Prospects for Shaping User-centric Mobile Application Workloads to Benefit the Cloud

Maciej Swiech
Northwestern University

Huaqian Cai
Peking University

Peter Dinda
Northwestern University

Gang Huang
Peking University

***Abstract*—Approaches to making cloud operation more efficient, for example through scheduling and power management, largely assume that the workload offered from mobile, user-facing applications is a given and that the cloud must simply adapt to it. We flip this assumption 180 degrees and ask to what extent can we instead shape the user-centric workload into a form that would benefit such approaches. Using a toolchain that allows us to interpose on frontend/backend interactions in popular Android applications, we add the ability to introduce delays and collect information about user satisfaction. We conduct an "in the wild" user study using this capability, and report on its results. Delays of up to 750 ms can be introduced with little effect on most users, although this is very much user and application dependent. Finally, given our study results, we consider reshaping the application requests by selective delays to have exponential interarrival times (Poisson arrivals), and find that we are often able to do so without exceeding the user's delay tolerance.**

I. Introduction

Policies for scheduling, mapping, resource allocation/reservation, power management, and similar mechanisms are generally designed with the assumption that the offered workload itself is sacrosanct. Even for closed systems, we hope that the system will have minimal effect on the offered workload. For the present paper, we consider three parts of this assumption: (1) the workload's statistical properties are a given, (2) the overall offered load is a given, and (3) the performance requirements are uniform across users. The design of a policy typically is then focused on the goal of minimizing cost, power, energy, latency, etc. given these.

As an example, consider making a modern user-facing cloud application such as Pinterest, Pandora, or Google Translate more energy or economically efficient. The application consists of a user interface (the "frontend"), for example an app on a smartphone, that communicates with the core application and systems software (the "backend"). The backend runs in the remote data centers and the network that are the physical underpinnings of the cloud. User interactions with the frontend result in requests to the backend. The data center brings up or shuts down hardware to accommodate the offered load of the requests from the application's users. The assumption given above puts major constraints on the possible policies to drive these mechanisms and what they can do. It is clear that a given offered load combined with uniform performance requirements will place a lower limit on how many machines need to be active, regardless of policy. Less obviously, the properties of the requests, for example the interarrival time distribution, request size distribution, and any correlations will affect the dynamics of the request stream and thus how much headroom (active hardware) the policy needs to preserve.

In this paper, we consider the prospects for relaxing this assumption. Instead of studying how the cloud or datacenter might respond better to a sacrosanct offered workload, we turn the problem around 180 degrees and consider a model in which the backend determines its desired workload characteristics and the frontend, or load balancer, enforces these characteristics. In effect, we consider shaping the user-driven workload analogously to how a packet stream might be shaped at entry to a network. We think this can be done by taking advantage of the variation in tolerance for a given level of performance that exists among individual users. We have found that such variation exists in many other contexts [14], [19], [25], [28], [40], and that its is possible to take advantage of it in those contexts [18]–[24], [35], [38], [39], [44].

We leverage a toolchain that lets us interpose on existing popular Android applications taken directly from the Google Play store. Using this toolchain, we modify a set of such applications so that their frontend/backend interaction passes through code that can selectively delay the interaction. Our additions to the applications also include mechanisms for user feedback about satisfaction with performance. This allowed us to conduct a user study where we introduced varying amounts of delay into the applications' frontend/backend interactions, and collected user feedback about their satisfaction with this added delay. A core outcome of the study is that it is

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-1265347, by a gift from Intel Corporation, and the Natural Science Foundation of China under Grant No. 61300002.

possible to introduce up to 750ms of delay without a change in user satisfaction (to within 10% with $> 95\%$ confidence) for our test applications. We also observed that user satisfaction with specific amounts of delay varied considerably.

We then consider an approach which selectively delays requests going from frontends to the backend so as to shape the arrival process at the backend as Poisson arrivals (exponentially distributed interarrival times). This is a well-known arrival process particularly suitable for leveraging classic queuing analysis in the design of scheduling systems. We simulate this using the traces acquired from the user study. These traces also allow us to determine the likely effect on per-user satisfaction of our introduced delays. From this, we can evaluate the trade-off between our backend-centric goals for introducing the delays (Poisson arrivals), and our frontend-centric goals (maintaining user satisfaction with performance). There exist trade-off points where we can make the arrival process considerably more Poisson-like while not introducing delay that leads to dissatisfaction.

The contributions of our work are:

- We introduce the concept of shaping user-driven requests originating from the frontend of a mobile application and going to the cloud backend to meet goals established by the backend. We refer to this concept as *user traffic shaping*.
- We show, via a user study involving 30 users running 5 popular Android applications on their mobile phones over a period of a week, that there is room to do user traffic shaping by introducing delays in the request stream. The tolerance for introduced delay exists across the whole subject group, and it varies across individuals.
- We describe a potential algorithm that uses this room and varying tolerance to introduce delays that shape the user request stream.
- We evaluate this algorithm, in trace-based simulation, and find that there exist trade-off points where we are able to more closely match the Poisson arrival and rate limiting goals, while not reducing user satisfaction in a significant way.

