Temporal Rate Limiting: cloud elasticity at a flat fee

John S. Otto
Northwestern University

Rade Stanojevic
Institute IMDEA Networks
Telefonica Research

Nikolaos Laoutaris
Telefonica Research

***Abstract*—In the current *usage-based* pricing scheme offered by most cloud computing providers, customers are charged based on the capacity and the lease time of the resources they capture (bandwidth, number of virtual machines, IOPS rate, *etc.*). Taking advantage of this pricing scheme, customers can implement *auto-scaling* purchase policies by leasing (*e.g.*, hourly) necessary amounts of resources to satisfy a desired QoS threshold under their current demand. Auto-scaling yields strict QoS and variable charges. Some customers, however, would be willing to settle for a more relaxed statistical QoS in exchange for a predictable *flat* charge. In this work we propose Temporal Rate Limiting (TRL), a purchase policy that permits a customer to allocate optimally a specified purchase budget over a predefined period of time. TRL offers the same expected QoS with auto-scaling but at a lower, flat charge. It also outperforms in terms of QoS a naive flat charge policy that splits the available budget uniformly in time. We quantify the benefits of TRL analytically and also deploy TRL on Amazon EC2 and perform a live validation in the context of a "blacklisting" application for Twitter.**

I. Introduction

The idea of *load aware resource sharing* has recently been applied to a variety of networking problems including wireless spectrum sharing [4], [11], [16], router DDoS protection [27], distributed rate limiting [22], [26], seeder bandwidth allocation [19], *etc.* What is common in all the above applications is their *spatial load awareness*: a set of resources is partitioned among a set of spatially distributed nodes based on their current loads. Here we take an orthogonal approach and look at *temporal load awareness*, which is used for partitioning a set of resources across time in accordance with the received demand. Whereas spatial load awareness becomes dominant in highly distributed applications like P2P networks, temporal load awareness becomes the primary concern when the resources are highly centralized. Applications utilizing centralized resources are faced with the problem of over-dimensioning and poor utilization of expensive resources resulting from the fact that owned infrastructure has to be provisioned according to the peak of a typically highly variable demand. Cloud-computing platforms like those offered by Amazon EC2, GoGrid or Rackspace have allowed applications to avoid over-dimensioning by applying elastic temporal load aware purchase policies driven by time-varying demand.

Auto-scaling: The existing approach for dealing with temporal variabilities in the demand of cloud-based applications is called auto-scaling [1], [15]. A customer purchases dynamically from the platform enough resources at each point of time (*e.g.*, hourly) to ensure a prescribed QoS level for its clients (*e.g.*, delay in completing a purchase, or bandwidth for a video stream). Since the cloud permits fast capturing and releasing of resources at a fine granularity, this simple policy can adapt to daily and weekly demand variability and guarantee strict QoS at a minimum charge. A direct consequence of the operation of auto-scaling is that the resulting charge paid to the cloud operator is generally variable and hard to predict. To quote a Gartner report[1]: "A consumer of an unlimited capability will consume unexpected amounts".

Can elasticity be offered at a flat fee? Imagine a customer that wants to take advantage of the elasticity offered by the cloud but would rather pay a flat predefined fee over a certain period of time (day, week, month) instead of a variable one. Various reasons can give rise to such a requirement. *Cost predictability* is of paramount importance for several companies, especially during their early life when the budget for leasing hosting resources is tight [12], [17]. At the other extreme, big customers are not driven by the requirement to guarantee a minimum QoS but would rather like to *maximize user satisfaction* granted a fixed budget that they can spend over a certain period of time [12]. Notice that the latter objective is not equivalent to a maintenance of QoS objective (a la auto-scaling), especially if one considers non-linearities in the satisfaction (user QoS) function and demand unpredictability. Last but not least, the cloud providers themselves can benefit by a scheme in which customers pay flat fees, committing in advance for longer periods of time. In auto-scaling, there is no financial long-term commitment on the part of customers and thus the cloud has to perform statistical multiplexing over short periods of time. However, under a flat-fee model, the commitment horizon is longer and thus the operator can improve the efficiency of statistical multiplexing and consequently the amortization of the platform costs. Overall, there exist several historical examples in which the success and increasing adoption of a service are achieved by pushing for simpler, flat pricing schemes [17].

