timecard does not collect any information that app developers cannot collect for themselves today. However, any logging and tracing system must carefully consider privacy implications, a topic for future work. For example, it is worth investigating the smallest amount of information that Timecard needs to log to function effectively. There are other security implications as well. For example, clients may manipulate the transaction data sent to the server, so that they get the "best possible" service.

Server processing time: We have not shown that the server processing time (S in Figure 1) is a significant portion of the user-perceived delay for popular mobile apps. To do so, we would need to instrument several third-party mobile services[6], which is a challenging, if not impossible, task. We also note that while several services such as search offer a clear trade-off between processing time and quality of results, such a trade-off is not possible for all services. However, even such services can use Timecard, as discussed next.

Other applications of Timecard: The API functions `GetElapsedTime()` and `GetRemainingTime()` can be used even by services that cannot control the response quality versus processing time trade-off. For instance, a server can use the APIs to prioritize the order in which requests are served, so that requests most in danger of missing user-perceived delay deadlines are served first. The server can also allocate different amount of resources to requests based on their deadline. A component on the mobile device may use the `GetElapsedTime()` to decide not to contact the

[6]Without such instrumentation, we cannot tease apart S and N_2.

server but use a cached response if the elapsed time is already too long. Alternatively, if the request involves the delivery of speech or sensor samples, it can adjust the sampling rate depending on the elapsed time. We leave the exploration of such alternative uses to future work.

Applicability to other platforms: The current implementation of Timecard focuses on Windows Phone apps and .NET services. However, we believe that Timecard can be easily ported to other platforms as well. In Timecard, the N_2 predictor, the C_2 predictor, and TimeSync are independent of the framework and can be reused with small modifications. The instrumentation and transaction tracking are specific to Silverlight and .NET. The client instrumentation can be ported to other mobile platforms [29]. The server instrumentation can be ported to any platform where we can correctly identify the entry points of a service, as well as all thread synchronization primitives.

8 Related Work

Mobile app monitoring and analysis: Timecard extends AppInsight's [29] instrumentation framework. AppInsight is primarily an analytic tool, focused on client performance. In contrast, Timecard allows developers to manage user-perceived delays. SIF [16] is another system closely related to AppInsight. Unlike AppInsight, SIF includes a programming framework to help the developer instrument selected code points and paths in the code. Like AppInsight, SIF focuses on client performance only. Other systems for monitoring mobile apps have primarily focused on profiling battery consumption [28, 25]. Flurry [11] and PreEmptive [27] provide mobile app usage monitoring. These systems do not provide tools for managing end-user response times, nor handle server-based mobile apps.

Predicting mobile network performance: A number of recent studies have focused on mobile network performance; we discuss two recent ones that have focused on prediction. Sprout [33] is a UDP-based end-to-end protocol for mobile applications such as videoconferencing that require both low delays for interactivity and high throughput. Sprout uses a model based on packet inter-arrival times to predict network performance over short time periods. Proteus [34] passively collects packet sequencing and timing information using a modified socket API, and uses a regression tree to predict network performance over short time periods. Proteus is also primarily applicable to UDP flows. Timecard can use any improvements in techniques to predict mobile network performance. Our implementation does not borrow from either Sprout or Proteus, however, because our primary focus is on apps that use TCP.

Server performance monitoring: The literature on monitoring transactions in distributed systems goes back several decades. We highlight three recent proposals. Magpie [4] monitors and models server workload, but unlike Timecard, has no client component. XTrace [12] and Pinpoint [8] trace the path of a request using a special identifier. Timecard uses similar techniques, with a focus on managing end-to-end delays.

Data center networking: Much effort has been devoted to understanding and minimizing delays in data centers that host delay-sensitive services. This work includes new network architectures [14, 2, 18], new transport protocols [3, 31], and techniques to rearrange computation and storage [1, 23]. None of these proposals is concerned with managing end-to-end deadlines.