II. Related Work

Achieving an effective and responsive user experience is critical to the success of cloud applications, but at the same time there is a growing interest in making the data centers that their backends run on more efficient and sustainable. By late 2011, Google's data center operations required 260 megawatts of continuous power, with an individual search costing 0.3 watt-hours of energy [12] while a single Amazon data center operated at 15 megawatts circa 2009 [15]. The EPA estimated in 2007 that data centers consumed 61 terawatt-hours of energy in 2006, 1.5% of the total U.S. consumption, that data centers were the fastest growing electricity consumers, and that an additional 10 power plants would be required for this growth by 2011 [41]. Research studies and reports dating back to the early 1990s (e.g., [29]) have consistently shown low server utilization, with recent reports placing it in the 10–30% ballpark (e.g., [2]). This low utilization feeds into the very public "cloud factories" charge [11] that clouds are bad for the environment and unsustainable. More recent work estimates that cloud data centers consume more than 2.4% of global electricity [27], and that this consumption is expected to grow 15-20% annually [26]. In 2011 it was estimated that datacenters produced CO_2 emissions equal to 2% of global emissions [4]. Our work hopes to address these issues by *incorporating the individual user*.

Numerous approaches to making data centers more energy efficient have been proposed. The example approach of dynamically choosing the number of servers that are powered up is that has been under investigation for some time. AutoScale [9] is arguably the state of the art here, and the AutoScale paper has a detailed survey of prior work. Given an offered workload, an SLA, and the time needed to bring up/down a server, AutoScale dynamically chooses to bring up or down servers with minimal headroom (additional active servers) and minimal chance of violating the SLAs. Other examples of adapting, in an energy efficient manner, to the offered workload include dynamic voltage and frequency scaling (DVFS) for servers [6], coordinated decisions across the data center [8], [30], and consolidation within the datacenter [42]. Our work is likely to be applicable to these and other approaches in that it offers the orthogonal capability of *changing the offered workload.*

SleepServer [1] is a proxy architecture that allows a host to enter low power sleep modes more frequently and opportunistically. A SleepServer machine maintains the network presence of a host and wakes it only when required. Another work [32] simplifies the overall architecture by using a client-side agent. Such proxies are a potential venue for user-centric traffic shaping.

Traffic shaping has had its greatest success in computer networks. It originated in ATM networks [16], [33] and then expanded widely [7], [10], for example into DiffServe [3], and today is widely deployed. The user-centric traffic shaping concept is named by analogy, but an important distinction is that we focus on *shaping the users' offered workload to the cloud*, as well as *shaping*

the users' perception of the performance that workload receives.

The results of our user study may at first seem contrary to observations by Google [5], [13], Amazon [37], and others that suggest that even small increases in delay negatively impacts users who depart from the service. However, our study differs in at least two ways. First, we are considering mobile applications, not web applications. Much of the user interface of a mobile application quite smooth under delay as it is implemented on the mobile device itself, not on the backend. Second, we are soliciting the satisfaction of the user directly by prompting them, instead of indirectly by seeing if they stop using the application. We claim that users should be treated differently based on their *individual* tolerance for delay.

III. Frontend augmentation

Our user study is based on popular Android Java applications that are available only in object code form, from the Google Play store. We modify these application frontends to add the following functionality, all within the mobile phone.

1) The ability to introduce delays to the frontend's network requests. Delays are selectively introduced according to test cases loaded onto the phone.
2) Continuous measurement of the phone's environment. This includes CPU load, network characterization (RSSI), which radio is in use, and others.
3) An interface by which the user can supply feedback about performance. A user can do so at any time, but can also be prompted to do so by a test case.

The specific choice of applications, test cases, arrival process for test cases, and users is the basis of our study.

Our application augmentation framework is based on Dpartner [45] and DelayDroid. Dpartner is a general framework for decomposing a compiled Java application into its constituent parts, adding interposition, instrumentation, and other elements, repartitioning the elements differently (for example, across the client/server boundary), and then reassembling the applications. It is intended to support various kinds of experimentation with existing, binary-only, distributed applications.

DelayDroid leverages Dpartner to augment mobile Android applications. We use DelayDroid to add delaying capabilities in applications. DelayDroid effectively introduces a proxy into an application through which both high-level (e.g. HTTPWebkit) and low-level (e.g. TCPSocket) network requests are passed. The framework

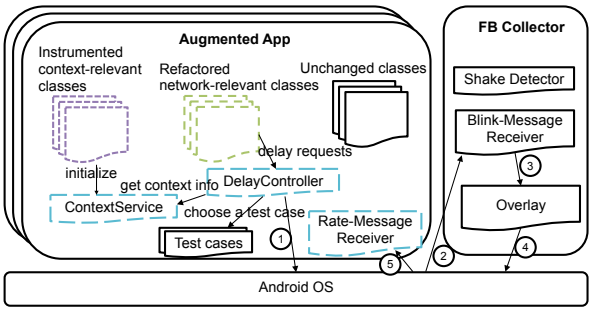

Fig. 1. Run-time architecture on the frontend.

interposes a delaying proxy on 110 networking-related functions from within 5 broad categories of the Android API: HTTPWebkit, HTTPApache, HTTPJaveNet, TCPSocket, and UDPSocket. This allows us to interpose on any network request, irrespective of the method the original app developer chose for their network communications. The proxy can delay requests through any of these interfaces, and is controllable via the test case. In the case of a 0 delay test case, the overhead of the proxy is negligible. The application binary produced as a result of DelayDroid interposition is only about 1-2% larger than the original binary.

In addition to the augmented application, our framework introduces a separate component, the Feedback Collector (FBCollector) that is responsible for coordinating augmented applications on the phone, and collecting user feedback.