Our contributions: The naive solution to realizing a flat charge model under the current pricing scheme is to partition an available budget of, say, C virtual machine instance-hours per day uniformly by leasing $C/24$ VMs continuously throughout the day. However, the demand of many services is known to exhibit strong variations over time, including daily diurnal patterns as well as day-of-week phenomena [3], [10], [14], [24]. Temporal load awareness calls for a non-uniform

[1] http://blogs.gartner.com/daryl_plummer/2009/03/11/cloud-elasticity-could-make-you-go-broke/

978-1-4673-1017-8/12/$31.00 ©2012 IEEE

split of the C instance-hours over some time period, such that the number of allocated instances is increased during peak hours to benefit end-user QoS, and shrunk during off-peak hours to avoid resource under-utilization. To that end, we introduce and analyze *Temporal Rate Limiting (TRL)*, a purchase policy for cloud resources that enables a customer (*e.g.*, a startup company) to allocate dynamically a predefined purchase budget over a certain period of time so as to optimize the QoS offered to its own clients.

II. Temporal Rate Limiting

In this section we first formulate the Temporal Rate Limiting problem and then present offline and online solutions for it.

A. Problem statement

Consider a customer that has a budget of C units (say VM-hours) per day for running elastic services in the cloud. The day is split in T time slots (say $T = 24$ hours) and the budget can be split between these slots in an arbitrary manner by buying C_t resource-units[2] at time t. We also assume that the cost of resource-unit can change[3] in time and is given by r_t at time slot t.

The demand is also measured in application-specific units (number of viewers in a VoD system, number of objects that need rendering in a photo sharing service, etc.) and is represented by the time series: $D_1, \ldots, D_T$. Then the performance at time slot t is measured through a metric q that is a (monotone) function of the demand intensity D_t and the capacity C_t:

$$q_t = f(C_t, D_t) = f_t(C_t).$$

The goal of TRL is finding the allocation $\mathbf{C} = (C_1, \ldots, C_T)$ of budget C that optimizes the expected daily performance:

$$Q(\mathbf{C}) = \sum_{t=1}^{T} q_t \tag{1}$$

$$\text{s.t.} \quad \sum_{t=1}^{T} r_t C_t = C. \tag{2}$$

In order to ease the technical exposition we set the following assumption. We note that this assumption is not critical for the design of TRL, as the dimensionality of the problem is low enough to allow exact numerical solution to TRL even for non-convex functions f_t (see also [6] and references therein). It, however, holds for a wide range of cost/utility functions and significantly reduces the complexity of the exposition.

Assumption 1: The functions $f_t(\cdot)$ that relate capacity to performance are convex (resp. concave) functions for all t if the optimization problem is minimum (resp. maximum) seeking.

Comment 1: The choice of performance metric is application dependent. What is a relevant performance indicator is driven by the application needs and it is hard to isolate a single performance metric usable in all conditions. This is the reason for the general presentation we follow here. Finally we note that whether the optimization problem is maximum or minimum seeking will be obvious from the context.

Comment 2: There are several sources of uncertainty (randomness) in the design of TRL. These include: (1) The demand time series (which can often can be estimated with a reasonable accuracy from the long- and short-term history) [3], [5], [14], [24]; (2) the model of the cost function $f(C_t, D_t)$ that needs to be estimated and is an input for the problem [18]; (3) hardware interference (which is relatively small for applications that are not I/O intensive, but can be considerable otherwise) [2]; and (4) cost of resources that may vary on the time scale of hours or even minutes [1]. In Section IV (and also the Technical report [25]) we show that TRL is robust to those sources of uncertainty.

