TTL-based Cloud Caches

Damiano Carra
University of Verona
damiano.carra@univr.it

Giovanni Neglia
Université Côte d'Azur, Inria
giovanni.neglia@inria.fr

Pietro Michiardi
Eurecom
pietro.michiardi@eurecom.fr

***Abstract*—We consider in-memory key-value stores used as caches, and their elastic provisioning in the cloud. The cost associated to such caches not only includes the storage cost, but also the cost due to misses: in fact, the cache miss ratio has a direct impact on the performance perceived by end users, and this directly affects the overall revenues for content providers. Our aim is to adapt dynamically the number of caches based on the traffic pattern, to minimize the overall costs.**

We present a dynamic algorithm for TTL caches whose goal is to obtain close-to-minimal costs. We then propose a practical implementation with limited computational complexity: our scheme requires constant overhead per request independently from the cache size. Using real-world traces collected from the Akamai content delivery network, we show that our solution achieves significant cost savings specially in highly dynamic settings that are likely to require elastic cloud services.

I. Introduction

In-memory key-value stores used as caches are a fundamental building block for a variety of services, including web services and Content Delivery Networks (CDN). With the advent of cloud computing, these services have been offered as managed platforms with a pay-as-you-go model. Amazon's ElastiCache [1] and Microsoft's Azure Redis Cache [2] are examples of caches that employ popular open source software such as Memcached [3] or Redis [4].

Elasticity, *i.e.*, the ability to adapt to workload changes, is a key characteristic of cloud computing: auto-scaling tools, configured by the users, determine the amount of cloud resources to deploy. The techniques used to drive the scaling process have been the subject of many studies in the past—see [5] and the references therein. These studies mainly focus on traditional services, such as computing, where the relation between the performance and the amount of deployed resources is almost linear.

When considering caches, the relation between a key performance index, the hit ratio, and the resources deployed is not linear, *e.g.*, doubling the cache size does not correspond to doubling the hit ratio. The analysis of dynamic adaptation of cloud caches has received little attention: the few studies have focused on minimizing storage costs for a given target hit ratio, ignoring that misses may have different costs and disregarding the possibility to tune the hit ratio itself.

Several studies have highlighted the cost of delay for web services [6], *i.e.*, a direct connection between the response time (or web page load time) and economic losses, for example because the customer does not finalize a purchase. Notice that, even a small increase in the miss ratio (*e.g.*, 1%), often translates into a high variation in the average latency (*e.g.*, 25%) [7]. Misses can also translate to infrastructure costs because of the additional load on back-end databases [8], [9]. Beyond these specific examples, we assume that it is possible to quantify the *cost* due to misses. Then, when sizing cache resource allocation, these costs should be considered.

In this paper we study the dynamic assignment of resources to in-memory data stores used as caches. To this aim, we take into account the cost of the storage *and* the cost of the misses, and we adapt the amount of resources to the traffic pattern minimizing the total cost. We consider an approach based on TTL caches [10], and we study a model in which the Time-To-Live (TTL) is adapted through stochastic approximation iterations and dynamically converges to the best setting. We operate the system using a virtual TTL cache, whose virtual size informs the elastic deployment of cache server instances to manage incoming requests.

High-throughput caches rely on low complexity operations: for instance, key lookups and updates in LRU caches have $O(1)$ complexity per request. This bound is considered a hard requirement for CDNs running on commodity hardware [11]. The auto-scaling tool, therefore, should not have higher complexity, otherwise it may represent a performance bottleneck. For this reason, we design a practical policy to automatically scale-out caches with $O(1)$ complexity per request.

We evaluate such TTL-based solution with a testbed, using real-world traces collected from Akamai, one of the largest Content Delivery Networks, for over than 30 days. We show that our approach can achieve the same savings obtained by adapting previously proposed solutions based on Miss Ratio Curves (MRCs) [12], which are less scalable because they have a per-request computational overhead that grows logarithmically with the cache size.

Contributions: We make the following contributions.

• *TTL-based approach:* We propose and study a dynamic algorithm for TTL caches, which adapts the TTL value to both misses and storage costs minimizing the total operational expenditure (§ IV).

• *Design and implementation of a horizontally scalable TTL-based solution:* We design and implement a system based on the TTL approach, which dynamically adds and removes cache instances in order to maintain the total cost at minimum. We pay particular attention to system scalability, and provide a $O(1)$ solution targeted at high-throughput caches (§ V).

• *Evaluation:* We evaluate the TTL-based solution in our testbed with real-world traces from Akamai, and show that

is able to track the optimal cache configuration and achieve significantly cost savings specially in highly dynamic settings (§ VI).

II. Background and problem definition

In-memory data stores. In-memory key-value stores represent a fundamental piece of web architectures. They are used to cache popular contents, so that the web application can access quickly the frequently requested data – see for instance the architecture deployed at Facebook [13].

Widely used in-memory stores are Memcached [3] and Redis [4], whose APIs allow setting a key-value entry, or retrieving the value given a key. If the cache is full and a new content needs to be inserted, both systems employ slight variations of the Least Recently Used policy (LRU). The amount of RAM assigned to Memcached or Redis instances is set when the instance is created, and cannot be changed at runtime. In order to achieve vertical scalability—*i.e.*, changing the amount of memory at runtime—the only option is to create a new instance with the desired amount of memory and transfer the content from the old instance to the new one. Since this approach requires time and resources, vertical scalability is usually not considered practical.

On the other hand, horizontal scalability is easy to achieve. Instances can be added to (or removed from) a cluster of nodes, with a *load balancer* tool (such as `mcrouter` [14]) that manages all aspects related to distributed caches: data placement and request routing, possibly data replication and instance failure management. In this paper, we consider the basic scenario where the content is not replicated across instances and one load balancer is sufficient for managing the cluster. The results can be easily extended to any replication factor the user may decide to adopt.