Figure 1 illustrates the run-time architecture of our system when deployed in a user study. The phone contains multiple augmented applications. For each augmented application, our framework has modified or introduced four kinds of classes: refactored network-relevant classes which we interpose on to send network requests, refactored context-relevant classes which we interpose on to track context, DelayDroid run-time classes which we add in order to inject delay, and unchanged classes.

The DelayDroid run-time consists of 3 components: ContextService, DelayController and Rate-Message Receiver. The ContextService collects and provides information about the context, such as the network status. The DelayController is in charge of injecting delays into the network requests. The DelayController chooses test cases randomly from within a predefined set. Operationally, when any network request occurs, control flow is detoured through the DelayController. The DelayController then delays the request according to the test case. The Rate-Message Receiver interacts with FBCollector, and its operation is explained below.

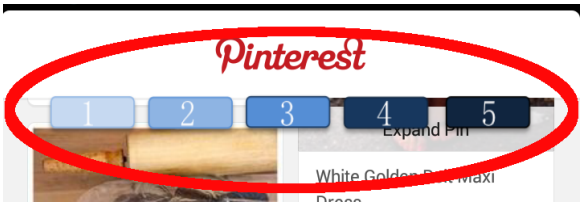

Fig. 2. Rating overlay in its active state, as hovering over Pinterest.

The FBCollector is a separate application that coordinates with augmented applications by sending and receiving Android broadcast messages. This is also the user-visible portion of the system. A Likert scale overlay of 1-5 hovers over the application, shown in Figure 2. The user can select a rating (i.e. rate their satisfaction) at any time. A rating of 1 indicates complete dissatisfaction with performance, and 5 indicates complete satisfaction with it. In addition to unprompted feedback, the user interface also supports prompted feedback, through the form of making the overlay blink a red border.

During the user study, test cases are started at random times by the DelayController. For any test case, a delay is randomly chosen and applied to all instrumented network requests sent over the duration of that test case. Once the test case finishes, the DelayController will send a message to the FBCollector ①, where it is received by the Blink Message Receiver ②, causing the overlay to blink, prompting the user ③. When a user provides feedback via the overlay, the FBCollector notifies ④ the Rate Message Receiver ⑤ component, which in turn logs the feedback and all relevant contextual information.

The log files produced by the system include a timestamp, satisfaction rating, if the rating was prompted, current test case, arrival time and duration of the request, running application, OS metrics (such as CPU load), and network metrics (WiFi or cellular, packet drops).

IV. User Study

The goal of our user study is to understand how changing performance via delaying the network requests flowing from the frontend to the backend affects user satisfaction. Simply put, how much delay can we introduce before ordinary users employing popular applications become dissatisfied? To answer this and related questions, we leveraged the framework of Section III to augment popular applications. We then designed a study in which existing users of these applications could participate on their own phones in their normal environments.

Applications: We chose five of the most popular applications from the Google Play Store, applications where we believed we would have no trouble recruiting existing users. We also wanted to test applications that had varying request rates as well as varying amounts of "computation" that would likely be done on the datacenter. For the study, we chose MapQuest, Pandora, Pinterest, WeatherBug, and Google Translate as a representative sample of popular applications.

Subjects: Our study was IRB approved[1], which allowed us to recruit participants from the broad Northwestern University community. We advertised our study by poster at several locations on campus, and also advertised by email. Our selection criteria was that the subject had to own and regularly use a mobile phone capable of running our augmented applications, and the subject had to self-identify as a regular user of at least one of our applications. We selected the first 30 respondents who met these criteria for our study.[2] As part of the study, each subject also filled out a questionnaire about their familiarity with phones, applications, etc. Each was given a $40 gift certificate at the end of the study.

Test Cases: We designed test cases—randomly selected periods of randomly selected additional delay—with an eye to inducing perceptibly different levels of performance in our applications as we ourselves perceived them. In a test case, each network request that occurs during a test case is delayed by a fixed amount. Our test cases all had a duration of one minute, and their delays were 0, 50, 250, 500, and 750 ms. Users were prompted for feedback in the middle of the test case (30 seconds in). Test cases themselves arrived randomly, with a user prompted an average of 152 times over the course of the study (one week). About 20% of prompts resulted in a response.

The only indication the subject had that a test case was running was being prompted, but the subject was also so prompted during the zero delay test case.

Methodology: The subject used his or her own personal smartphone for the duration of the study, albeit with our augmented test applications replacing the original applications. All logs were kept on the subject's smartphone, and were removed at the conclusion of the study, along with the augmented applications. The duration of the study was one week.

When a subject first arrived, we had them fill out a questionnaire designed to determine their level of knowledge and comfort with a modern mobile device, as well as collecting demographic information. During this time we installed the augmented applications.

The subject was then instructed how to use the user interface for the duration of the study. This was

[1]Northwestern IRB Project Number STU00093881.
[2]Full demographics available in Northwestern TR NU-EECS-15-02.

done with a written document that was identical for all subjects. The document stressed that our interest was in the level of performance of the applications and not in their content. It did *not* indicate to the subject how performance might change. We indicated that it was important to provide feedback *about performance* whenever the interface flashed, and we described the intent of the scale as "1 being completely dissatisfied, 5 being the most satisfied, and 3 being neutral [with/about performance]". The subject would then leave the lab, and use their phone as they normally would for one week, answering rating prompts when appropriate.