Comment 3: TRL can be implemented either by the customer or the service provider. The customer who wants to control its daily bill of the cloud services can easily incorporate a controller that leases the resources subject to daily budget and strive to optimize the performance. Also, a cloud services provider can offer TRL as a feature for maximizing the daily-averaged performance the customer receives subject to a daily budget.

B. Off-line solution to TRL

Assuming that all the parameters are known in advance, the solution to the problem (1)-(2) is relatively straightforward and is discussed below. It is also possible to derive closed-form solutions for some specific scenarios as we show later in Section III-A.

The problem (1)-(2) is a standard non-linear convex optimization problem with a linear constraint solvable by, for example the gradient ascent method. Alternatively one can derive more intuition on the nature of the optimal point by taking advantage of the structure of the optimization problem (which will also be used later in Section II-C3 for the design of online TRL). Namely, let λ be the Lagrange multiplier of the optimization problem, then the Lagrange function is

$$\Lambda(\mathbf{C}, \lambda) = Q(\mathbf{C}) - \lambda(\sum_{t=1}^{T} C_t - C) = \sum_{t=1}^{T} f_t(C_t) - \lambda(\sum_{t=1}^{T} r_t C_t - C).$$

The vector $\mathbf{C}$ that minimizes $Q(\mathbf{C})$ must satisfy:

$$\frac{\partial \Lambda}{\partial C_t} = 0 \quad \text{for all } t,$$

which is equivalent to:

$$\frac{f_t'(C_t)}{r_t} = \lambda \quad \text{for all } t. \tag{3}$$

[2]The resource-unit is application specific abstraction and in the context of cloud computing can be a VM-hour, $Mbps$-hour, etc.

[3]Note that majority of cloud providers have fixed (time-independent) price for the resource. However, some providers, such as Amazon EC2, start to offer variable, demand-dependent, prices for the resources. However over the last 6 months the variation in time of the cost of on-spot EC2 instances is very small, typically under 10%.

Finding $(\mathbf{C}, \lambda)$ that satisfy (2) and (3) can be done numerically by solving the following equation:

$$h(\lambda) := \sum_{t=1}^{T} r_T (f_t')^{-1}(r_t \lambda) - C = 0. \quad (4)$$

Given that f_t is a convex function, f_t' is a monotone function and indeed has the inverse $(f_t')^{-1}$ that is also a monotone function. The function $h(\cdot)$ is therefore a monotone function and equation (4) can be solved simply by any zero finding method (binary search, Newton method, etc).

C. Implementing TRL online

To decide on how many resources (virtual machines) we need to buy at time slot t under a budgeting constraint to optimize the performance over a time horizon we need to solve several practical problems hinted at in Section II-A: (1) demand forecasting; (2) accurate identification of the relationship $f(C, D)$ between offered demand D (in jobs per second) and capacity C (in number of VMs); and (3) actual control system for determining the number of VMs to buy, subject to the budget and optimization criteria.

1) Demand forecasting: For demand forecasting we use the Sparse Periodic Auto-Regression (SPAR) from [5]. The demand at time t is forecasted as:

$$D_t = \sum_{i=1}^{n_0} \alpha_i D_{t-i \cdot T} + \sum_{j=1}^{n_1} \beta_j \Delta D_{t-j},$$

$$\Delta D_{t-j} = D_{t-j} - \frac{1}{n_0} \sum_{i=1}^{n_0} D_{t-j-i \cdot T}.$$

where α's and β's are obtained through the least squares method. The first part of the above model does the periodic prediction over a time period T that corresponds to 24 hours in this paper[4]. Interestingly, higher order models result in minor improvement over the first order model ($n_0 = n_1 = 1$) and throughout the paper we use $n_0 = n_1 = 1$.

2) Mapping demand and capacity to performance: In order to use the insights from Section II-B and solve the optimization problem (1)-(2) we need to know the functions $f_t(C_t) = f(D_t, C_t) = q_i$, that relate performance q_i with generated demand D_i and capacity C_i. These functions can be modeled for some simple systems with known job size distributions. However, in general, accurate identification of function $f(\cdot, \cdot)$ requires fine-grained benchmarks for a range of values of demand and capacity that we perform in an off-line manner.

