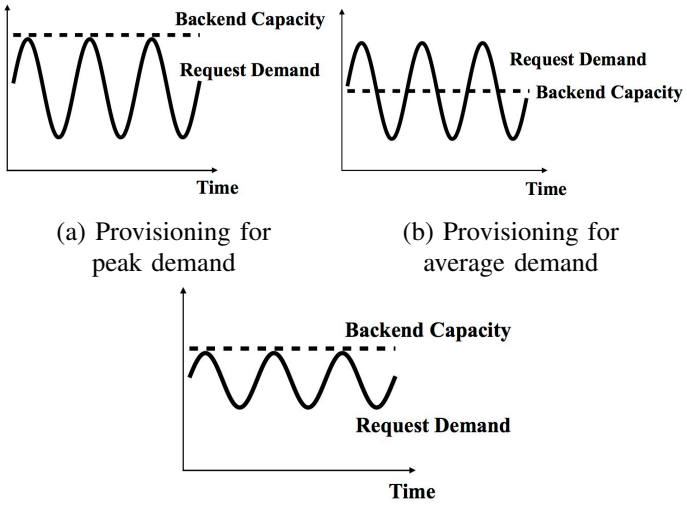
Clockwork: Scheduling Cloud Requests in Mobile Applications

Yanjiao Chen[‡], Zixuan Yu[§], Baochun Li[§]

[§]State Key Lab of Software Engineering, School of Computer, Wuhan University, China
§Department of Electrical and Computer Engineering, University of Toronto
Email: ‡chenyanjiao@whu.edu.cn, §{zixuan.yu, bli}@ece.toronto.edu

***Abstract*—It is essential for mobile application developers to manage backend resources to serve dynamic user requests from the frontend. For a typical mobile application, the rate at which the user requests arrive at the backend fluctuates dramatically. However, it is difficult or expensive to frequently adjust the capacity of the backend to meet the request demand. In this paper, we present *Clockwork*, a third-party cloud service, which smooths the demand profile by redistributing delay-tolerant requests and prioritizing delay-sensitive requests, so that sufficient capacity can be provided with reduced cost and wastage. To begin with, Clockwork plans the optimal backend capacity on a relatively long timescale based on future demand estimated by machine learning algorithms. We discuss pros and cons of various simple machine learning algorithms and advanced deep learning algorithms, in terms of their prediction accuracy and training time. Then, Clockwork schedules user requests on a shorter timescale through a fair and Pareto-optimal rate allocation. We implemented a fully-functional prototype of Clockwork on cloud servers and user mobile devices. The experimental results show that Clockwork can effectively help developers cut cost, as well as improve the backend utilization.**



(a) Provisioning for peak demand

(b) Provisioning for average demand

(c) Clockwork

Fig. 1: Backend provisioning comparison.

I. Introduction

The market for mobile applications is booming. In 2015, global application downloads are estimated to be 179 billion [1], generating $45 billion revenues [2] and involving 5.5 million developers [3]. To compete for market share and profits, application developers are motivated to provide quality service with reduced cost.

Backend provisioning is one of the fundamental concerns for developers, as all user requests from the frontend of an application need to be handled by the backend. Developers can either use Mobile-backend-as-a-Service (MBaaS) or build their own backends on cloud platforms (such as Amazon EC2). In the former case, developers can directly enjoy backend services offered by MBaaS providers, who charge fees based on the number of requests sent to the MBaaS backend. In the latter case, developers can construct their backends using instances, and pay for their usage.

One of the biggest challenges facing backend provisioning is a mismatch between timescales of request demand variance and backend resource availability. The rate at which user requests arrive at the backend varies from second to second, but developers can neither change the MBaaS service plan nor adjust instance configurations at such a fine granularity of time. Though Amazon EC2 also provides auto-scaling mechanisms, it requires developers to determine how many instances to add or reduce, and it takes time to boot up or shut down computers. Rather than trying to let the backend capacity closely *track* the demand profile, which is difficult or even impossible to realize, we explore a new way of backend provisioning. We *smooth* the demand profile by redistributing delay-tolerant requests and prioritizing delay-sensitive requests, which potentially lightens the burdens on the backend and saves money for developers.

To achieve such a goal, we present Clockwork, a third-party cloud service designed to help developers manage backend provisioning. Clockwork features a two-tier architecture: backend capacity planning on a long timescale, and rate-based request scheduling on a short timescale. Both tiers are indispensable in helping developers maintain a sufficient and cost-effective backend. As shown in Fig. 1(a), if the backend capacity caters to the peak demand to avoid any performance degradation, there will be a considerable wastage of resources and money. Nevertheless, if the backend capacity falls short of the peak demand, as shown in Fig. 1(b), some requests will have to be dropped or experience unacceptably long processing times. In comparison, Clockwork schedules requests according

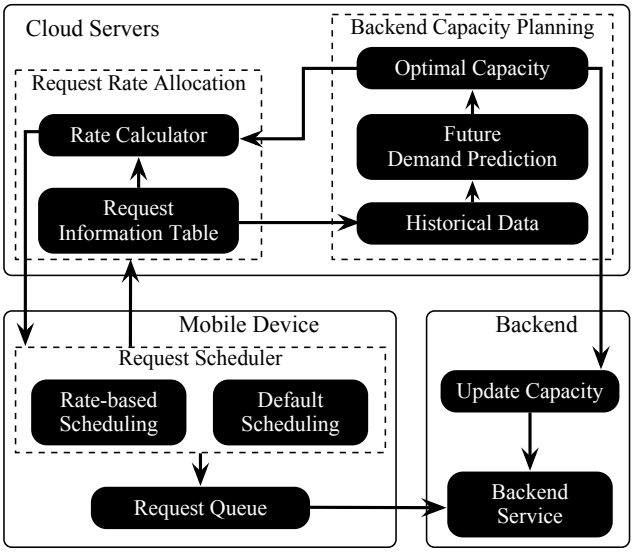
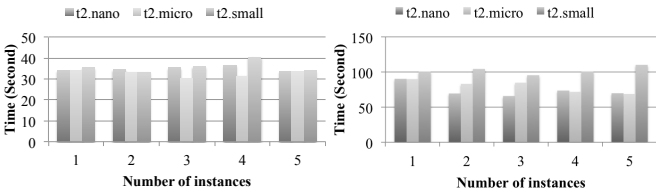


Fig. 2: Architecture of Clockwork.