At the conclusion of the week, the subject returned to the lab, and filled out an exit questionnaire. As they filled this out, we connected to their smartphone, downloaded the study data, and removed the test applications from their phone, replacing them with the original applications. Other than our interaction with them, and the user interface, changes to their normal experience of the applications was intentionally kept to a minimum.

V. Study results

Our study produced ratings from 27, and network traces from 29 of the 30 subjects. Recall from Section III that ratings could be provided as a result of a testcase prompt or independently. In the following, we consider only the prompted responses, which correspond to test cases of any delay, including 0, on a scale of 1 to 5.

Given these constraints, our study produced 850 data points, each of which is the outcome of an intervention (the application of a test case) that resulted in a prompted response from the user. Given this number, as we decompose the results, for example by application or user, it is important that we highlight which conclusions have strong statistical support from the data. Hence, when we present p-values, we bold those that have $p < 0.05$ (95% confidence level).

To account for any anchoring effect due to user-based interpretation of satisfaction, we did our analysis based on the differences in satisfaction between delayed and undelayed application ratings for each user individually. In this way the comparisons that we make should be immune to differences in how users define their satisfaction levels.

A. Users tolerate significant added delay

If we look across all of our users and applications, we see the results of Figure 3. Here we record the average satisfaction for each level of delay, the average change in satisfaction compared to that of the zero delay level, and a p-value. In aggregate, the data points to the possibility of introducing up to 750 ms of additional delay without having a significant effect on user satisfaction. The p-values reported are for a two one-sided t-test (TOST), which measures how easily we can discard the null hypothesis that the mean satisfaction at a given delay level is *different* from the mean satisfaction at a delay of zero. In all cases, we can discard this hypothesis with at least 97% confidence. In such comparisons, the threshold of difference is also important to consider. The results in the figure are for a threshold of 0.5, or 10% of the 1–5 Likert rating scale we use. Given no other information, it appears very clear that we can add up to 750 ms of delay without changing the rating by more than 10%.

App	50ms	250ms	500ms	750ms
MapQuest	0.591	0.731	n/a	0.268
Pandora	0.133	0.127	**0.034**	0.291
Pinterest	0.131	**0.025**	0.101	0.356
WeatherBug	0.303	0.254	0.158	0.289
Translate	0.576	0.217	0.646	**0.000**

Fig. 4. TOST apps, threshold = 0.5, p-values for testing that average satisfaction is no different for the given delay value and a delay of zero. Bold values are < 0.05.

App	50ms	250ms	500ms	750ms
MapQuest	0.488	0.416	n/a	0.127
Pandora	**0.000**	**0.004**	**0.000**	**0.002**
Pinterest	**0.011**	**0.000**	**0.003**	**0.034**
WeatherBug	0.105	0.071	**0.023**	0.055
Translate	0.155	**0.003**	0.292	**0.000**

Fig. 5. TOST apps, threshold = 1.0, p-values for testing that average satisfaction is no different for the given delay value and a delay of zero. Bold values are < 0.05.

B. User tolerance for additional delay varies by application

We also considered the effects of introducing delay into individual applications, while still grouping all users together. Once again, we used TOST tests to identify where user satisfaction changed significantly compared to the no-delay case. These results are shown in Figures 4 (threshold of 0.5) and 5 (threshold of 1.0).

As one might expect, some applications experience more detrimental effects from introduced delays than others. For Pandora and Pinterest, we find that for a threshold of 1.0 (that is, one satisfaction rating) there is no statistically significant change in satisfaction caused by injecting delays ($p < 0.05$). For Google Translate more variation occurs, and for WeatherBug and MapQuest we can see that these applications are much more sensitive to additional delays. If we lower the TOST threshold to 0.5, we have less confidence that there is no change to satisfaction, although this may be

Delay [ms]	Average Satisfaction	Average Change in Satisfaction	p-value for Comparison to No Delay
No Delay	4.0773	0	n/a
50	4.1000	0.0227	**0.002**
250	4.1233	0.0460	**0.001**
500	3.9408	0.1725	**0.022**
750	4.1975	0.1202	**0.005**

Fig. 3. User satisfaction is largely unaffected by the introduction of delays of up to 750 ms into network requests made from Pandora, Pinterest, WeatherBug, and Google Translate. The p-values indicate that there was no statistically significant change in user satisfaction as request delay was added. The threshold here is 0.5 (10% of the rating scale).

App	0ms	50ms	250ms	500ms	750ms
MapQuest	0.91	2.25	1.39	0.00	0.25
Pandora	1.10	0.96	1.22	1.08	0.88
Pinterest	1.08	1.49	0.95	0.89	2.06
WeatherBug	0.88	1.96	1.65	1.41	1.64
Translate	0.14	1.69	0.54	1.86	0.07

Fig. 6. User satisfaction varies considerably across users. Rating variance across users for each combination of application and delay.

simply due to the relatively small amount of data we were able to collect at this granularity.

C. User tolerance for additional delay varies across users

Figure 6 shows variance of user-perceived satisfaction for each of the delay levels. Recall that our test cases are randomly chosen and arrive at random times. What we are resolving here is that a given level of delay is likely to affect different users differently, and also the same user differently across time.

Figure 7 illustrates this further. Here, for each application, we computed the variance of satisfaction for each individual user, aggregating over the different delays (which have equal probability). We then present each user's satisfaction variance, sorted by variance. We see that, for example, user 11 has the highest variance for MapQuest and Pandora, but is second or third in the rankings for the other applications.