3) Online TRL: For those services that have purely periodic demand pattern, we can have accurate enough long-term demand prediction and apply the framework from Section II-B to directly solve the problem in an off-line manner. However, many services have a demand pattern that is more volatile and harder to predict at time scales greater than one hour. That is the reason we choose to solve the optimization problem with 'soft' budget constraint as follows. The key observation is that the point $\mathbf{C} = (C_1, \ldots, C_T)$ that maximizes the expected utility satisfies (3). Instead of picking one λ to solve the system exactly, we continually search for it, using a feedback control loop. Namely, we keep internal variable $\lambda(t)$, and determine C_t as:

$$C_t = (f_t')^{-1}(r_t \lambda(t)). \quad (5)$$

where $f_t(x) = f(D_t, x)$ is the function for the forecasted value of demand D_t at time t.

Then we update λ to account for the difference between used resource during the preceding T time slots and the cost constraint C:

$$\lambda(t+1) = \lambda(t)(1 + \eta(C - \sum_{i=0}^{T-1} r_{t-i} C_{t-i})), \quad (6)$$

where $\eta > 0$ is the gain parameter such that in steady state $\lambda(t)$ is stabilized around the optimal value. Throughout this paper we use the gain parameter $\eta = \frac{0.1}{C}$. More detailed analysis on the stability of the controller (6) is out of scope of this paper. The online TRL does not strictly enforce the cost constraint (2) over the time period of T time slots, but rather strives to keep long-term average cost at the desired level C. The pseudocode of online TRL is given in Figure 1.

```
1 TRL_online()
2     At time slot t do
3       D_t = forecast(D_0, ..., D_{t-1})
4       C_t is such that df(D_t,C_t)/dC_t = r_t λ(t)
5       λ(t+1) = λ(t)(1 + η(Σ_{i=0}^{T-1} r_{t-i} C_{t-i} − C))
6     enddo
```

Fig. 1: Pseudo-code of online TRL

III. Simple analytic model for TRL

In this section we look at the particular problem of minimizing the daily mean/median completion time of a photo sharing online service, like SmugMug [15]. We model the service as a $M/M/1$ system, where the demand and resources are measured through the request and service rates, and the goal is the minimization of the average per-photo rendering time, subject to a daily budget C: the number of VM-hours used. The request rate, D_i, is measured in uploaded photos per second. The service rate, S_i represents the average number of photos that can be rendered per second and is a linear function of the number of running VMs: $S_i = \alpha C_i$, where α is the relative speed of one VM. Without loss of generality we take $\alpha = 1$. For brevity, we assume that the cost of a resource (VM) does not change in time: $r_t = 1$ for all t. Extending to arbitrary r_t is straightforward.

A. Mean response times

At time slot t, the performance metric, the sum of mean-response-times for all D_t jobs, is a simple function of the arrival D_t and service rate C_t (that corresponds to the number of online VM at time t)

$$q_t = D_t \frac{1}{C_t - D_t}.$$

[4]Taking into account weekly periodicity would improve the accuracy of the forecasting algorithm. However, most of our traces are too short to allow prediction on the weekly time scales.

The expected daily job mean response time can be approximated [9] by:

$$Q(\mathbf{C}) = \frac{1}{\sum_{t=1}^{T} D_t} \sum_{t=1}^{T} \frac{D_t}{C_t - D_t}. \quad (7)$$

Proposition 1: The vector $\mathbf{C} = (C_1, \ldots, C_T)$ that minimizes the expected daily job mean response time $Q(\mathbf{C})$ in $M/M/1$ scenario described above is given by:

$$C_t = D_t + \frac{C - \sum_{j=1}^{T} D_j}{\sum_{j=1}^{T} \sqrt{D_j}} \sqrt{D_t}. \quad (8)$$

Proof: See the Technical report [25]. ■

We conclude with the observation that similar analysis is possible for many other queueing models (including straightforward generalization to $M/G/1$ queue), that allow closed-form dependence between the capacity C_t and the performance q_t (expressed as a function of the statistical parameters of the job size distribution).