(a) Launch instances (b) Terminate instances

Fig. 3: Time for launching or terminate instances

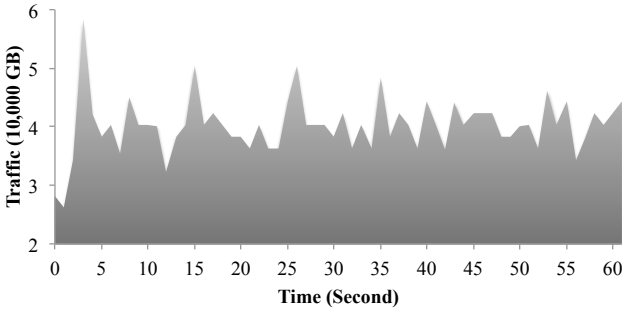


Fig. 4: Network traffic fluctuations.

to delay tolerance, as shown in Fig. 1(c), thus flattening the request demand and slashing the required backend capacity.

There are several challenges facing the implementation of Clockwork. To plan for the long-term backend capacity, it is crucial to have an accurate prediction of the future request demand. There is a wide variety of machine learning algorithms for data prediction, each of which has their advantages and disadvantages. We compare the prediction accuracy and training time of four machine learning algorithms (Section III-B). It is shown that deep learning algorithms, including convolutional neural networks and deep belief networks, yield better prediction results but require a much longer time to train than simpler machine learning algorithms, such as logistic regression and multilayer perceptron. With the estimated future demand, the Clockwork derives the optimal backend capacity that minimizes developer's cost, while conforming to the constraint on request delay (Section III-A).

Given the long-term capacity, to ensure that the backend is not overwhelmed by instant request demand, we adopt a rate-based resource allocation strategy (Section IV). Clockwork centrally assigns rates to users based on the quantity and delay tolerance of their requests, then individual users autonomously schedule their own requests. We build a network utility maximization (NUM) model to decide the rate allocation, which is proved to be fair and Pareto-optimal.

We have implemented the server-side of Clockwork using the Amazon Web Service (AWS), and the client-side of Clockwork on iOS-based mobile devices. We present the implementation details (Section V) and evaluate the performance of Clockwork by a small-scale pilot experiment and a large-scale simulation (Section VI). The results show that Clockwork can help developers substantially cut down the backend cost, as well as improve the backend utilization.

II. Clockwork: Motivations and Architecture

Mobile application development comprises two major parts: frontend design and backend support. Simply put, the frontend of an application is what users can see and experience on their mobile devices. The frontend differentiates one application from another, thus it is vastly diversified for different applications. The backend of an application supports its frontend functionalities. When a user interacts with the application through the frontend, the corresponding operations are accommodated via API requests to the backend. Typical API requests of a mobile application include queries, saves, and logins, amongst other kinds of operations invoked by users to perform a task. Some requests are urgent, while some others are delay-tolerant. Take social messaging applications as an instance. The request of *sending a message* is urgent and should be served straightaway, while the request of *updating the user profile* can wait to be served later.

In this paper, we focus on the backend of the application development, whose cost model is more common across all applications. Mobile application developers can use the ready-made and customizable backend provided by MBaaS or build their own backends on cloud platforms such as Amazon EC2.

MBaaS providers, such as Appcelerator, appery.io, and Kumulos, charge developers based on the number of API requests sent by users to the MBaaS backend. More specifically, the developer should subscribe to a service plan, usually on an hourly or monthly basis. The service plan specifies the limit on the maximum number of requests that can be sent within a unit time, e.g., one minute. If such a limit is hit, extra requests will be discarded with error messages to users. A service plan with a higher request limit will cost more.

If a developer utilizes Amazon EC2 to set up the backend, her cost depends on the number and the type of instances that

are launched. For example, a t2.nano instance with 1 CPU and 0.5GB memory costs $\$0.0059$ per hour, and a t2.small instance with 1 CPUs and 2GB memory costs $\$0.023$ per hour. The payment is prorated by the hour. This indicates that once a developer turns on a t2.large instance, she will immediately be charged $\$0.094$, even if she doesn't plan to use the instance for the entire hour.

Fig. 4 illustrates the world network traffic by the second[1], from which we can infer that there is a dramatic fluctuation in mobile backend traffic. The demand peak may be much higher than the average, but is only momentary. In contrast, developers can only change MBaaS service plan on an hourly basis. As for cloud services, Fig. 3 shows that it takes around one minute to launch or terminate an instance, and a new instance is charged for a full hour even if it is only needed for a short period of time.

To address the above challenge, the architecture of Clockwork consists of two major modules: backend capacity planning and request rate allocation, as shown in Fig. 2. Backend capacity planning is conducted on a relatively long timescale, e.g., per hour, since it is either difficult or costly to frequently change the backend capacity. Request rate allocation is performed on a short timescale, e.g., per minute, based on the dynamics of request demand.

III. Backend Capacity Planning

A. Backend Capacity Optimization

Without loss of generality, we define the backend capacity as the maximum number of requests that can be served per minute, which is directly linked to the cost of the backend (e.g., to pay for the MBaaS service plan or the instances on Amazon EC2). We assume that the developer is able to adjust the backend capacity on an hourly basis. Let N denote the required backend capacity, to be determined by the optimization model. The developer classifies all possible requests of the application into K types, according to their delay tolerance. For example, there can be $K = 3$ types of requests: urgent, medium, and delay-tolerant. An hour consists of $i = 1, 2, ..., 60$ minutes, during which the backend capacity is fixed. Let n_i^k denote the estimated number of type k requests that will arrive at the i-th minute in the next hour. In this section, we assume that $n_i^k, i \in [1, 60], k \in [1, K]$ are available as inputs to the optimization model, and we will discuss how to predict n_i^k in details in the next section. Without Clockwork, to guarantee the performance of the backend, the developer has to make sure that the backend capacity is greater than the peak demand, i.e., $N \geq \max_i \sum_k n_i^k$. We will show that such over-provisioning is unnecessary with the request scheduling mechanism of Clockwork.

We make the simplified assumption that all requests generated in a specific hour will be served within that hour. In other words, during the current hour, the backend will neither serve requests from the previous hour, nor put off requests to the next hour. At the i-th minute, let $\delta_{ij}^k \in [0, 1]$ denote the proportion of the n_i^k requests that will be postponed to the j-th minute. We have $i \leq j \leq 60$, and δ_{ii}^k is the proportion of requests that are not delayed. All requests to be served at the j-th minute, denoted by N_j, include those deferred from the previous minutes to the j-th minute, and those generated and instantly served in the j-th minute, i.e., $N_j = \sum_{i=1}^{j-1} \sum_{k=1}^{K} \delta_{ij}^k n_i^k + \sum_{k=1}^{K} \delta_{jj}^k n_j^k = \sum_{i=1}^{j} \sum_{k=1}^{K} \delta_{ij}^k n_i^k$. To guarantee the performance of the backend, its capacity should be larger than the peak demand after request scheduling, i.e., $N \geq \max_{j \in [1,60]} N_j$.