Elastic on-demand services. Cloud computing enables services to be instantiated on demand, according to traffic volume. In the case of web architectures, for instance, one can augment the number of servers to accommodate increasing traffic. Service providers have recently included, among the different services, in-memory data stores used as caches. Prominent examples are Amazon's ElastiCache [1], Microsoft's Azure Redis Cache [2] and Google's Cloud Memorystore [15]. These managed solutions take care of the details of the caches, such as software update and maintenance, and provide simple APIs to create and shut down instances, and manage the corresponding cluster of such instances.

The user can choose among a set of possible configurations for each instance. For example, Amazon's ElastiCache [16] allows the customer to choose among instances with different RAM sizes and numbers of cores (vCPUs). Different types of instances are also available, like regular, spot and burstable ones. The latter two types refer to instances whose capacity may be changed (reclaimed) by Amazon. Here, we consider regular instances.

Problem definition. In this work, we focus on the caches, without considering the other elements of the cloud caching service, such as the web server, the back-end databases or the origin server if the cache is part of a CDN. Our aim is to adapt over time the total cache size to the content request pattern in order to minimize the total cost, that is the sum of storage cost and cost due to the misses.

The storage cost is immediate to evaluate, because it is determined by the pricing scheme of the cloud provider (we provide later specific examples for Amazon ElastiCache service). The provider offers different possible configurations with different costs. As a design choice, when scaling horizontally, we focus on homogenous instances. Since the cost model of the service providers usually has a specific granularity (typically, one hour), we consider fixed intervals that we call epochs, and the choice of changing the number of instances is done at the end of each epoch. Let $I(h)$ be the number of instances selected during the h-th epoch and c^s be the cost of one instance. The storage cost over the first k epochs is then

$$C^s(1,k) = \sum_{h=1}^{k} c^s I(h).$$

The other component of the total cost is due to misses. The cost of a miss can correspond to the additional delay experienced by the final user or to the additional load on the origin server, *e.g.* in terms of number of requests or bytes to serve. In this work, we assume that the service provider has the ability to quantify monetarily the miss cost. For example, there are several studies on the connection between delay and revenues [6]. We denote by m_o the miss cost for object o, and we assume it to be deterministic and constant over time (our theoretical results can easily be extended to the case when miss costs for each object are i.i.d. random variables). Let $r(n)$ be the object requested by the n-th miss. With some abuse of notation, we let $n \in [k_1, k_2]$ denote that the n-th miss occurred in the time interval corresponding to the epochs k_1, k_1+1, $\dots k_2$, with $k_1 < k_2$. The total miss cost per time unit during the first k epochs is then

$$C^m(1,k) = \sum_{n\in[1,k]} m_{r(n)}.$$

Our goal is to select the number of instances $I(1)$, $I(2)$, $\dots$ $I(k)$, in order to minimize the total (storage + miss) cost. The tradeoff is evident. At any epoch, a larger number of instances decreases the number of misses—and therefore the corresponding cost—but it causes a higher storage cost. Conversely, a smaller number of instances increases the cost due to misses, but it decreases the storage cost. In what follows we present a policy that, at the end of each epoch, determines the number of instances to allocate such that the (expected) total cost for the next epoch is minimal.

On the complexity of the solution. In order to deliver high throughput, caches require small processing overheads, *i.e.*, $O(1)$ time complexity *per request*, which is considered a hard requirement for CDN caches [11]. At high request rates, more complex operations can pose an intolerable load on the CPU causing spurious misses [17], *i.e.* a requested content may not be served even if present in the cache. Eviction policies

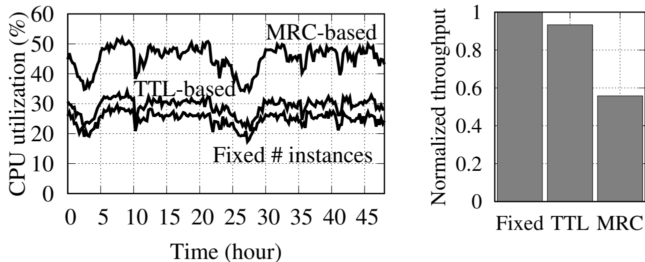

Fig. 1: Left: CPU load using a fixed number of instances routing scheme, our TTL-based solution (both with $O(1)$ time complexity) and an MRC-based solution (with $O(\log M)$ time complexity). Right: Throughput normalized to the fixed scheme case.

with $O(\log M)$ time complexity per request (where M is the number of objects in the cache) are unpractical, since they pose high burden on the CPU, as shown also in [18].

Not only the eviction policy, but any operation related to the cache—including load balancing and cache resizing—needs to have $O(1)$ time complexity. To show the impact of the computational complexity on the system, we set up an experiment focusing on the load balancer. Using the traces described in Sect. VI, we compare a basic scenario—a fixed number of cache instances, where the load balancer simply routes requests—with two improved load balancers based *i)* on our TTL-based solution, which has $O(1)$ time complexity, and *ii)* on a MRC-based solution, which has $O(\log M)$ time complexity (see Sect. III for a discussion on the complexity of the MRC computation).

Figure 1, left, shows the CPU load over time for two representative days when we replay the requests following the timestamps provided in the trace. The MRC-based solution leads to almost double the CPU usage compared to the basic scenario, as well as to our TTL solution. Figure 1, right, shows the maximum throughput (normalized w.r.t. the basic scenario) achievable by the different schemes when the requests are backlogged. While our TTL solution experiences about 8% throughput reduction due to the additional data structure we maintain, the MRC solution almost halves the achievable throughput.

III. Related work

Elastic resource provisioning of cloud services has been the subject of many studies. The authors in [5] provide a general overview of the techniques, such as control theory as used in [19]. Despite the broad set of results, they are computationally too expensive, and it is not clear if they can be applied in the context we consider, where the relation between the resource deployed and the key performance index (the hit ratio) is not linear. Moreover, none of them uses stochastic approximation for resource allocation as we do. Another prominent example of a general approach for auto-scaling is given by [20] but the proposed solution is based on methods (*e.g.*, time series prediction) that are too computationally intensive for the high-throughput scenario we consider.