B. Median and αth-percentile response times

Because of the space limitation we omit the analytical results on TRL with objective functions given by median and αth-percentile response times, which can be found in the technical report [25].

C. TRL vs. auto-scaling

We note that in the model described above, it is possible to derive closed-form expression for the difference in cost of auto-scaling and (optimal) TRL that result in the same expected daily average response time. For the detailed analysis we refer the reader to [25]

IV. Deploying TRL on Amazon EC2

In order to understand the behavior of TRL on a real cloud platform and factor the potential impacts of virtualization we deployed TRL on Amazon's EC2 cloud and used it to control the number of virtual machines that check tweets for evidence of spam. In this case too, TRL provided a significant improvement in mean application response time, which in certain cases was an order of magnitude better than uniform allocation.

A. Twitter Blacklisting

We adopt a blacklisting application for a microblogging service (e.g.,Twitter) that searches the content and linked web pages of microblog posts ("tweets") for a set of blacklisted words or patterns. A similar service is offered by filttr[5], a startup that processes tweets based on appropriate keyword filters.

This application can be used for blocking spam or allow users to build personalized filters. In this work, we search for a subset of the patterns in the regular expression blacklist used by wikimedia.org [6] to block spam.

[5]https://filttr.com/

[6]http://meta.wikimedia.org/wiki/Spam_blacklist

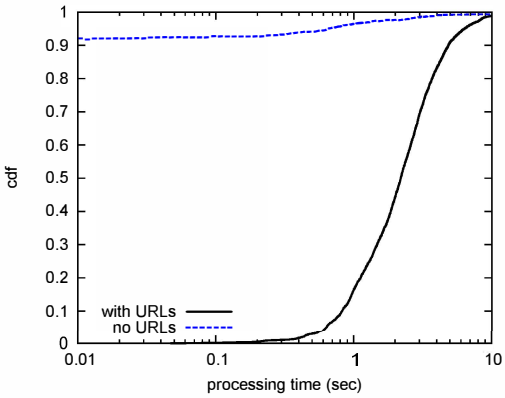


Fig. 2: Processing time for tweets with and without web page links.

To process a tweet, we first identify URLs and fetch the content of linked documents. For each HTTP URL, we read the headers to determine the MIME type of the document (following any HTTP 30X redirections), and download the entire content if it is a text or HTML file. Then, we iterate over each of the elements in the blacklist, scanning the text of the tweet and that of all its linked web pages. The application returns the tweet with a list of all the blacklist elements that matched, as well as a list of the web pages that were scanned.

B. Dataset and Processing

We characterize the requirements of our application using a trace of posts to Twitter collected by [20], and examine several aspects of the dataset. On our target platform (m1.small instance on Amazon EC2), we find that the average processing time per tweet is approximately 0.8 seconds.

The distribution of processing time is bimodal. The tweets that have URLs take significantly longer to process than the tweets without URLs. First, it takes up to several seconds to open a connection and fetch the content of the page. Additionally, it takes much longer to do the regular expression search on webpages (sometimes as large as ˜100KB) compared to tweets with no URLs, which have at most 140 characters.

Approximately 36% of the tweets in the dataset we used contained at least one link to an HTML or text document. In Figure 2, we show the distributions of processing time for tweets with and without URLs. Without URLs, the average processing time is only 173 msec, which is significantly higher than the median of 0.9 msec because about 10% of these tweets have URLs that point to documents that are not HTML or text files. In these cases, it takes time to read the HTTP headers to determine the type of content on the page. For tweets with URLs, the average processing time is about 2.66 sec.