Delaying requests will affect the user experience, thus the developer needs to have a control over how many and how long a certain type of requests can be delayed. Clockwork allows the developer to set the upper-bound of δ_{ij}^k as $\overline{\delta_{ij}^k}$, which depends on the request type k and the length of delay $j-i$. For instance, in a social messaging application, if *sending a message* is regarded as the most urgent type of requests and should not be delayed, the developer can simply set $\forall j > i, \overline{\delta_{ij}^{\text{send a message}}} = 0$; if no request should be delayed for more than half an hour, the developer can simply stipulate that $\forall k, \overline{\delta_{ij}^k} = 0, \text{if } j - i > 30$.

As the backend cost will monotonically increase with the backend capacity, we set the objective of Clockwork as to minimize the required backend capacity, without violating the constraint on δ_{ij}^k designated by the developer.

$$\min_{\delta_{ij}^k} \ N, \tag{1}$$

$$\text{subject to } N \geq \max_{j \in [1,60]} N_j, \tag{2}$$

$$N_j = \sum_{i=1}^{j} \sum_{k=1}^{K} \delta_{ij}^k n_i^k, \forall j, \tag{3}$$

$$\sum_{j=i}^{60} \delta_{ij}^k = 1, \forall i, \forall k, \tag{4}$$

$$0 \leq \delta_{ij}^k \leq \overline{\delta_{ij}^k}, \forall i, \forall j, \forall k. \tag{5}$$

Constraint (2) guarantees that the backend capacity is large enough to meet the peak request demand. Constraint (3) shows the number of requests per minute after scheduling. Constraint (4) ensures that all the requests initiated in the i-th minute are served, either instantly or in later minutes. The objective function is minimized through variables δ_{ij}^k, and we can get the optimal backend capacity as $N^* = \max_{j \in [0,60]} N_j^*$, in which $N_j^* = \sum_{i=1}^{j} \sum_{k=1}^{K} {\delta_{ij}^k}^* n_i^k$. The optimization problem (1) is a linear programming problem, and can be solved by existing algorithms such as Simplex and Interior point algorithms. Fig. 5 gives an example of how the backend capacity planning of Clockwork works. It is shown that the demand profile is smoothed as Clockwork schedules requests to *cut* the peak and *fill* the valley.

B. Future Demand Prediction

The optimization problem (1) requires the input of n_i^k, the expected number of type k requests generated at the

[1]http://www.internetlivestats.com/

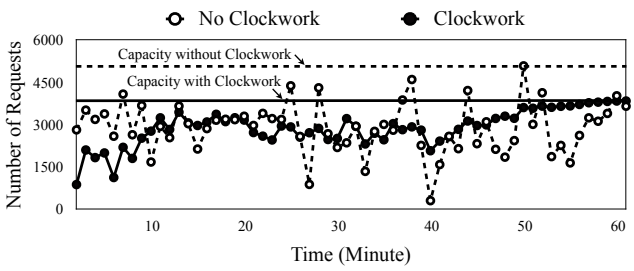


Fig. 5: Request demand smoothing by Clockwork.

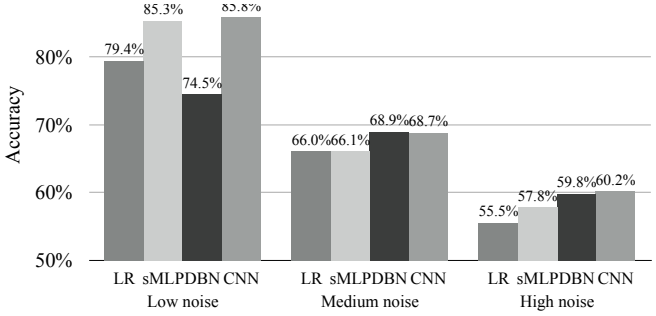


Fig. 6: Prediction accuracy.

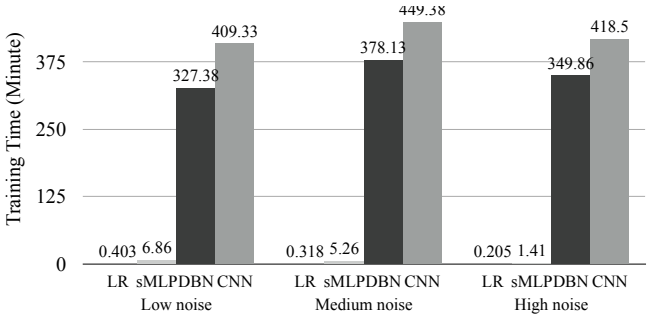


Fig. 7: Training time.

i-th minute of the next hour, which can be learned from historical data using machine learning algorithms. We focus on four widely-used algorithms: logistic regression (LR), single-hidden-layer multilayer perceptron (sMLP), deep belief networks (DBN), and convolutional neural networks (CNN). Logistic regression and sMLP are simple machine learning models, while DBN and CNN are deep learning models. For simplicity, in this paper, we use these models for classification, while they can easily be adapted for regression. For instance, $n_i^k \in (1, 100]$ and $n_i^k \in (100, 200]$ can be defined as class 1 and class 2, respectively.

Historical data are collected in the form of the number of requests generated in each minute. The input vector $\boldsymbol{x}$ to the machine learning model should contain the best predictors for n_i^k. Two potential factors need to be taken into consideration. One is *temporal-proximity*, i.e., the most recent demand indicates the near future. The other is *diurnal* effect, i.e., demands at the same time of each day have a similar trend. Therefore, we use the demand of the previous two hours and the demand at the same time of the previous two days as the input vector to predict n_i^k, i.e., $\boldsymbol{x} = (n_{i-180}^k, \cdots, n_{i-61}^k, n_{i-1440-180}^k, \cdots, n_{i-1440+179}^k, n_{i-1440*2-180}^k, \cdots n_{i-1440*2+179}^k)$, and $Y = n_i^k$. Furthermore, we normalize entries in the input vectors as $\boldsymbol{x} \rightarrow \boldsymbol{x}/\max\{n_i^k\}$, and discretize the output into 10 levels for classification.

We use synthetic datasets to evaluate the performance of the four machine learning models as follows. Assume that there are 100 users, each generating requests according to a Poisson process. Without loss of generality, we only consider one type of requests. We first generate a series of values with a diurnal pattern (low demand at working time, and high demand at leisure time), to represent the request arrival rate at different time of a day for an average user. Then, we compute the arrival rate for each individual user by adding noise to the arrival rate of the average user. Aggregating the number of requests from all users at each minute yields the demand profile. We run the four machine learning algorithms on a MacBook Pro laptop with 2.9 GHz Intel CPU and 8 GB memory.