Memory management is related to our problem but it rather aims to determine how the available memory should be shared among competing applications. Moreover, the proposed solutions, such as [7] [21] [22], all require computations with higher complexity than our approach.

As for minimizing costs in a cloud computing environment, the authors in [23] and [24] explore the use of spot instances for different aims, such as content replication or decreasing the overall storage cost. Beside their computational complexity, the proposed schemes do not take into consideration the cost due to misses, as we do. The authors in [24] also consider a policy for modulating the allocation of on-demand instances to match dynamic needs, but they do not describe it in detail.

Part of our TTL-based solution is based on the concept of a virtual cache, which maintains the metadata of cacheable objects, but not their content. These metadata are sometimes referred to as *ghosts*. This additional information is used in many caching schemes (*e.g.*, in the popular ARC [18]) to decide how to manage the objects in the physical cache. Instead, we use a virtual cache to size (multiple instances of) the physical one. A recent work [25] also explores how to adapt the TTL value to the request pattern by using stochastic approximation. In particular, the authors focus on vertical scaling and aim to achieve a target hit ratio, possibly with a small cache size. On the contrary, our approach addresses horizontal scaling to minimize the total operational cost.

MRC-based solutions. Miss Ratio Curves (MRCs) are a well-known tool for cache profiling [12]: in a single graph it is possible to observe the relation between cache size and miss ratio, therefore one can compute the storage cost and estimate the cost of the misses for each possible cache size. The main issue with MRCs is their computational complexity: the state-of-the-art algorithm proposed by Olken [26] relies on a tree data structure and it has $O(\log M)$ operations per request.

In order to achieve the target $O(1)$-complexity per request, many solutions have been proposed in the literature that compute approximate MRCs [12] [27] [28]. Such solutions share a common characteristic: they have been designed considering objects with uniform sizes. On the contrary, many caching applications exhibit contents with very different sizes.

In order to show the impact of heterogeneous content size on the accuracy of the approximate computation, we set up a simple test. We consider the method proposed in [28], which is based on sampling (but the others operate in a similar way). We use an IRM trace for which the distributions of object popularities and object sizes are reported in Fig. 3.

For each request, besides the timestamp and the object identifier, we have the object size. First, we ignore the actual object size and assume it to be uniform. In this case, as observed in [28], the method predicts the MRC with a prediction error smaller than $3 * 10^{-3}$ for different sampling rates.[1] When we

[1] The error is evaluated by measuring the absolute difference between the exact and the approximated MRCs over all meaningful cache sizes, and then by computing the mean of these absolute differences.

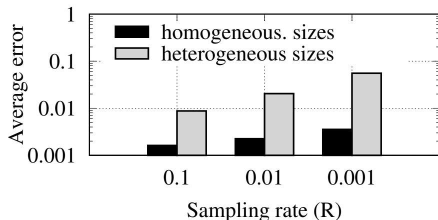

Fig. 2: Accuracy of the approximate MRC computation through sampling, with uniform and nonuniform object sizes.

consider the object sizes, the error increases by one order of magnitude – note that we took particular care in adapting the approximate MRC computation in [28] to the heterogenous case, but it seems that more sophisticated sampling methods are required.

In summary, the approximate computation of the MRC with $O(1)$ time complexity per request still needs to be studied in depth, especially when contents have different sizes. Thus, the only option is to compute the MRCs exactly, which has $O(\log M)$ complexity per request.

IV. Adaptive TTL based solutions

In this section, we begin with a key building block for the design of a horizontally scalable caching system. Our work draws inspiration from Time-To-Live (TTL) caches, *i.e.* caches that are managed by a TTL policy. There are two families of TTL policy: with and without renewal. In both cases, upon a miss, the content is stored locally and a timer with duration equal to T is activated and the content is evicted when the timer expires. The difference is that, in the case with renewal, the timer is reset by the following hits for the content, while it is not affected by them in the case without renewal. TTL caches are a natural model for DNS caches, but they have also been proposed as an approximate model to study the performance of existing replacement policies like LRU [29]. Moreover, different papers have suggested their practical use because of their higher configurability as well as amenability to analysis [10], [25], [30]. While a replacement policy maintains in the cache as many contents as the available space buffer allows (contents are evicted only if needed to make space), under a TTL policy the actual storage vary over time and is, in theory, potentially unbounded. A real implementation of a TTL cache will have a finite capacity and then it may need to evict some contents even if their timer has not expired yet. Some of these practical issues are discussed in [10]. In our solution a TTL cache with renewal is used as a virtual cache, storing only content metadata:[2] by computing its virtual size, our approach steers the addition or removal of cache server instances.

[2]For some contents the metadata can have a size comparable with the content itself, but overall in our experiment the total storage required by the virtual cache was negligible.

A. Dynamic adaptation

We present an adaptive mechanism based on stochastic approximation by which the timer value converges to the value that minimizes the total cost.

The theoretical results hold in the following scenario. We consider a finite catalogue with N contents and that requests for the different contents occur according to independent renewal processes. We denote by λ_i the request rate for content i. When the processes are Poisson ones, a given request will be for content i with probability $\lambda_i / \sum_{j=1}^{N} \lambda_j$ independently from any previous request. This is (a continuous version of) the well known Independent Reference Model (IRM) [31].

In what follows, we consider an ideal TTL cache with renewal and assume that the cloud service charges the user only for the instantaneous storage occupancy. This differs from the more realistic scenario described above where the user needs to pay for the instances independently from their usage, but we will come back to practical billing in Sect. V-A. Let s_i be the size of object i and c be the cost per unit time to store a unit of content ([24] shows that prices are almost linear also for real cloud services). Then, the total cost to store content i over a time window of duration τ is $c s_i \tau$. For simplicity, we denote $c_i = s_i c$. A miss for content i incurs a cost equal to m_i.