Time synchronization: A number of innovative proposals for time synchronization in various domains, such as the Internet [26, 21, 20], wireless sensor networks [10, 19], and large globally-distributed databases [9] have been developed. Timecard currently uses algorithms proposed in [26, 21], but can benefit from any appropriate advances in this area.

9 Conclusion

Timecard helps manage end-to-end delays for server-based interactive mobile apps. For any user transaction, Timecard tracks the elapsed time and predicts the remaining time, allowing the server to adjust its work time to control the end-to-end delay of the transaction. Timecard incorporates techniques to track delays across multiple asynchronous activities, handle time skew between client and server, and estimate network transfer times. Our experiments showed that Timecard can effectively manage user-perceived delays in interactive mobile applications. These results suggest that Timecard's API functions may be used by developers to re-design their services to achieve good trade-offs between the quality of responses to requests and user-perceived delay.

Acknowledgments

We thank the SOSP reviewers and our shepherd, Mike Swift, for several useful comments and suggestions that improved this paper. LR and HB were supported in part by the National Science Foundation under Grant 0931508 and the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT).

References

[1] A. Agarwal, S. Kandula, N. Bruno, M. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-Parallel Computing. In *NSDI*, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[5] J. Brutlag. Speed matters for Google web search. http://googleresearch.blogspot.com/2009/06/speed-matters.html#!/2009/06/speed-matters.html, 2009.

[6] Background Transfer Service for Windows Phone. http://msdn.microsoft.com/en-us/library/hh202955(v=vs.92).aspx.

[7] C. Chelba, D. Bikel, M. Shugrina, P. Nguyen, and S. Kumar. Large scale language modeling in automatic speech recognition. Technical report, Google, 2012.

[8] M. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Mangement. In *NSDI*, 2004.

[9] J. C. Corbett et al. Spanner: Googles Globally-Distributed Database. In *OSDI*, 2012.

[10] J. Elson and D. Estrin. Time Synchronization for Wireless Sensor Networks. In *IPDPS*, 2001.

[11] Flurry. http://www.flurry.com/.

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[13] G. Görz and M. Kesseler. Anytime algorithms for speech parsing? In *Proceedings of the 15th conference on Computational linguistics*, 1994.

[14] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[15] J. Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx.

[16] S. Hao, D. Li, W. Halfond, and R. Govindan. SIF: A Selective Instrumenation Framework for Mobile Applications. In *MobiSys*, 2013.

[17] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *MobiSys*, 2012.

[18] S. Kandula, J. Padhye, and P. Bahl. Flyways To De-Congest Data Center Networks. In *HotNets*, 2009.

[19] B. Kusy, P. Dutta, P. Levis, M. Maroti, A. Ledeczi, and D. Culler. Elapsed Time on Arrival: A simple and versatile primitive for canonical time synchronization services. *International Journal of Ad hoc and Ubiquitous Computing*, 2(1), 2006.

[20] D. Mills, J. Martin, J. Burbank, and W. Kiasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, 2010.

[21] S. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay measurements. In *INFOCOM*, 2009.

[22] S. Nath, F. Lin, L. Ravindranath, and J. Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *MobiSys*, 2013.

[23] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *OSDI*, 2012.

[24] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *SIGCOMM*, 1998.

[25] A. Pathak, Y. C. Hu, and M. Zhang. Where Is The Energy Spent Inside My App? Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys*, 2012.

[26] V. Paxson. On calibrating measurements of packet transit times. In *SIGMETRICS*, 1998.

[27] Preemptive. http://www.preemptive.com/.

[28] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-Layer Approach. In *MobiSys*, 2011.

[29] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI*, 2012.

[30] E. Shurman. The user and business impact of server delays, additional bytes, and http chunking in web search. O'Reilly Velocity, 2009.

[31] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM*, 2009.

[32] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *SIGCOMM*, 2011.

[33] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

[34] Q. Xu, S. Mehrotra, Z. M. Mao, and J. Li. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications . In *MobiSys*, 2013.