As shown in Fig. 6, deep learning models outperform simple models when the dataset is more noisy, since deep learning models are more powerful in discovering the intricate relationship between the input and the output. Unexpectedly, at the low noise level, DBN gives the worst prediction result, even inferior to the logistic regression model. One possible reason is that the complicated structure of DBN leads to the over-fitting problem. In other words, the model exaggerates the noise in the training data, instead of learning the general trend, therefore gives poor predictions when applied to the testing data.

As shown in Fig. 7, the training time of deep learning models is far higher than that of simple models, which is not surprising since the deep learning models contain far more parameters to be learned. The training time of DBN and CNN also depends on the choice of the number of layers and neurons in each layer. Though the training time diverges considerably, given a new input, it takes almost the same time (less than a second) for the trained models of all four algorithms to yield the prediction result.

IV. Request Rate Allocation

As shown in Fig. 2, after having been notified of the optimal backend capacity by Clockwork, the developer will purchase a new MBaaS service plan or adjust the instance configuration on Amazon EC2 accordingly. Given the backend capacity fixed for the upcoming hour, there are three potential ways for Clockwork to conduct request scheduling for all users, as shown in Fig. 8.

Proxy mode. As shown in Fig. 8(a), users send all their requests to the cloud servers of Clockwork, which acts as a proxy to redirect these requests to the backend. Given that the backend capacity is N, we can simply sort all requests according to their delay tolerance, and send the top N most urgent requests every minute. Nevertheless, an important concern with this approach is privacy. Users are, in general, not willing

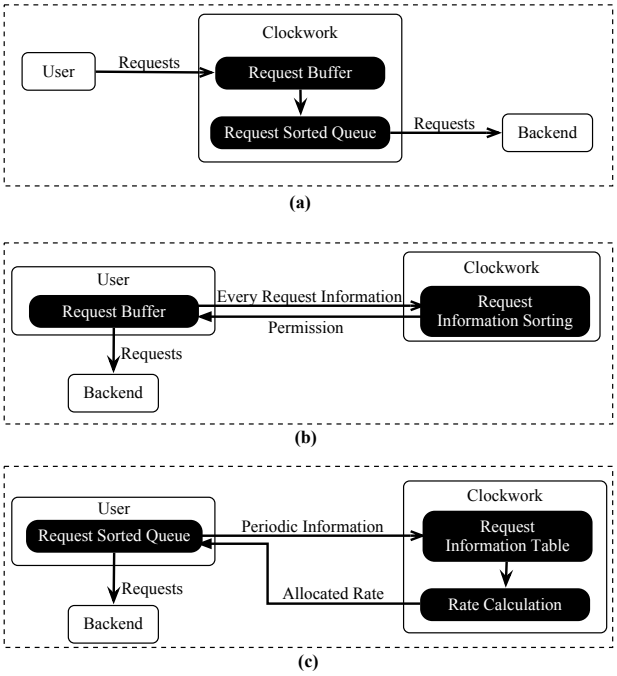


Fig. 8: Three ways of requests scheduling.

to reveal sensitive information to a third-party cloud service provider such as Clockwork. For this reason, it is better for users to send requests directly to the backend by themselves, but with scheduling instructions from Clockwork.

Tight control. As shown in Fig. 8(b), whenever a new request is generated, instead of sending it to Clockwork, the user informs Clockwork of the type and initiation time of the request, then waits for the permission to send the request to the backend. Similar to the proxy mode, every minute, Clockwork can simply give permissions to the top N most urgent requests. Though this approach avoids the privacy issues, it is not scalable as the frequent communications between users and Clockwork may lead to an excessive amount of overhead and long latencies.

Rate allocation. To overcome the drawbacks of a stringent centralized control mechanism and give users more autonomy, we adopt a rate-based scheduling strategy, as shown in Fig. 8(c). Within each minute, Clockwork periodically assigns rates to users, based on their reported request information. Users then schedule their own requests according to the allocated rates. In the remainder of this section, we derive a rate allocation strategy that is proved to be fair and Pareto-optimal.

Assume that there is a finite set of S users. We divide each minute into $1, 2, ..., T$ time slots, and users contact the cloud servers of Clockwork for a new rate at the start of each time slot. Given the backend capacity as N, the cap for the number of requests per time slot is N/T, shared by all users. At time slot τ, the number of requests of user s is denoted by $X_s(\tau)$, so the overall request state is $\boldsymbol{X}(\tau) = \big(X_1(\tau), \cdots, X_S(\tau)\big)$. Clockwork allocates a rate of $R_s(\tau)$ to user s, the maximum number of requests that user s can send within time slot τ. We have $\boldsymbol{R}(\tau) = \big(R_1(\tau), \cdots, R_S(\tau)\big)$. A rate allocation result $\boldsymbol{R}(\tau)$ is feasible if the cap is not hit, that is, $\sum_{s=1}^{S} R_s(\tau) \leq N/T$. Leveraging the definition of utility functions of the network utility maximization (NUM) problem [4], we assume that the utility of assigning rate $R_s(\tau)$ to user s is $X_s(\tau) \cdot U_s\big(\frac{R_s(\tau)}{X_s(\tau)}\big)$, in which $U_s(\cdot)$ is a function of $\frac{R_s(\tau)}{X_s(\tau)}$. Our objective is to maximize the weighted sum of utilities of all users.

$$\max_{R_s(\tau)} \sum_{s=1}^{S} \omega_s(\tau) * X_s(\tau) * U_s\big(\frac{R_s(\tau)}{X_s(\tau)}\big),$$
$$\text{subject to } \sum_{s=1}^{S} R_s(\tau) \leq \frac{N}{T}. \tag{6}$$

in which $\omega_s(\tau)$ is the weight of user s, specified by the developer. The weight $\omega_s(\tau)$ is added to account for the delay tolerance of requests. If most of the requests of user s are urgent, ω_s should be larger. To treat all users without bias, we adopt the same utility function for all users, i.e., $U_s(\cdot) = U(\cdot), \forall s$. Utility function $U(\cdot)$ is usually assumed to be non-decreasing and concave [4]. This is reasonable because a higher rate will naturally lead to a higher utility, and the increment of utility is more significant when the rate is low, but less significant when the rate is already high. We will show that, with a proper utility function $U(\cdot)$, the rate allocation result $\boldsymbol{R}(\tau)$, as the solution to the maximization problem in (6), can be fair and Pareto-optimal. More specifically, we choose the α-fair utility function, widely used in the field of network resource allocation, to help achieve these ideal properties [4].