Let $X_i(t)$ be the indicator function for the event "content i is stored in the cache at time t" and $M_i(t)$ the counting process of content i misses in the interval $[0, t]$. We can define the storage cost and the miss cost analogously to what done in Sect. II. The total cost over the interval $[0, t]$ is then

$$\begin{aligned} C(0,t) &= C^s(0,t) + C^m(0,t) \\ &= \sum_{i=1}^{N} \int_0^t X_i(\tau) c_i \, d\tau + M_i(t) m_i. \end{aligned} \tag{1}$$

If the caching policy uses a constant TTL value equal to T, then each process $X_i(t)$ is a renewal process whose regeneration points are the time instants at which misses of content i occur. The renewal reward theorem guarantees that, for each content, the time-average cost is equal to the expected cost over a renewal period divided by the expected duration of a renewal period, *i.e.*

$$\lim_{t \to \infty} \frac{\int_0^t X_i(t) c_i \, dt + M_i(t) m_i}{t} = \frac{c_i \tau_{S,i} + m_i}{\tau_{M,i}},$$

where $\tau_{S,i}$ is the expected sojourn time of content i in the cache and $\tau_{M,i}$ is the expected time between two misses.

The asymptotic time average cost ($\mathcal{C}$) of the system as a function of T is then

$$\mathcal{C}(T) = \lim_{t \to \infty} \frac{C(0,t)}{t} = \sum_{i=1}^{N} \frac{c_i \tau_{S,i} + m_i}{\tau_{M,i}}. \tag{2}$$

We observe that $\tau_{S,i}/\tau_{M,i}$ is the asymptotic fraction of time content i spends in the cache or equivalently, the probability that content i is in the cache at a random time, that is often called the occupancy probability and we will denote it by o_i.

The inverse of $\tau_{M,i}$ is the rate at which misses occur that we can also write as $\lambda_i(1-h_i)$, where h_i is the hit ratio, *i.e.* the fraction of requests for content i that incurs a hit. Then we can rewrite (2) as

$$\mathcal{C}(T) = \sum_{i=1}^{N} c_i o_i + \lambda_i m_i (1-h_i). \qquad (3)$$

When arrivals follow a Poisson process, it holds $o_i = h_i$ because of PASTA property and moreover $h_i = 1 - e^{-\lambda_i T}$ [10]. Then, (3) becomes

$$\mathcal{C}(T) = \sum_{i=1}^{N} c_i + (\lambda_i m_i - c_i) e^{-\lambda_i T}. \qquad (4)$$

We can check that if $T = 0$, *i.e.* no content is stored in the cache, the cost per time unit is equal to $\sum_{i=1}^{N} \lambda_i m_i$: we pay systematically for all misses. Instead, if $T_i = \infty$, all contents are stored indefinitely and the corresponding cost per time unit is $\sum_{i=1}^{N} c_i$.

We could look for the value T^* that minimizes the cost (4) by applying a gradient algorithm as follows:

$$\begin{aligned} T(n+1) &= T(n) - \epsilon(n) \frac{d\mathcal{C}}{dT}\Big|_{T(n)} \\ &= T(n) + \epsilon(n) \sum_{i=1}^{N} \lambda_i e^{-\lambda_i T(n)} \left(\lambda_i m_i - c_i\right), \end{aligned}$$

where the sequence $\epsilon(n)$ converges to zero as n diverges, but it is not summable, *i.e.* $\sum_{n\in\mathbb{N}} \epsilon(n) = \infty$. This approach is not viable because in a realistic scenario popularities are unknown, keep changing over time and are not easy to estimate. The gradient algorithm suggests us a practical solution based on stochastic approximation [32]. We observe that $\lambda_i e^{-\lambda_i T} = \lambda_i(1-h_i)$ is equal to the miss rate for content i. Upon a miss, this is for content i with probability proportional to $\lambda_i(1-h_i)$. Let $r(n)$ be the object requested at the n-th miss and $\hat{\lambda}_i(n)$ be a random unbiased estimate of the arrival rate λ_i. Consider the following update rule for the variable $T(n)$:

$$T(n+1) = T(n) + \epsilon(n) \left(\hat{\lambda}_{r(n)} m_{r(n)} - c_{r(n)}\right), \qquad (5)$$

where the correction term $\hat{\lambda}_{r(n)} m_{r(n)} - c_{r(n)}$ is a random variable because *i)* content requests occur according to IRM and *ii)* the estimator itself is a random variable. The correction corresponds "in average" to the gradient $d\mathcal{C}/dT$ because, upon a miss, the fraction of requests for content i is proportional to $\lambda_i e^{-\lambda_i T(n)}$, and then $\mathbb{E}(\hat{\lambda}_i m_i - c_i) = \lambda_i m_i - c_i$. The following proposition makes this result formal.

Proposition 1. *Let* $\{X(n, T(n))\}$ *be a sequence of independent random variables such that* $X(n, T(n))$ *is equal to* $\hat{\lambda}_i m_i - c_i$ *with probability* $\lambda_i e^{-\lambda_i T(n)} / (\sum_{j=1}^{N} \lambda_j e^{-\lambda_j T(n)})$. *Let* $\{\epsilon(n)\}$ *be a non-negative sequence converging to* 0, *such that* $\sum_{n\in\mathbb{N}} \epsilon(n) = \infty$ *and* $\sum_{n\in\mathbb{N}} \epsilon^2(n) < \infty$. *Consider the update rule*

$$T(n+1) = \Pi_{[0,T_{\max}]}(T(n) + \epsilon(n) X(n, T(n))),$$

where $\Pi_{[0,T_{\max}]}(x) = \min(\max(0,x), T_{\max})$ *is the projection operator over the interval* $[0, T_{\max}]$, *then the sequence* $T(n)$ *converges with probability one to i) a stationary point of* $\mathcal{C}(T)$ *or ii)* 0 *or* $T_{\max}$, *if* 0 *and* $T_{\max}$ *are local minima of* $\mathcal{C}(T)$.