Next, we determine the number of servers required to handle various system loads (i.e. tweets per second), given these characteristics of the tweets in our dataset and the amount of time needed to process each tweet. Since each tweet takes on average 0.8 sec to be processed, then each server can process $\frac{1}{0.8} = 1.25$ tweets per second. The system load levels generated by our Twitter trace range from 4 to

12 tweets/second peak on a typical weekday. In terms of the number of servers required to service this load, this translates to a minimum of 4 to 10 servers, depending on the time of day.

C. System Design

We design a system that runs on Amazon EC2 to execute this Twitter blacklisting application, and which implements both the TRL and uniform allocation policies.

We divide the responsibilities of the system among several different virtual machines running in EC2. In one virtual machine, we run a coordinator that accepts connections from clients (who submit tweets for processing) and manages job handling with workers via a queue of jobs. In another, we run a client that plays back the tweets in the Twitter trace into the coordinator. We use additional virtual machines for running the servers that do the actual processing of the tweets.

The coordinator implements a first-in, first-out queue of tweets to be processed, and workers follow a pull-based model to fetch jobs from the queue. Our design is preferable to a push-based model, in which tweets are committed to a single queue, and may be subject to indeterminate delays because a single job at that worker took a long time to complete. Although our design is simple and could be optimized for performance (e.g. processing multiple tweets per worker), it does provide a valid comparison between the tested allocation policies.

Under our pull-based model, short processing jobs at the head of the queue will continue to be served quickly as long as there is at least one worker that is not busy processing a long job. As a result, having more workers reduces the probability that short processing jobs are delayed by longer jobs saturating all the workers.

D. Performance Evaluation

We evaluate the performance of our application running under TRL in comparison to a uniform allocation model in terms of several response time metrics. We define response time to be the latency from when a job is sent by the client until the response it is received at the client.

For each experiment, we select either TRL or the uniform allocator as the policy to define how many servers to use, and define a daily budget to limit the total number of server hours. Our smallest budget is that which would allow the uniform allocation policy to have sufficient server time to run the necessary 10 servers (at full processing capacity) for the whole day – 240 server hours. From here, we increase the number of servers that can be run for the whole day to 18 (432 server hours).

We use the same two-day region of the Twitter trace, covering all of a Tuesday and Wednesday, for each experiment. Figure 3 shows the trends in demand – tweets per second – for these two days of the trace. For the TRL experiments, we train the SPAR model with the demand from the first day and report the performance from the second day of the trace. For the uniform allocation runs, we simply report the performance statistics from the second day of the trace.

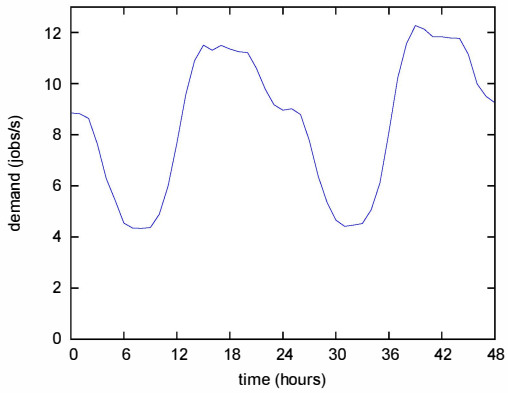


Fig. 3: Demand rate (tweets per second) for a 48-hour segment of the Twitter trace.

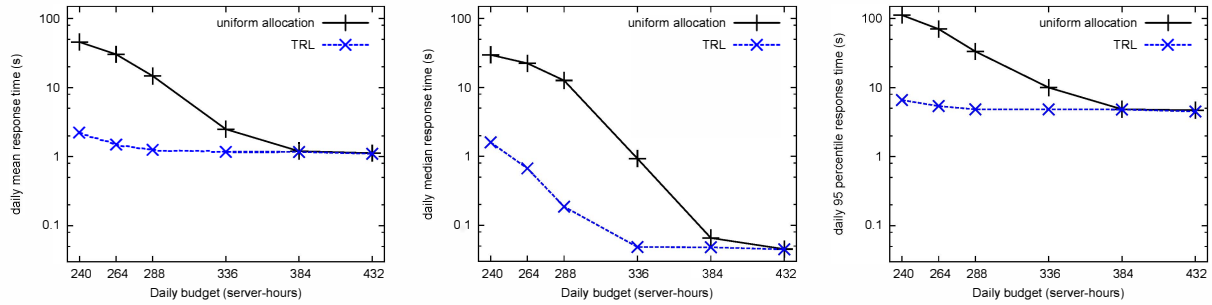
For all the budgets we tested, TRL provides performance at least as good as the uniform allocator across all metrics: mean, median, and 95th percentile response time (see Figure 4).