Definition 1. *(α-fair function). α-fair function, introduced by Mo and Walrand [5], is defined as:*

$$U^{\alpha}(x) = \begin{cases} \frac{x^{1-\alpha}}{1-\alpha}, & \textit{for } \alpha \in (0, \infty) \setminus \{1\}, \\ \log x, & \textit{for } \alpha = 1. \end{cases} \tag{7}$$

Definition 2. *(α-fair rate allocation). A feasible rate allocation result $\boldsymbol{R}$ is called (weighted) α-fair if, for any other feasible rate allocation result $\boldsymbol{R}'$, we have:*

$$\sum_{s=1}^{S} w_s \frac{R'_s - R_s}{R_s^{\alpha}} \leq 0. \tag{8}$$

α-fair rate allocation is equivalent to proportional fair rate allocation when $\alpha \to 1$, and max-min fair rate allocation when $\alpha \to \infty$. In the following context, we first give the rate allocation result according to the maximization problem in (6) with α-fair function, then prove that it is α-fair and Pareto-optimal.

Proposition 1. *At time slot τ, given the request state $\boldsymbol{X}(\tau)$ and the cap of the number of requests N/T, the rate allocation result based on α-fair function is:*

$$R_s^*(\tau) = \frac{(\omega_s(\tau))^{1/\alpha} X_s(\tau)}{\sum_{s=1}^{S} (\omega_s(\tau))^{1/\alpha} X_s(\tau)} \cdot \frac{N}{T}, \forall s. \tag{9}$$

Proof. The Lagrangian for the problem is given by:

$$L(\boldsymbol{R},\mu)=\sum_{s=1}^{S}\frac{\omega_s(\tau)X_s(\tau)}{1-\alpha}\big(\frac{R_s(\tau)}{X_s(\tau)}\big)^{1-\alpha}+\mu(\frac{N}{T}-\sum_{s=1}^{S}R_s(\tau)).$$

The optimal solution corresponds to the stationary point of the Lagrangian function:

$$\begin{aligned}\frac{\partial L}{\partial R_s(\tau)}&=\omega_s(\tau)(\frac{R_s(\tau)}{X_s(\tau)})^{-\alpha}-\mu,\forall s,\\ \frac{\partial L}{\partial \mu}&=\frac{N}{T}-\sum_{s=1}^{S}R_s(\tau).\end{aligned}\tag{10}$$

By setting $\partial L/\partial R_s(\tau)=0$, we have:

$$R_s(\tau)=X_s(\tau)(\frac{\mu}{\omega_s(\tau)})^{-\frac{1}{\alpha}},\forall s.\tag{11}$$

By setting $\partial L/\partial \mu=0$, we have:

$$\sum_{s=1}^{S}R_s(\tau)=\frac{N}{T}=\sum_{s=1}^{S}X_s(\tau)(\frac{\mu}{\omega_s(\tau)})^{-\frac{1}{\alpha}}.\tag{12}$$

Combining (11) and (12), we can get the rate allocation result as (9). □

Theorem 1. *The rate allocation result given by Proposition 1 is weighted α-fair.*

Proof. Let $\boldsymbol{R}(\tau)\neq\boldsymbol{R}^*(\tau)$ denote an arbitrary feasible rate allocation result. Multiplying (11) by $(R_s(\tau)-R_s^*(\tau))$, we have:

$$\big(R_s(\tau)-R_s^*(\tau)\big)\omega_s(\tau)\big(\frac{R_s^*(\tau)}{X_s(\tau)}\big)^{-\alpha}=\mu\big(R_s(\tau)-R_s^*(\tau)\big).$$

Summing over all s, we have:

$$\sum_{s=1}^{S}\omega_s(\tau)X_s(\tau)\frac{R_s(\tau)-R_s^*(\tau)}{(R_s^*(\tau))^{\alpha}}=\mu\sum_{s=1}^{S}\big(R_s(\tau)-R_s^*(\tau)\big).$$

According to (10), we have $\sum_{s=1}^{S}R_s^*(\tau)=N/T$ and $\sum_{s=1}^{S}R_s(\tau)\leq N/T$. Hence, we have:

$$\begin{aligned}\sum_{s=1}^{S}\omega_s(\tau)X_s(\tau)\frac{R_s(\tau)-R_s^*(\tau)}{(R_s^*(\tau))^{\alpha}}&=\mu\sum_{s=1}^{S}\big(R_s(\tau)-R_s^*(\tau)\big)\\ &=\mu(\sum_{s=1}^{S}R_s(\tau)-\sum_{s=1}^{S}R_s^*(\tau))\leq 0.\end{aligned}$$

The strict inequality holds for all $\boldsymbol{R}(\tau)\neq\boldsymbol{R}^*(\tau)$, due to the uniqueness of $\boldsymbol{R}^*(\tau)$. Based on definition 2, the rate allocation result given by (9) is weighted α-fair. □

The efficiency of the rate allocation result given by Theorem 1 is characterized by Pareto-optimality as follows.

Definition 3. *(Pareto-optimality). A rate allocation result $\boldsymbol{R}(\tau)$ is said to be Pareto-optimal, if for any rate allocation result $\boldsymbol{R}'(\tau)$, which satisfies $\forall s, R_s'(\tau)\geq R_s(\tau)$, it must be true that $\boldsymbol{R}(\tau)=\boldsymbol{R}'(\tau)$.*

Pareto-optimality implies that there is no other rate allocation result that can improve every user's utility by assigning everyone a higher rate.

Theorem 2. *The rate allocation result given by Proposition 1 is Pareto-optimal.*

Proof. Assume that there exists a feasible rate allocation result $\boldsymbol{R}(\tau)$, which satisfies $\forall s, R_s(\tau)\geq R_s^*(\tau)$. We have:

$$\begin{aligned}\sum_{s=1}^{S}R_s(\tau)&\geq\sum_{s=1}^{S}R_s^*(\tau)\\ &=\sum_{s=1}^{S}\frac{(\omega_s(\tau))^{1/\alpha}X_s(\tau)}{\sum_{s=1}^{S}(\omega_s(\tau))^{1/\alpha}X_s(\tau)}\cdot\frac{N}{T}=\frac{N}{T}.\end{aligned}$$

If $\boldsymbol{R}(\tau)\neq\boldsymbol{R}^*(\tau)$, there exists at least one user s, $R_s(\tau)>R_s^*(\tau)$, and the cap on the number of requests per time slot will be violated. Therefore, it must be true that $\boldsymbol{R}(\tau)=\boldsymbol{R}^*(\tau)$. □

V. System Implementation

In this section, we first discuss implementation issues regarding the computation of rate allocation on the cloud servers. Then, we address the problem of request scheduling on user devices.