Proof. The result follows from Theorem 2.1 in [32]. All hypotheses (A2.1)$-$(A2.7) are satisfied with $f(.) = \mathcal{C}(.)$. □

If, instead of letting the weights $\epsilon(n)$ converge to zero, we keep them equal to a small constant value ϵ_0, then, in a stationary setting, $T(n)$ converges to a neighbourhood of the limits indicated in Proposition 1. Note that a constant weight makes it possible to track changes in the system, for example when popularities keep varying over time.

B. An optimal clairvoyant TTL Policy

In this section we present the optimal TTL policy (referred to as TTL-OPT), that minimizes the total cost when the sequence of future requests is known. The cost achieved by this clairvoyant policy is clearly a lower-bound for any feasible policy. Among the TTL policies, TTL-OPT plays the same role as Bélády's algorithm [33] for replacement policies. Indeed, Bélády's algorithm minimizes the miss ratio under knowledge of the future requests and uniform content sizes. Interestingly, the optimal clairvoyant TTL policy has polynomial complexity under heterogeneous content sizes and miss costs, while in such case finding an optimal replacement policy is an NP-hard problem [34] (and Bélády's policy is no more optimal).

Algorithm 1: Optimal Clairvoyant TTL policy (TTL-OPT)

input : $\{c_i\}$, storage costs per unit of time, and $\{m_i\}$, miss costs
input : request sequence

```
1 foreach request r do
2     j ← obj id of request r
3     t_{j,next} ← time of the next request for obj j
4     c_j^S ← c_j × (t_{j,next} − t_now)
5     if (c_j^S < m_j) then
6         T_j ← t_{j,next} − t_now // store j until its next request
7     else
8         T_j ← 0 // do not store j
```

We allow the optimal policy TTL-OPT to select a different TTL value for each content and for each request. The policy is described in Algorithm 1 and is very simple: given a request for a content, say j, at time t_{now}, if the cost to store the content until its next request (at time $t_{j,\text{next}}$) is smaller than the cost of a miss for this object, then the content should be stored in the cache until the next request, *i.e.* the timer should be set equal to $t_{j,\text{next}} - t_{\text{now}}$. Otherwise, the object should be served but not stored. The formal proof of optimality for TTL-OPT follows.

Proposition 2. *The clairvoyant policy TTL-OPT in Algorithm 1 minimizes the sum of storage and miss costs.*

Proof. Let $C_i(0,t)$ denote the total cost paid during the interval $[0,t]$ for content i, i.e. $C_i(0,t) = \int_0^t X_i(t) c_i \, \mathrm{d}t + M_i(t) m_i$.

The total cost $C(0,t)$ in (1) is then given by the sum of the costs for each content. The possibility to choose the timer value independently for each content reduces the minimization of the total cost $C(0,t)$ to separately minimize each term $C_i(0,t)$. Let $\{t_{i,k}, k \in \mathbb{N}\}$ be the sequence of time instants of the requests for content i. A TTL policy needs to select a TTL value for each request, let us denote as $T_{i,k}$ the timer for the k-th request occurring at time $t_{i,k}$. We observe that we can restrict ourselves to consider $T_{i,k} \in \{0, t_{i,k+1} - t_{i,k}\}$. In fact, consider any sequence of timer values $\{\hat{T}_{i,k}, k \in \mathbb{N}\}$, and let $\hat{T}_{i,h}$ be a timer such that $\hat{T}_{i,h} < t_{i,h+1} - t_{i,h}$. If we replace $\hat{T}_{i,h}$ with $T_{i,h} = 0$, the cost $C_i(0,t)$ cannot increase. Similarly, we can replace any value $\hat{T}_{i,h}$ such that $\hat{T}_{i,h} > t_{i,h+1} - t_{i,h}$ with $T_{i,h} = t_{i,h+1} - t_{i,h}$, without increasing the cost $C_i(0,t)$. Let then $Z_{i,k}$ be an indicator function such that $Z_{i,k} = 1$ if $T_{i,k} = t_{i,k+1} - t_{i,k}$, and $Z_{i,k} = 0$ if $T_{i,k} = 0$. The total cost for content i can then be rewritten as follows:

$$C_i(0,t_{i,k}) = m_i + \sum_{h=0}^{k-1} \left(Z_{i,h} c_i(t_{i,h+1} - t_{i,h}) + (1 - Z_{i,h}) m_i \right), \tag{6}$$

where the first term on the right hand side corresponds to the fact that the first request for content i generates always a miss. From (6) if follows that $C_i(0,t_{i,k})$ is minimized by choosing $Z_{i,h} = 1$ if $c_i(t_{i,h+1} - t_{i,h}) < m_i$ and $Z_{i,h} = 0$ otherwise. This corresponds to what TTL-OPT does. □

Clearly, the TTL-OPT policy cannot be used online. Nevertheless, given a trace, its cost can be computed (in polynomial time) and used as a reference.

V. Implementation

A. Practical implementation of the TTL-based scheme

In what follows we progressively introduce some practical issues one needs to address to implement the TTL-policy.