Since the tolerance for additional delay varies across applications, and users, it seems natural that a real system should try to identify the more delay-tolerant users as particular opportunities for improving the request process.

VI. User traffic shaping

We now consider shaping user traffic while staying within acceptable boundaries. Given the results of the study, we selected 750ms as the amount of added delay that a user is willing to tolerate for most applications. We implemented a queue-based simulator which takes in

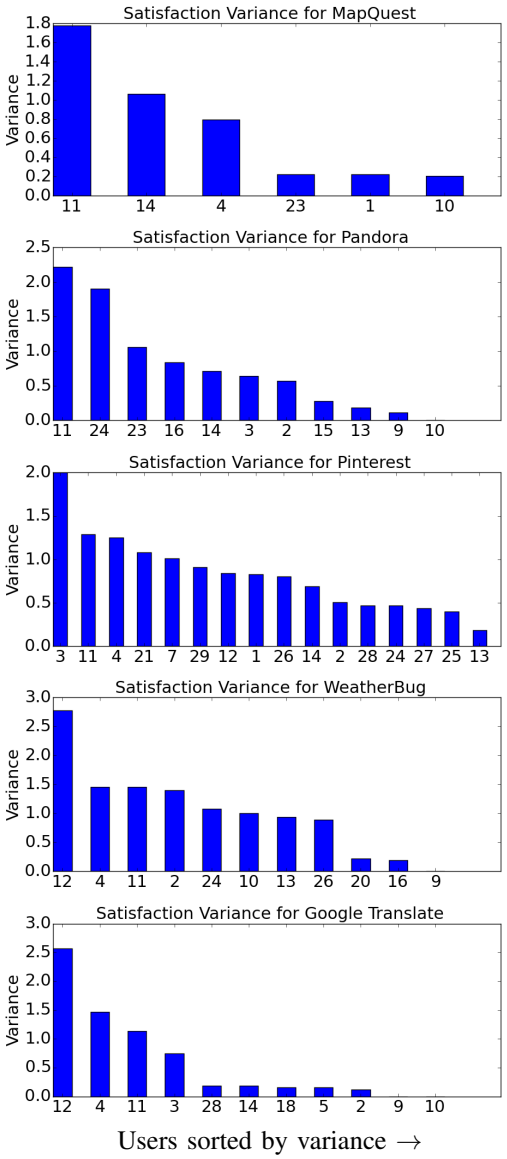

Fig. 7. User satisfaction varies considerably across users. Variance of satisfaction for each user (horizontal axis), ranked by variance.

user request arrivals, and delays each incoming request according to the specified shaping method. In this way, we are able to simulate shaping that might occur at any point along the request path, and stay agnostic to any final system, but rather explore the potential of shaping.

A. Algorithm

Our algorithm attempts to shape request arrivals such that they appear to have been drawn from a Poisson distribution. Poisson arrivals are a desirable property both in terms of easing the analysis of a system from a queuing theory perspective, and because they are less prone to the Noah Effect [43] in which traffic bursts aggregate across multiple timescales. It is important to note that other forms of shaping are certainly possible—our algorithm is intended as a demonstration of the concept of using the leeway provided by user delay tolerance to do shaping of some form.

Our algorithm has three main components: a delay generator, and two rate estimators. The algorithm is driven by request arrivals. First, the arrival rate is estimated via an exponentially weighted moving average (EWMA). This rate is then compared to the desired rate, and used to generate the delay. Each delay is picked from a exponential random distribution, since a Poisson process has interarrivals that are exponentially distributed. Because the rate of the input arrivals can fluctuate significantly, we employ a PI controller to continuously adjust the mean of the exponential distribution to keep the output rate close to the desired rate. As the system emits the delayed requests a second EWMA is used to estimate the output rate, which in turn is used to calculate the error for the PI controller. The whole algorithm is described in Figure 8.

B. Evaluation

We evaluate the algorithm in simulation by feeding it with traces that we collected during the user study. The traces collected contained tuples of the form

$$\{time, appName, user, totalTime, delay, testCase\}$$

Recall that each test case ran for one minute, and during this time *all* network requests were delayed. For this reason, we have considerably more requests than test cases, and for our simulation we use all 140,401 collected records. For each simulation, the trace was segmented into "sessions", where a session was defined to be any section of requests where no interarrival was greater than 10 seconds. In other words, we consider bursts of arrivals that correspond to application activity that is typically driven by the user.

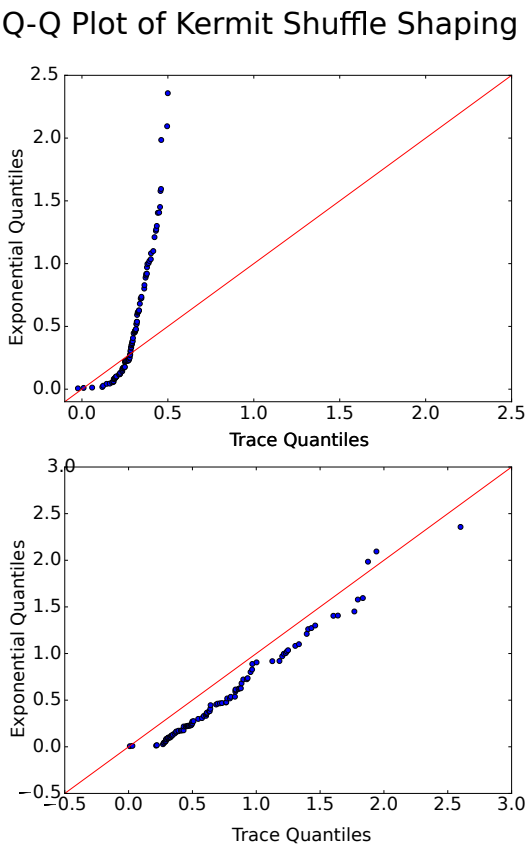

Fig. 9. Example of the effect of the algorithm on shaping an input. The input (top graph) has been improved by$\Delta R^2 = 0.1822$, leading to the much more Poisson-like output (bottom graph).