A. Rate Computation on Cloud Servers

1) User Synchronization: In our implementation, we set the number of time slots within a minute as $T=4$, i.e., users report their request information to cloud servers of Clockwork and obtain the allocated rate every 15 seconds. Three kinds of HTTP requests are used for users to communicate with cloud servers. The first kind of HTTP requests are sent when users launch the application or restart the application from background. The cloud servers will reply with an initial rate R_0 and the synchronization reference. R_0 is above zero, since we find that upon logging in, users tend to perform many operations, thus sending many requests. If $R_0=0$, these requests can not be sent until a new rate is assigned at the next time slot, which may lead to a poor user experience.

The second kind of HTTP requests are used to periodically report users' request information and ask for rate allocation. The cloud servers maintain a table of all active users, including the user ID, number of requests $\boldsymbol{X}(\tau)$, weight $\boldsymbol{\omega}(\tau)$, and the rate allocation result $\boldsymbol{R}(\tau)$. Users log in at different times, and the local time on their mobile devices may be different from the time on cloud servers. We need to synchronize users to ensure that they update $\boldsymbol{X}(\tau)$ and $\boldsymbol{\omega}(\tau)$ at (almost) the same time for cloud servers to compute the rate allocation $\boldsymbol{R}(\tau)$. Recall that the response to the first kind of HTTP requests upon user arrival includes a synchronization reference. We use the *second* of the server time as this synchronization reference. For example, if a new user arrives at 00:00:43 of the server time, the synchronization reference will be 43, which is then fed into a (one-time) *timer* function on the user device. The function will be executed after $15-(43 \mod 15)=2$ seconds, i.e., at 00:00:45 of the server time, triggering another (iterative) *timer* function that will communicate with the cloud

servers every 15 seconds. In this way, all users will be synchronized to report their request information at the start of each time slot.

The last kind of HTTP requests are used to inform the cloud servers that the application goes to background or is terminated, so that the user will be removed from the rate allocation table.

If a user's HTTP requests fail, the application will get an error message. In this case, without allocated rate from the cloud servers, the user device will carry out a default request scheduling mechanism, which we will describe in Section V-B2). Meanwhile, the user will keep trying to contact the cloud servers every 15 seconds.

2) Rate Allocation Computation: We classify all requests into $K = 3$ types: type 3 requests are the most urgent, and type 1 requests are the least urgent. Then we define that the weight of a user is the sum of types of all her requests. For example, if user s has 15 type 1 requests, 10 type 2 requests, and 5 type 3 requests, her weight is $\omega_s(\tau) = 15 * 1 + 10 * 2 + 5 * 3 = 50$. It can be easily checked that a user with more urgent requests will get a higher weight.

Since users will turn up for new rates at the start of time slot $\tau + 1$, the cloud servers need to calculate the rate allocation result $\boldsymbol{R}(\tau + 1)$ before the end of time slot τ, by which the reports on $\boldsymbol{X}(\tau)$ and $\boldsymbol{\omega}(\tau)$ haven't been submitted yet. One option is for the cloud servers to wait for $\boldsymbol{X}(\tau)$ and $\boldsymbol{\omega}(\tau)$ at the beginning of time slot $\tau + 1$, and then compute $\boldsymbol{R}(\tau + 1)$ accordingly. However, users' HTTP requests will not arrive at exactly the same time, due to different network conditions. Early users may have to wait an undesirably long time for all other users to report their information, after which the new rate allocation can be calculated. To avoid this situation, we decide that the cloud servers will work out the new rate allocation $\boldsymbol{R}(\tau + 1)$ 5 seconds before the end of each time slot, using the available information $\boldsymbol{X}(\tau - 1)$ and $\boldsymbol{\omega}(\tau - 1)$.

3) Request Cap Adjustment: When computing rate allocation for each time slot, we assume that the cap on the number of requests to be sent is N/T, in which N is the backend capacity, and T is the number of time slots within a minute. However, as some of the allocated rates may be unused (as $X_s(\tau) < R_s(\tau)$), we can adjust the cap to make a better use of the backend capacity. At the first time slot of each minute, the cap will be N/T; at the following time slots, the cap will be N/T plus the remaining rates from the previous time slot. For example, if the backend capacity is 1800, and each minute is divided into $T = 4$ time slots. At time slot 1, the request cap is $1800/4 = 450$. If the actual number of requests sent to the backend during time slot 1 is 430, the request cap at time slot 2 will be $450 + (450 - 430) = 470$.

B. Request Scheduling on User Devices

1) Dynamic Request Scheduling: The user device will keep a queue of requests ranked in a non-ascending order of priorities that are jointly determined by the type and the initiation time of a request. A new request will be inserted into the queue, and wait for its turn to be sent to the backend. Let $\boldsymbol{Q}_s(\tau) = (q_{s,1}(\tau), q_{s,2}(\tau), \cdots)$ denote the request queue of user s, in which $q_{s,i}(\tau)$ is the priority of the i-th request, and we have $q_{s,1}(\tau) \geq q_{s,2}(\tau) \geq \cdots$. The number of requests in the queue is denoted by $|\boldsymbol{Q}_s(\tau)|$.

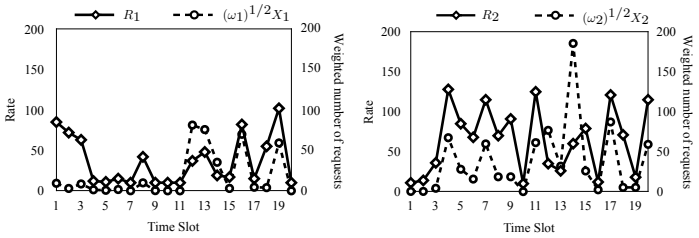
A major problem facing the request scheduling on the user device is that the new rate obtained at the start of a time slot may not be able to cater to the new requests arrived later during the time slot. More specifically, at the start of time slot τ, upon receiving a new rate $R_s(\tau)$, if user s instantly sends the top $R_s(\tau)$ requests in the queue $\boldsymbol{Q}_s(\tau - 1)$ (given that $|\boldsymbol{Q}_s(\tau-1)| > R_s(\tau)$), an urgent request generated later during time slot τ can no longer be sent. For example, at 00:00:15, with a queue of $(2, 1, 1)$ and an allocated rate of 2, the user device sends the first 2 requests in the queue right away to the backend. At 00:00:25, a new request with a priority of 3 is generated, but has to be detained as the allocated rate is drained.