TRL provides the most significant performance improvement – over an order of magnitude reduction in response time – relative to the uniform allocator when the system is given a small budget. For example, when the uniform allocator is given only 240 server hours it is only able to run 10 servers. During the heavily loaded portion of the day (e.g. hours 38-44 in Figure 3), the uniform allocator system has significant queuing delays. In contrast, TRL allocates fewer of its server hours during periods of low demand, and is thus able to afford to scale up the number of servers for periods of high demand. As a result, TRL is better able to avoid queuing delays, even when demand is high.

For both systems, increasing the budget improves performance for all three metrics – but only to a point. For example, increasing the budget by 40% from 240 to 336 server hours results in an order-of-magnitude reduction in response time for the uniform allocator system. However, increasing the budget from 384 to 432 server hours yields only a very small increase in performance for both systems.

The reduction in marginal performance gains as we increase the budget occurs because we reach the lower bound of these metrics, given our application's characteristics. For example, our average per-tweet processing time is about 0.8 sec, and our daily mean response time plateaus at just over 1 sec (with a budget of 432 server hours). Likewise, the 95th percentile metric curves also flatten out (at about 10 sec) under large budgets because the 5% of the tweets with the longest processing time take at least 10 sec.

The median response time metric provides a rough estimate of the proportion of tweets that are subject to queuing delays. Since the majority of tweets (about 64%) do not have any URLs to be fetched, they complete in less than 1 msec. As a result, when we have low median response times, that means that the majority of tweets are not subject to significant queuing delays. Consequently, for a daily budget of 336 server hours, the uniform allocator system has a median response



(a) Mean (b) Median (c) 95th Percentile

Fig. 4: Mean, median and 95th percentile daily response times for TRL and uniform allocation policies, for various budgets.

time of about 1 second. We expect that the median tweet – one with no URLs to process – should complete very quickly, so we can conclude that the majority of tweets are subject to queuing delays of up to one second. Alternatively, TRL's median performance with a budget of 336 server hours gives a median response time of only 50 msec, implying significantly smaller queuing delays than the uniform allocator.

V. Related work

The mechanism we propose here is motivated by the following maxim "provide more resource *when* resource is needed". The spatial equivalent of the maxim "provide more resource *where* resource is needed" has been utilized in a number of designs (briefly discussed below) that have recently been presented [4], [8], [11], [16], [22], [27], [19]. The key observation is that the spare resource of the lightly-loaded node is worth more on the heavily-loaded nodes.

The topic of cost-effective running cloud based services and data centers has lately attracted significant attention in the networking community, see [10] and references therein. A major concern of these efforts is power control [5] as it accounts for the largest part of the non-fixed (adaptable) expenses [10].

Pricing of service by most cloud providers is usage-based [1], [2], [22]. However, in the history of communications, pricing of various services (e.g. ordinary mail, the telegraph, the telephone, and the Internet) followed a similar pattern: it started with usage-based pricing and converged to some form of flat-fee pricing. Moreover, enterprises tend to prefer fixed cost of an IT service rather than unlimited/unpredictable usage-based cost, see [12] and [17].

VI. Summary

In this paper we propose TRL, a scheme for introducing temporal load awareness in the control of elastic resources leased from a cloud infrastructure. We argue that for the same daily budget, TRL allocation can exhibit significantly better performance than nonelastic load-oblivious schemes. The level of improvement depends on the demand variability as well as the daily budget. TRL's gains peak in the most difficult case of highly varying demand and constrained purchase budget.