For each session, we compare the original and shaped session using quantile-quantile (or Q-Q) plots, an example of which is shown in Figure 9. The desired shape in each plot is the solid diagonal line. If the points were to line up exactly on that line, we could confidently say that the two data sets plotted were drawn from the same distribution. We fit a line to each graph using a least squares fit and quantify the fit with an R^2 value. We can then compare the original and shaped graphs via the difference in R^2 values, which we refer to as ΔR^2. In the figure, the ΔR^2 is 0.1822. R^2 ranges from -1 to 1, so this represents a fairly significant shape in changing.

It is important to note that evaluations were done across sessions, in order to capture local shaping (as sessions may have differing rates), however the algorithm has no awareness of such structure, and simply operates on the rate provided via the estimators, thus needing no *a priori* knowledge.

1: **procedure** SHUFFLE($arrival, inter_d$)
2: $inter_u \leftarrow$ EWMA(input) ▷ Estimate interarrivals of input using EWMA
3: $inter_s \leftarrow$ EWMA(output) ▷ Estimate interarrivals of output using EWMA

4: $m \leftarrow min(inter_u, inter_d)$ ▷ Limit the shaping interarrival
5: $err \leftarrow inter_s - inter_d$ ▷ how much is the shape off
6: $nudge \leftarrow PI(err)$ ▷ nudge with PI controller
7: $m \leftarrow max(0.001, m + nudge)$ ▷ ensure nudging is not negative
8: $delay \leftarrow exp(m)$
9: $arrival \mathrel{+}= delay$
10: **end procedure**

Fig. 8. User Traffic Shaping Algorithm.

For each of the evaluations, the shaping was run 30 times on each trace, and the results averaged in order to estimate the ensemble average behavior. In addition, for each session the Q-Q comparison is run 30 times, as the distribution being compared to in the plot is itself randomly generated each time. For the PI controller and EWMA estimator, we conducted multiple runs and settled on constant values of $K_p = 0.9$, $K_i = 0.5$, $\alpha = 0.8$ as optimal.

We additionally want to see how the algorithm performs across varying loads. To do so, we set the desired rate according to a variable system load, where load is defined as $load_{sys} = \frac{inter_{desired}}{inter_{trace}}$, and $inter_{trace}$ was defined as the average of the trace. It is important to note that this information is not needed during shaping, it simply provides us with a method of evaluating performance across various loads.

Figure 10 presents the results of running the algorithm on an individual user. For each load factor, we report the average ΔR^2, the average delay introduced to each request, as well as the 95^{th} and 90^{th} percentile of the delay introduced. For this evaluation, we consider the point where the 95^{th} percentile delay grows beyond the acceptable tolerance envelope as the limit for shaping. We can see from the figure that for User 1, the limit occurs at a load of 0.5, with $\Delta R^2 = 0.0665$. We ran the same analysis on each user from the study, and the results are enumerated in Figure 12. We can see that the ability to shape and the load at which we can shape traffic without annoyance varies quite a bit between users, which indicates that shaping at the user level will produce the most beneficial results overall.

To evaluate the performance of the algorithm on an entire user base, we combined all of the user traces from the study into one aggregate stream and fed this into the simulator. The results are shown in Figure 11. We see that the limit of shaping for aggregate users occurs at a lower load of 0.1, with a ΔR^2 of 0.0053, and 95^{th} percentile delay of 0.4518.

Load	ΔR^2	Avg Delay	95 %ile	90 %ile
0.1	0.0102	0.0389	0.1479	0.1054
0.2	0.0236	0.1014	0.2963	0.2118
0.3	0.0408	0.0947	0.4448	0.3186
0.4	0.0537	0.2548	0.5925	0.4239
0.5	**0.0665**	**0.2206**	**0.7403**	**0.5324**
0.6	0.0771	0.2458	0.8949	0.6394
0.7	0.0892	0.4656	1.0505	0.7472
0.8	0.0990	0.2271	1.1949	0.8515
0.9	0.1071	0.3547	1.3407	0.9604
1.0	*0.1204*	*0.4582*	*1.4912*	*1.0669*

Fig. 10. Shaping of trace of User 1. Ability of the algorithm to shape an individual user's traffic increases with load. For this user, the algorithm can produce a $\Delta R^2 = 0.0665$ while staying within the delay tolerance envelope supported by the user study 95% of the time (bold). It produces $\Delta R^2 = 0.1204$ by staying in envelope on average (italic).