When to estimate the request rates. A straight application of (5) would require to update the timer immediately upon a miss, and then popularity estimates should be available for contents that are not in the cache. Instead, we will start estimating content popularity immediately after the content is stored in the cache and we will then postpone the timer update to the moment when the estimate is available. The detailed description follows. Let $T(t)$ be the timer value at time t. If the timer is updated at t, then we denote as $T^-(t)$ the value immediately before the update. Updates are, as above, driven by misses, and we denote as t_n the time of the n-th miss and $r(n)$ the corresponding content. Upon a miss, content $r(n)$ is stored and its timer is set to the current value $T(t_n)$. Any new request for content $r(n)$ before the timer expiration will be a hit and will reset the timer to $T(t_n)$. The number of hits for content $r(n)$ during the interval $[t_n, t_n + T(t_n)]$ is recorded. Let us denote this number as $H_{r(n)}$. The ratio $H_{r(n)}/T(t_n)$ is an unbiased estimator of the rate $\lambda_{r(n)}$. Once this estimate is available at time $t_n + T(t_n)$, the timer is updated as follows:

$$\begin{aligned} T(t_n+T(t_n)) &= T^-(t_n + T(t_n)) \\ &\quad + \epsilon(n) \left(-c_{r(n)} + \frac{H_{r(n)}}{T_{r(n)}} m_{r(n)} \right). \end{aligned} \tag{7}$$

We observe that $T^-(t_n+T(t_n))$ may be different from $T(t_n)$, since the timer may have been updated during $[t_n, t_n+T(t_n)]$ as effect of misses for contents other than $r(n)$.

When to update the timers. As a further refinement, we notice that the cache is driven by two main events: request arrival and object eviction. The updates of the timer should be done when these events occur. This adds an additional small delay: given a content, the TTL update is triggered by the hit after the first timer or, if no hit occurs after this time, by the content eviction. With the above modifications, we observe that Proposition 1 does not hold, since *i)* the different updates are not independent and identically distributed (conditioned on the current timer value), and *ii)* the update delays could in principle affect convergence. There are theoretical results for stochastic approximation algorithms when the correction terms are correlated and when updates are delayed, but we let the study of convergence for further investigation.

The timers are used as keys for organizing the metadata according to the approach proposed in [9], that manages a partially ordered data structure with O(1) complexity.

B. Horizontally scalable cache system

The TTL-based scheme discussed so far considers a single TTL cache, whose billing is based on its instantaneous storage. In other words, we have considered a perfect vertically-scalable system, where memory resources can be smoothly added and removed. We now discuss the design of a practical horizontally-scalable system inspired by the TTL-based approach, where storage can only change at finite epochs.

In a horizontally-scalable solution, cache instances can be added or removed from the cluster, and all instances have the same configuration. The first design choice to face is the configuration of a generic instance.

Cache instances. These are the physical caches storing the contents and have fixed size. They can be implemented using Memcached or Redis with a simple eviction policy like LRU.

Load balancer. The load balancer performs the ordinary operations, such as request routing, and content insertion, *i.e.*, in case of a miss, after retrieving the object from the origin or the back-end, it stores it in one of the cache instances. In addition, the load balancer maintains a virtual cache, with the references of the requested objects: this virtual cache is going to be managed as a TTL cache according to the description in Sect. V-A with $O(1)$ computational cost per request. The size of the virtual cache depends on the timer value T, which in turn depends on the number of hits and misses, and on the corresponding costs for the storage and for the misses. Thus, the size of the virtual cache can be used to determine the number of actual instances to employ in the cluster.

Operation. Our scheme is described in Algorithm 2. At every request, we look for the object key in the virtual cache, update

its position in case of a hit, or add it in case of a miss. Then, we start evicting objects from the virtual cache if they are expired. While inserting a new object or removing expired objects, we update the total size of the cache (the sum of the sizes of non-expired objects). Clearly, object sizes can be heterogeneous. At the end of the epoch (line 7), we look at the size of the virtual cache and we select the number of instances such that the sum of the sizes is the closest to the virtual cache size (line 8). At the end of the observation interval, if the number of instances has changed, the load balancer reassigns the responsibility of the hash space to the current instances.

Algorithm 2: TTL-based scaling

input : VC, Virtual Cache
input : S_p, Physical cache size
output: $I(k+1)$, # of the instances in $k+1$-th epoch

1 **foreach** *request* r **do**
2   **if** *($r \in VC$)* **then**
3   REMOVE(r, VC)
4     $r.expire \leftarrow t_{\text{now}} + TTL_{\text{now}}$
5   INSERT(r, VC)
6   EVICTEXPIRED(VC)
7 **if** *(epoch k ended)* **then**
8     $I(k+1) \leftarrow$ ROUND(VC.size / S_p)

Additional considerations. We observe that, when cloud instances are added or removed, the object key space responsibility must be rearranged, which may lead to spurious misses due to route changes. We have experimentally observed that, since the number of requests within an epoch is usually very high, the effect of such spurious misses is negligible.

VI. EXPERIMENTAL EVALUATION

Testbed. We evaluate our approach using a testbed that is representative of a typical web architecture. An application server is connected to a database and to a cluster of caches. The application receives the requests and checks if the content is stored in the cache. If the content is not in the cache, the application server retrieves the object from the database, serves the client and stores the object in the cache. For the operations related to the cache (*e.g.*, object lookup), the application server relies on a *load balancer*. We have implemented the scheme described in Sect. V-B in a custom tool similar to `mcrouter` [14]. The tool is able to add or remove the cache instances from a local cluster, but it can be easily extended to use the APIs of any cloud cache provider.

Trace. The requests sent to the application server are generated by reproducing two anonymized traces collected respectively for 30 and 5 days at two vantage points of the Akamai CDN. The traces we tested contain the timestamp of the request arrival, the anonymized object ID and the size of the requested object. We report the results for the 30-day trace, since we obtain similar qualitative results with the 5-day trace. In the 30-day trace, there are $2 \cdot 10^9$ requests for 110 millions contents, whose size varies from a few bytes to tens of MB. Figure 3 (left-hand side) shows the number of requests for each object, sorted by rank (in terms of popularity). The right-hand side shows the empirical Cumulative Distribution Function (CDF) for the size of the requested objects (without aggregating requests for the same object).

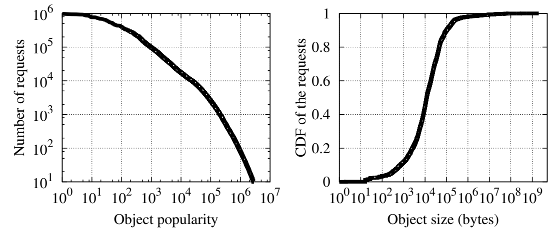

Fig. 3: Number of requests per object, ordered by rank (left), and cumulative fraction of the requests for objects up to a given size (right).