The above-mentioned problem arises as the user device can not predict future request dynamics and preserve enough rate for them. To mitigate this problem, we design a threshold-based request scheduling mechanism that works as follows. Firstly, we set a priority threshold θ_q. At the beginning of time slot τ, with the new rate $R_s(\tau)$, user s will immediately send the requests whose priority exceeds θ_q to the backend. If the number of such requests is greater than $R_s(\tau)$, the first $R_s(\tau)$ will be sent; otherwise, the rest of the rate is reserved for later requests with high priorities. If a newly-generated request has a priority lower than θ_q, it will be inserted in the queue; otherwise, it will be checked whether the allocated rate is exhausted. If yes, the request will be inserted in the queue; otherwise, it will be sent at once to the backend (note that in this case, the queue only contains requests whose priority is lower than θ_q). At the end of time slot τ, if the remaining rate is greater than zero, the user device will send as many queued requests as possible, before asking for a new rate.

2) Default Request Scheduling: If the user device fails to get responses from cloud servers of Clockwork, either upon logging in or during periodic rate allocation, a default request scheduling mechanism based on the Additive Increase Multiplicative Decrease (AIMD) algorithm, will come into effect. The requests will be sent to the backend at a lower rate if it is inferred that the current overall request demand is high, vice versa. There are various indications of the current request demand. For example, when using MBaaS, if the request limit is hit, further requests will be rejected with an error message of error code 155. If a user receives such an error message, she should multiplicatively decrease the rate by b; otherwise, she will additively increase the rate by a:

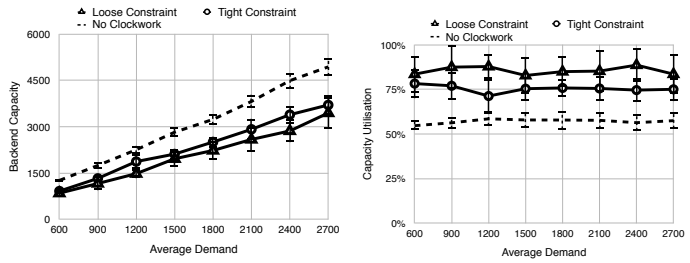
$$R_s(\tau) = \begin{cases} R_s(\tau - 1) + a, & \text{if no error code,} \\ R_s(\tau - 1)/b, & \text{if error code.} \end{cases} \tag{13}$$

The default request scheduling will continue until the user device succeeds in getting new allocated rate from cloud servers of Clockwork.



(a) User 1 (b) User 2

Fig. 9: Experimental result of rate allocation.



(a) Required backend capacity Backend capacity utilization

Fig. 10: Simulation result of backend capacity planning.

VI. Performance Evaluation

A. Experimental Result

We have developed the client-side of Clockwork as a third-party framework (library) that can be readily used by mobile application developers, using Swift on the iOS platform as a proof-of-concept. It is implemented using Java on the Amazon Web Service (AWS) platform. For the purpose of our real-world experiments, we have also developed a social messaging iOS application, using Facebook Parse (an MBaaS platform) as its backend, supported by our third-party framework. We identify 17 different kinds of requests, such as *send a message, send friend request, change username* and so on. We divide them into $K = 3$ types: type 3 requests are the most urgent and type 1 requests are the most delay-tolerant.

To evaluate the performance of Clockwork, we carried out a pilot trial with 15 iPhone users. We collected data for 20 hours. Only the statistics regarding the request demand are logged, but not message contents, in order to protect user privacy.

Rate allocation. We set $\alpha = 2$ in the $\alpha-$fair utility function, and show the rate allocation results of two users in Fig. 9. Generally speaking, a user will achieve a higher rate if her weighted number of requests is larger. Nevertheless, the allocated rate of a user also depends on the request demand of other users. For example, there is a plunge in user 1's request demand at time slot 15, but the allocated rate is almost the same as that of the previous time slot. This is because the request demand of user 2 also falls dramatically at time slot 15, making the ratio of the two users' request demand (i.e., $\frac{(\omega_1)^{1/2}X_1}{(\omega_2)^{1/2}X_2}$) in time slot 15 approximately the same as that in time slot 14. Thus, at time slot 15, each user is assigned a rate similar to that in the previous time slot.

Capacity planning. We use the collected data trace as inputs to the optimization problem (1). We compare Clockwork and the baseline, where the backend capacity equals the peak demand. The backend utilization is calculated as the ratio of the average request demand to the backend capacity. The results are shown in Table I. With Clockwork, the backend cost for the developer can be reduced as much as 67.7%. Furthermore, Clockwork allows the developer to make a much better use of backend resources. Without Clockwork, the backend utilization is mostly below 50%, since the peak demand is much higher than the average demand. With Clockwork, the backend utilization can be increased by as much as 76.8%.

TABLE I: Experimental Result of Capacity Planning

Hour	1	2	3	4	5
Cost reduction	67.7%	0.0%	21.9%	32.5%	5.5%
Utilization(baseline)	20.3%	23.2%	48.5%	45.9%	37.1%
Utilization(Clockwork)	63.0%	23.2%	62.0%	67.9%	39.2%
Hour	6	7	8	9	10
Cost reduction	31.3%	55.3%	21.4%	0.0%	40.6%
Utilization(baseline)	39.2%	23.5%	19.0%	11.8%	39.1%
Utilization(Clockwork)	57.0%	52.6%	24.2%	11.8%	65.8%
Hour	11	12	13	14	15
Cost reduction	63.0%	39.6%	75.2%	0.0%	56.6%
Utilization(baseline)	18.6%	32.2%	19.0%	17.4%	27.2%
Utilization(Clockwork)	50.3%	53.3%	76.8%	17.4%	62.8%
Hour	16	17	18	19	20
Cost reduction	23.4%	65.4%	50.7%	55.9%	24.0%
Utilization(baseline)	44.4%	20.9%	35.6%	24.6%	15.2%
Utilization(Clockwork)	57.9%	60.5%	72.3%	55.7%	20.0%

B. Simulation Result

We evaluate the proposed backend capacity planning optimization model using a synthetic dataset generated as follows. There is a total of 100 users, whose request generation processes follow independent and identical Poisson distributions. We assume that there are 3 types of requests. Type 3 requests are most urgent, $\overline{\delta_{ij}^3} = 0, \forall i, j$. No request is to be delayed for more than 5 minutes, and the upper bounds for type 2 and type 1 requests are $\forall j - i \leq 5, \overline{\delta_{ij}^2} = m - 0.02 * (j - i)$, and $\overline{\delta_{ij}^1} = m - 0.01 * (j - i)$, respectively. If m is higher, more requests can be delayed, meaning that the constraint on request delay is looser, vice versa.