Load	ΔR^2	Avg Delay	95 %ile	90 %ile
0.1	**0.0053**	**0.1316**	**0.4518**	**0.3246**
0.2	0.0101	0.2632	0.9048	0.6494
0.3	0.0170	0.3948	1.3543	0.9654
0.4	0.0223	0.5265	1.8060	1.2982
0.5	*0.0299*	*0.6581*	*2.2828*	*1.6272*
0.6	0.0337	0.7898	2.7273	1.9482
0.7	0.0386	0.9214	3.1769	2.2785
0.8	0.0462	1.0531	3.6138	2.6083
0.9	0.0509	1.1847	4.0911	2.9127
1.0	0.0552	1.3163	4.5210	3.2436

Fig. 11. Ability of the algorithm to shape aggregate traffic increases with load. Bold indicates staying within delay tolerance 95% of the time. Italic indicates staying within delay tolerance on average.

VII. CONCLUSIONS

We have considered the prospects for shaping the interactions between the frontends and cloud/datacenter

User	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ΔR^2	0.067	0.012	0.007	0.017	0.000	0.012	0.019	0.005	0.008	0.005	0.011	0.012	0.006	0.005	0.008
Load	0.5	0.2	0.2	0.2	0.1	0.5	0.2	0.1	0.1	0.1	0.2	0.3	0.1	0.1	0.1
User	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
ΔR^2	0.006	0.018	0.012	0.003	0.000	0.056	0.009	0.009	0.009	0.033	0.003	0.039	0.009	0.024	n/a
Load	0.1	0.3	0.1	0.1	0.1	0.4	0.1	0.1	0.2	0.3	0.5	0.3	0.2	0.2	n/a

Fig. 12. Shaping results for each individual user, showing the maximum load and ΔR^2 achievable while staying within the 750 ms delay tolerance envelope supported by the user study 95% of the time.

backends of mobile applications. In particular, we considered delaying requests produced by the frontend and sent to the backend. Introducing such delays could be the mechanism for shaping the arrival process of the requests at the backend. Of course, too many delays or delays that are too large could irritate users.

There seems to be considerable opportunity to introduce delays without affecting user satisfaction. We developed a system that augments Android mobile applications with a delay component, and applied it to a range of popular applications. We then conducted an "in the wild" user study in which users employed our augmented applications instead of the ones they would normally use. The augmented applications would randomly add delays and accept user feedback about satisfaction. Analysis of the study data shows, among other things, the surprising result that delays of up to 750 ms can be introduced most of the time for most users without a major change in their measured satisfaction.

As a proof of concept of user traffic shaping, we developed an algorithm that introduces delay to requests in a controlled manner to attempt to make their arrival process at the backend have exponential interarrival times (Poisson arrivals). We simulated the algorithm using the trace data from the study. While keeping the introduced delays within the tolerance determined by the study, the algorithm is able to appreciably affect the arrival process, pushing it significantly closer to Poisson arrivals.

We are currently expanding these results along three lines. First, we are working on user interfaces that provide continuous individual user satisfaction feedback with little to no cognitive effort on the part of the user (e.g. [17], [31], [34], [36]). The more successful such a technique is, the more we would be able to take advantage of the variation in delay tolerance among users. Second, we are trying to expand the delay tolerance of users by tying the concept to environmentalism, sustainability, and peer pressure. We are currently testing whether a user interface that allows the user to set a delay tolerance and provides feedback about the environmental and/or sustainability impact of their setting, as well as how the user's peer group has set it. Finally, we are considering other ways in which to use delay tolerance in shaping user traffic.

REFERENCES

[1] AGARWAL, Y., SAVAGE, S., AND GUPTA, R. Sleepserver: A software-only approach for reducing the energy consumption of pcs within enterprise environments. In *USENIX Annual Technical Conference* (2010).

[2] BARROSO, L., AND HOELZLE, U. The case for energy-proportionate computing. *IEEE Computer 40*, 12 (December 2007), 33–37.

[3] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An architecture for differentiated services. Tech. Rep. RFC 2475, Network Working Group, December 1998.

[4] BROWN, R., ET AL. Report to congress on server and data center energy efficiency: Public law 109-431. *Lawrence Berkeley National Laboratory* (2008).

[5] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (February 2013), 74–80.

[6] DENG, Q., MEISNER, D., BHATTACHARJEE, A., WENISCH, T., AND BIANCHINI, R. Coscale: Coordinating cpu and memory systems dvfs in server systems. In *Proceedings of the 45th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO 2012)* (August 2012).

[7] ELWALID, A., AND MITRA, D. Traffic shaping at a network node: Theory, optimal design, and admission control. In *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 97)* (April 1997).

[8] GANDHI, A., HARCHOL-BALTER, M., DAS, R., AND LEFURGY, C. Optimal power allocation in server farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2009)* (June 2009).

[9] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems 30*, 4 (November 2012), 1–26.

[10] GEORGIADIS, L., GUERIN, R., PERIS, V., AND SIVARAJAN, K. Efficient network qos provisioning based n per-node traffic shaping. *IEEE/ACM Transactions on Networking 4*, 4 (1996), 482–501.

[11] GLANZ, J. The cloud factories: Power pollution and the internet. New York Times, September 22, 2012.

[12] GLANZ, J. Google details, and defends, its use of electricity. New York Times, September 8, 2011.

[13] Speed matters: Google research blog. http://googleresearch.blogspot.com/2009/06/speed-matters.html. Accessed: 2016-05-21.

[14] GUPTA, A., LIN, B., AND DINDA, P. A. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)* (June 2004).

[15] HAMILTON, J. Where does the power go in high scale data centers? Keynote of ACM SIGMETRICS 2009.

[16] JAIN, R. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems 28*, 13 (1996), 1723–1738.