Settings. For the configurations and the costs, we refer to Amazon ElastiCache service [16]. For the duration of the epoch, we consider the minimum billing time, which is one hour. Among the different instances' options, we selected the *cache.t2.micro* with 0.555 GB RAM and one vCPU, which costs 0.017$/hour (Jan. 2018, US based). We use a small instance since it provides a fine granularity when we resize the cluster: our experimental results shows that one small instance is sufficient during low traffic periods. Moreover, bigger instances (*e.g.*, with 3.22 GB or 6.05 GB) have just two vCPUs, which may limit the cache throughput. Replicating small instances each with a vCPU helps in maintaining the throughput while scaling the cluster. As for the cache, we tested both Redis and Memcached: even if they are both able to handle heterogeneous object sizes, we report the results for Redis, since Memcached provides slightly worse performance due to calcification [21], [22].

In order to determine plausible miss costs, we reasoned as follows. The production server from which our trace was collected had an in-memory cache of 4 GB, *i.e.* roughly corresponding to eight *cache.t2.micro* instances. We assume that this system has been engineered so that storage and miss costs are equal, a reasonable rule of thumb to achieve a small total cost. The storage cost can be determined in our case considering the corresponding hourly cost of eight *cache.t2.micro* instances. By dividing this cost by the average number of misses observed during one hour in production, we obtain the cost per miss (in our case, 1.4676 $\times 10^{-7}$\$ per miss). Below we also evaluate the effect of different miss costs.

Previous solutions. Because of the considerations above, we consider as baseline a scenario with eight *cache.t2.micro* instances. We compare also our results with an elastic resource allocation scenario driven by the MRC-approach, as described in [12] and discussed in Sect. III. In addition, as a reference, we consider the scenario with an ideal, vertically scalable, TTL cache, billed according to its instantaneous size. A practical TTL-based policy can approach the performance of this scheme only if *i)* billing periods become arbitrarily small, and *ii)* caches of any size can be rented.

Results. We present here the results for the trace described

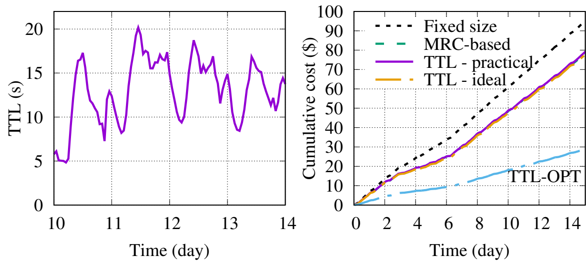

Fig. 4: TTL over time of the virtual cache (left), and cumulative cost of different policies (right).

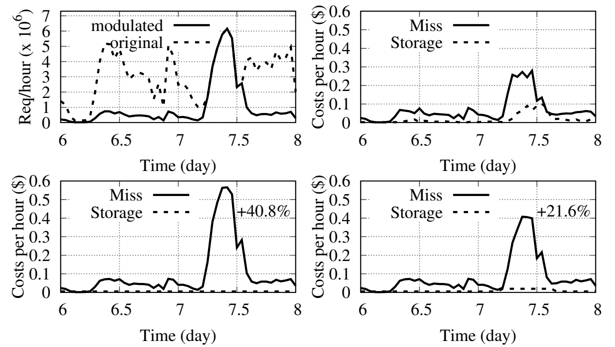

Fig. 5: Traffic increase at day 7 (increase factor: 7). Number of requests (top left). Storage and miss costs with TTL (top right), with fixed number instances when the event is not known in advance (bottom left) and when it is (bottom right).

above. We have also performed an extensive study using synthetic traces generated according to the IRM model—which is the arrival pattern for which the theoretical results in Proposition 1 hold. In such experiments, it is possible to see that the TTL indeed reaches a stable value, which corresponds to the minimum cost.

With a real trace, the arrival pattern varies over time. Our TTL approach continuously tracks such a variation: this is shown in Fig. 4 (left), where we plot the value of the TTL for an interval of four representative days: the evolution clearly follows a daily pattern. The TTL is mirrored by the virtual cache size (not shown here), which varies from zero (the cost of the few misses does not justify the storage of the object) to 3.5 GB. The virtual cache size translates into the number of instances used in the cluster. From this, it is possible to compute the total cost for storage and misses. In Fig. 4 (right) we show the cumulative costs for the first 15 days for the TTL-based system, and we compare it with a 8-instance fixed-size cache (corresponding to our reference in-memory production cache) and the MRC-based approach. The figure plots also the total cumulative cost of an ideal TTL-cache. The results show that the TTL-based approach obtains similar cumulative costs (indistinguishable in the figure) as the MRC-based approach, but with a $O(1)$ complexity instead of $O(\log M)$ complexity. Overall, with respect to the baseline fixed-size approach, the TTL-based approach is able to save 17% of the costs.

There is a slight difference (less than 2%) between the ideal and the practical TTL-based implementation due to the discretization of cache sizes and of billing periods, and to the spurious misses caused by object key reallocation. Interestingly, this result suggests that, at least for typical CDN applications, there is no need for finer-grained billing periods or cache sizes, but most of the potential improvement is already achievable with the current offer.

Figure 4 (right) shows also the results of the clairvoyant TTL-OPT described in Sect IV-B. We see that there is room for even more significant cost savings: TTL-OPT achieves a cost that is one third of the baseline. TTL-OPT assumes to know the sequence of future requests and is thus unpractical. Nevertheless, this result suggests that potential improvements can come from TTL policies that use different TTL values for different contents (as TTL-OPT does) selecting the timer value on the basis of a forecast for the next inter-arrival time. In the future we plan to investigate this possibility.