Acknowledgments

This work was done while J.S.O was an intern in Telefonica Research, Barcelona. This work and its dissemination efforts have been supported in part by the ENVISION FP7 project of the European Union.

References

[1] Amazon Elastic Compute Cloud: http://aws.amazon.com/ec2.
[2] M. Armbrust et al. "Above the Clouds: A Berkeley View of Cloud Computing". Tech. Report UCB/EECS-2009-28.
[3] L.A. Barroso. U. Hlzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines". Morgan & Claypool, 2009.
[4] R. Chandra et al. "A Case for Adapting Channel Width in Wireless Networks". In Proc. of ACM SIGCOMM 2008.
[5] G. Chen et al."Energy-aware server provisioning and load dispatching for connection-intensive internet services. In Proc. of NSDI 2008.
[6] M. Chiang et al." Network utility maximization with nonconcave, coupled, and reliability-based uilities". In Proc. of SIGMETRICS 2005.
[7] N. Gans et al. "Telephone call centers: Tutorial, review, and research prospects". Manufacturing and Service Operations Management, vol. 5(2) 2003.
[8] D. Giustiniano et al. "Fair WLAN Backhaul Aggregation". In Proc. of ACM MOBICOM 2010.
[9] L. Green, P. Kolesar. "The pointwise stationary approximation for queues with nonstationary arrivals". Management Science, vol. 37(1), 1991.
[10] A. Greenberg et al. "The Cost of a Cloud: Research Problems in Data Center Networks". ACM Computer Communications Review, vol 39(1) 2009.
[11] R. Gummadi, H. Balakrishnan. "Wireless Networks Should Spread Spectrum On Demand". In Proc. of ACM HotNets 2008.
[12] D. Hinchcliffe. "2007: The year enterprises open thier SOAs to the Internet". Enterprise Web 2.0, Jan. 2007.
[13] R. Jain. "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling". John Wiley and Sons, INC., 1991.
[14] N. Laoutaris et al. "Delay Tolerant Bulk Data Transfers on the Internet". In Proc. of ACM SIGMETRICS 2009.
[15] D. MacAskill. "SkyNet: auto-scaling for SmugMug". http://blogs.smugmug.com/don/tag/skynet/.
[16] T. Moscibroda et al. "Load-aware spectrum distribution in wireless LANs". In Proc. of IEEE ICNP 2008.
[17] A. Odlyzko. "Internet pricing and the history of communications". Computer Networks, vol. 36, 2001.
[18] T. Osogami. "Accuracy of measured throughputs and mean response times". In Proc. of MAMA 2007, San Diego, CA, USA.
[19] R. Peterson, E. G. Sirer. "AntFarm: Efficient Content Distribution with Managed Swarms" In Proc. of NSDI 2009.
[20] J. M. Pujol et al. "The Little Engine(s) that Could: Scaling Online Social Networks". Proc. ACM SIGCOMM 2010.
[21] A. Qureshi. "Plugging Into Energy Market Diversity". Hotnets 2008.
[22] B. Raghavan, K. Vishwanath, S. Rambhadran, K. Yocum, A. Snoeren. "Cloud Control with Distributed Rate Limiting". In Proc. of ACM SIGCOMM 2007.
[23] G. Rosen. "Anatomy of an Amazon EC2 Resource ID".
[24] M. Roughan, et al. "Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning". In Proc. of IMW, 2002.
[25] R.Stanojevic, J. S. Otto, N. Laoutaris. "Temporal Rate Limiting: cloud elasticity at a flat fee". Technical report, http://www.cs.northwestern.edu/~jot836/TRL_TR.pdf.
[26] R. Stanojevic, R. Shorten. "Fully decentralized emulation of best-effort and processor sharing queues". In Proc. of ACM SIGMETRICS 2008.
[27] C.W. Tan et al. "A Distributed Throttling Approach for Handling High Bandwidth Aggregates". IEEE Trans. Par. Dist. Systems, vol 18(7), 2007