It is shown in Fig. 10(a) that the backend capacity should augment with the average demand, but Clockwork can cut down the required backend capacity by as much as 36.3%. The capacity reduction is more significant under loose delay constraint ($m = 0.4$) than under tight delay constraint ($m = 0.1$). As shown in Fig. 10(b), provisioning for the peak demand will result in resource wastage as the backend capacity is underutilized for around 55% at most of the time. With request scheduling of Clockwork, the demand profile becomes smoother, and the backend capacity can be utilized more efficiently. The improvement in the backend capacity utilization can be as high as 57.3% under loose constraint, and 43.2% under tight constraint.

VII. Related Work

Cloud Resource Provisioning. Many existing works have proposed to use predicative auto-scaling for dynamic resource provisioning in cloud computing. In [6], [7], a statistical model is used to predict the demand of videos, based on which

the video service provider can dynamically book bandwidth resources to match the fluctuated demand. In [8], [9], an epidemic model is built to forecast the viewing requests in a social media application. Similar predictive auto-scaling models have also been built for resource allocation [10]–[12] and power consumption [13], [14] in cloud systems. In this paper, we determine the backend capacity based on predicted request demand, but our work distinguishes from previous works in two aspects. First, there is a mismatch between timescales of request demand fluctuations and backend capacity adjustment, so that fine-grained auto-scaling is infeasible. Second, unlike the rigid demand in existing models, we can exploit the delay tolerance of requests to change the demand profile and help the application developer cut cost. This is similar to [15], in which pricing is used to incentivize users to shift their demand. But rather than relying on users' subjective decisions, our request scheduling mechanism is directly controlled by Clockwork.

NUM-based Resource Allocation. The pioneering work of Kelly et al. [16] first introduced the novel idea of Network Utility Maximization (NUM) based resource allocation. A nice survey of the research on network utility maximization problem is given by [4]. Mo and Walrand [5] first introduced the α-fair utility function to be used as the objective function. NUM-based network resource allocation has been widely applied to rate allocation in Internet congestion control protocol [17], resource allocation in cellular networks [18], and congestion control in wireless ad hoc networks [19]. We build a similar model but relate it to request scheduling in mobile applications. In particular, we leverage the model to allocate rates to users according to the quantity and delay tolerance of their requests.

VIII. Conclusion

Backend maintenance is essential for mobile application developers. However, the mismatch of the timescales between the request demand variance and the backend capacity adaptation makes it difficult to manage the resource and the cost of the backend. To address this problem, we design a third-party cloud service Clockwork, which plans the backend capacity on a long timescale, and schedules requests on a short timescale. To start with, Clockwork exploit machine learning algorithms to predict future demand based on historical data. We show that deep learning models are powerful in prediction but take a much longer time to train than simpler models. Given the estimated future demand, we help developers minimize the backend cost with assurance of limited request delay. Then, abiding by the backend capacity, Clockwork schedules requests from all users with a fair and Pareto-optimal rate allocation mechanism, enabling individual users to prioritize their own requests, thus protecting user privacy. We have implemented the server-side of Clockwork on Amazon Web Service (AWS) platform, and the client-side on iOS-based mobile devices. Our evaluation results confirm that Clockwork can effectively help developers trim down backend cost and make a better use of backend resources.

IX. Acknowledgement

The co-authors would like to acknowledge the generous research support from a NSERC Discovery Research Program and a NSERC Strategic Partnership Grant titled "A Cloud-Assisted Crowdsourcing Machine-to-Machine Networking Platform for Vehicular Applications at the University of Toronto.

References

[1] Statistas, "Number of Mobile App Downloads Worldwide from 2009 to 2017 (in millions)," http://goo.gl/BqEVrK.

[2] A. Dogtiev, "App Revenue Statistics 2015," http://goo.gl/MwJ5R1, November 16, 2015.

[3] V. Mobile, "Developer Economics Q1 2015: State of the Developer Nation," http://goo.gl/Iq1EEI, February, 2015.

[4] Y. Yi and M. Chiang, "Stochastic Network Utility Maximisation-A tribute to Kelly's paper published in this journal a decade ago," *European Transactions on Telecommunications*, vol. 19, no. 4, pp. 421–442, 2008.

[5] J. Mo and J. Walrand, "Fair End-to-End Window-Based Congestion Control," *IEEE/ACM Transactions on Networking*, vol. 8, no. 5, pp. 556–567, 2000.

[6] D. Niu, Z. Liu, B. Li, and S. Zhao, "Demand Forecast and Performance Prediction in Peer-Assisted On-Demand Streaming Systems," in *IEEE International Conference on Computer Communications, mini-conference*, 2011.

[7] D. Niu, H. Xu, B. Li, and S. Zhao, "Quality-Assured Cloud Bandwidth Auto-Scaling for Video-on-Demand Applications," in *IEEE International Conference on Computer Communications*, 2012.

[8] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. Lau, "Scaling Social Media Applications Into Geo-Distributed Clouds," in *IEEE International Conference on Computer Communications*, 2012.

[9] ——, "Scaling Social Media Applications Into Geo-Distributed Clouds," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 689–702, 2015.

[10] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A Scalable Application Placement Controller for Enterprise Data Centers," in *ACM International Conference on World Wide Web*, 2007.

[11] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive Elastic Resource Scaling for Cloud Systems," in *IEEE International Conference on Network and Service Management*, 2010.

[12] M. Wang, X. Meng, and L. Zhang, "Consolidating Virtual Machines with Dynamic Bandwidth Demand in Cata Centers," in *IEEE International Conference on Computer Communications, mini-conference*, 2011.

[13] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and Performance Management of Virtualized Computing Environments via Lookahead Control," *Cluster computing*, vol. 12, no. 1, pp. 1–15, 2009.

[14] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska, "Dynamic Right-Sizing for Power-Proportional Data Centers," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1378–1391, 2013.

[15] S. Ha, S. Sen, C. Joe-Wong, Y. Im, and M. Chiang, "TUBE: Time-Dependent Pricing for Mobile Data," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 247–258, 2012.

[16] F. Kelly, A. Maulloo, and D. Tan, "Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability," *Journal of the Operational Research Society*, pp. 237–252, 1998.

[17] M. Chiang, D. Shah, and A. Tang, "Stochastic Stability Under Network Utility Maximization: General File Size Distribution," in *Proceedings of 44th Allerton Conference on Communication, Control and Computing*. Citeseer, 2006.

[18] A. L. Stolyar, "On the Asymptotic Optimality of the Gradient Scheduling Algorithm for Multiuser Throughput Allocation," *Operations Research*, vol. 53, no. 1, pp. 12–25, 2005.

[19] M. J. Neely, E. Modiano, and C.-P. Li, "Fairness and Optimal Stochastic Control for Heterogeneous Networks," *IEEE/ACM Transactions on Networking*, vol. 16, no. 2, pp. 396–409, 2008.