[17] KAPOOR, A., BURLESON, W., AND PICARD, R. W. Automatic prediction of frustration. *Intl. Journal of Human-Computer Studies* (August 2007), 724–736.

[18] LANGE, J., DINDA, P., AND ROSSOFF, S. Experiences with client-based speculative remote display. In *Proceedings of the USENIX Annual Technical Conference (USENIX)* (2008).

[19] LANGE, J. R., MILLER, J. S., AND DINDA, P. A. Emnet: Satisfying the individual user through empathic home networks. In *Proceedings of the 29th IEEE Conference on Computer Communications (INFOCOM)* (March 2010).

[20] LIN, B., AND DINDA, P. User-driven scheduling of interactive virtual machines. In *Proceedings of the Fifth International Workshop on Grid Computing* (November 2004).

[21] LIN, B., AND DINDA, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing)* (November 2005).

[22] LIN, B., AND DINDA, P. Towards scheduling virtual machines based on direct user input. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (2006).

[23] LIN, B., MALLIK, A., DINDA, P., MEMIK, G., AND DICK, R. User- and process-driven dynamic voltage and frequency scaling. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (April 2009).

[24] MALLIK, A., COSGROVE, J., DICK, R., MEMIK, G., AND DINDA, P. Picsel: Mearuing user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th International Conference in Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008).

[25] MALLIK, A., LIN, B., MEMIK, G., DINDA, P., AND MEMIK, G. User-driven frequency scaling. *IEEE Computer Architecture Letters 5*, 2 (2006).

[26] MASANET, E., SHEHABI, A., AND KOOMEY, J. Characteristics of low-carbon data centres. *Nature Climate Change 3*, 7 (2013), 627–630.

[27] MEIJER, G. Cooling energy-hungry data centers. *Science 328*, 5976 (2010), 318–319.

[28] MILLER, J. S., MONDAL, A., POTHARAJU, R., DINDA, P. A., AND KUZMANOVIC, A. Understanding end-user perception of network problems. In *Proceedings of the SIGCOMM Workshop on Measurements Up the STack (W-MUST 2011)* (August 2011). Extended version available as Northwestern University Technical Report NWU-EECS-10-04.

[29] MUTKA, M. W., AND LIVNY, M. The available capacity of a privately owned workstation environment. *Performance Evaluation 12*, 4 (July 1991), 269–284.

[30] NATHUJI, R., AND SCHWAN, K. Virtualpower: Coordinated power management in virtualized enterprise systems. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP 2007)* (October 2007).

[31] PICARD, R. W. *Affective Computing*. MIT Press, Cambridge, 1997.

[32] REICH, J., GORACZKO, M., AND KANSAL, A. Sleepless in seattle no longer. In *USENIX Annual Technical Conference* (2010).

[33] REXFORD, J., BONOMI, F., GREENBERG, A., AND WONG, A. Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches. *IEEE Journal on Selected Areas in Communications 15*, 5 (1997), 938–950.

[34] SCHUCHHARDT, M., SCHOLBROCK, B., PAMUKSUZ, U., MEMIK, G., DINDA, P., AND DICK, R. Understanding the impact of laptop power saving options on user satisfaction using physiological sensors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED 2012)* (July-August 2012).

[35] SHYE, A., B. OZISIKYILMAZ, A. M., MEMIK, G., DINDA, P., DICK, R., AND CHOUDHARY, A. Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)* (June 2008).

[36] SHYE, A., PAN, Y., SCHOLBROCK, B., MILLER, J. S., MEMIK, G., DINDA, P., AND DICK, R. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Lake Como, Italy, Nov. 2008 [Nominated for Best Paper Award]).

[37] SRINIVASAN, S. V. Lessons learned in building real-time big data systems. In *Proceedings of the 20th International Conference on Management of Data* (Mumbai, India, India, 2014), COMAD '14, Computer Society of India, pp. 5–6.

[38] SWIECH, M., AND DINDA, P. Making javascript better by making it even slower. In *Proceedings of the 21st IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)* (August 2013).

[39] TARZIA, S., DICK, R., DINDA, P., AND MEMIK, G. Sonar-based measurement of user presence and attention. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp 2009)* (September 2009). Poster also appeared in Usenix 2009.

[40] TARZIA, S., DINDA, P., DICK, R., AND MEMIK, G. Display power management policies in practice. In *Proceedings of the 7th IEEE International Conference on Autonomic Computing (ICAC 2010).* (June 2010).

[41] UNITED STATES ENVIRONMENTAL PROTECTION AGENCY. Report to congress on server and data center energy efficiency public law 109-431, August 2007.

[42] VERMA, A., AND DASGUPTA, G. Server workload analysis for power minimization using consolidation. In *USENIX Annual Technical Conference* (2009).

[43] WILLINGER, W., TAQQU, M. S., SHERMAN, R., AND WILSON, D. V. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. In *Proceedings of ACM SIGCOMM '95* (1995), pp. 100–113.

[44] YANG, L., DICK, R., MEMIK, G., AND DINDA, P. Happe: Human and application driven frequency scaling for processor power efficiency. *IEEE Transactions on Mobile Computing (TMC) 12*, 8 (August 2013), 1546–1557. Selected as a Spotlight Paper.

[45] ZHANG, Y., HUANG, G., LIU, X., ZHANG, W., MEI, H., AND YANG, S. Refactoring android java code for on-demand computational offloading. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA 2012)* (October 2012).