Sensitivity analysis. The update of the TTL is based on (7), which contains the weights $\epsilon(n)$. As discussed in Sect. IV-A we keep them equal to a small constant value ϵ_0. We have experimentally verified that ϵ_0 may vary by 4 orders of magnitudes with negligible effects (for the experiments shown here $\epsilon_0 = 10^{-4}$). We also tested the sensitivity to the miss costs, by scaling them by a factor γ in comparison to the reference value computed as described above. The TTL-based approach consistently achieves almost the same cost of the more complex MRC-based approach (results not shown here because of space constraints).

A highly dynamic scenario: the Super Bowl case. Cloud-based services are particularly adapted to time-limited and highly dynamic settings for which the costs to deploy and manage an ad hoc infrastructure would not be justified. Our traces do not correspond to such a scenario. The CDN from which the trace were collected has been indeed engineered to satisfy long-term service level agreements with a given number of content providers. Moreover, the traces have been collected directly at a cache that is located behind a load balancer that tries to keep the request rate at a given cache as uniform as possible. We were not able to find real traces representative of a highly dynamic request scenario. We decided then to gauge part of our traces to qualitatively reproduce the traffic variability at a large-scale event as the Super Bowl. To this aim, we consider the wireless data traffic generated by the attendees of the Super Bowl XLVII as described in [35]. In particular the authors of [35] mention that data traffic at the stadium increased by a factor of seven during an interval of about 8 hours (from a couple of hours before the beginning of the game until midnight). To reproduce such a scenario, we consider 8 hours of our 30-day trace, and we sample the traffic before and after this interval, such that the traffic during the 8-hour interval is seven time larger than the average. Figure 5 (top-left) reports the original traffic pattern, and the one shaped to mymic the Super Bowl one.

We then compare our dynamic TTL-based configuration

with fixed static configurations in two different scenarios. In the first scenario, the large-scale event is unexpected and then a static cache system is (optimally) sized on the basis of the usual traffic (seven times less than the Super Bowl peak). In the second scenario instead, the occurrence of the event is known and a cache is instantiated for its duration and sized on the basis of a traffic forecast that underestimate the peak traffic by 80%. We derive also this forecast error from [35]. In fact, the data traffic of Super Bowl XLVII exceeded that of the previous edition by 80% also due to a half-hour power outage that caused the game to be suspended, and then people to spend more time on their mobile (the wireless network was not affected by the outage). The costs of our adaptive TTL solution are shown in Fig. 5 (top-right). The figure shows how the number of instances changes during the game period in order to amortize the cost due to misses. The corresponding plots for the two static configuration scenarios are in the bottom part of the figure. A static configuration incurs a total cost 40% larger than the TTL dynamic configuration when the event is not known in advance and 21% larger when the cache is sized on the basis of the previous edition of the Super Bowl. In summary, while a fixed number of instances can be engineered based on past traffic, unexpected events may put a burden on the caching system. Our dynamic TTL approach is able to adapt to these sudden changes, and it provides the minimum cost – we computed the cost also with a MRC-based approach, obtaining similar results – still maintaining the computational complexity low. We also observe that the TTL-policy has no a priori information about the ongoing event and we have kept the epoch duration to one hour, hence the number of instances is only updated 8 times during the whole event. If there is some a priori knowledge about the event like its duration or/and a traffic forecast, one could exploit it, *e.g.*, by making the TTL-policy adapt faster during this event or setting the initial number of instances to the optimal static value predicted on the basis of the traffic forecast. This would further reduce the cost of our TTL solution.

VII. Conclusion

Dynamic sizing of cloud caches allows cloud users to adapt the cache size to the traffic pattern and minimize their total cost, which is given by the cost of the storage and the cost of the misses. We studied a TTL-based solution to dynamically track the required cache size. We provided a theoretical lower bound for the cost achievable by TTL solutions: in fact we characterize the optimal TTL policy (TTL-OPT) when the sequence of future requests is known. Moreover, we discussed a practical low-complexity implementation of a TTL solution, and evaluated it using real-world traces. Our experiments shows that our solution is able to track the optimal cache configuration and achieve significantly cost savings specially in highly dynamic settings

References

[1] "AWS ElastiCache," https://aws.amazon.com/elasticache/.

[2] "Microsoft Azure Redis Cache," https://azure.microsoft.com/en-us/services/cache/.

[3] "Memcached," https://memcached.org/.

[4] "Redis," https://redis.io/.

[5] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[6] "How Loading Time Affects Your Bottom Line," https://blog.kissmetrics.com/loading-time/.

[7] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching." in *HotStorage*, 2015.

[8] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *EuroSys*, 2015, p. 5.

[9] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *USENIX ATC*, 2017.

[10] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Computer Networks*, vol. 65, pp. 212 – 231, 2014.

[11] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network." in *NSDI*, 2017, pp. 483–498.

[12] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *ACM SoCC*, 2014.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS/PERFORMANCE*, 2012.

[14] "Facebook mcrouter," https://github.com/facebook/mcrouter.

[15] "Google Cloud Memorystore," https://cloud.google.com/memorystore/.

[16] "ElastiCache Pricing," https://aws.amazon.com/elasticache/pricing/.

[17] G. Neglia, D. Carra, M. Feng, V. Janardhan, P. Michiardi, and D. Tsigkari, "Access-time-aware cache algorithms," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 4, pp. 21:1–21:29, Nov. 2017.

[18] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, 2003, pp. 115–130.

[19] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *ACM ICAC*, 2010, pp. 1–10.

[20] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *ACM SoCC*, 2011, p. 5.

[21] D. Carra and P. Michiardi, "Memory partitioning in memcached: An experimental performance analysis," in *IEEE ICC*, 2014, pp. 1154–1159.

[22] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Lama: Optimized locality-aware memory allocation for key-value cache," in *USENIX ATC*, 2015, pp. 57–69.

[23] Z. Xu, C. Stewart, N. Deng, and X. Wang, "Blending on-demand and spot instances to lower costs for in-memory storage," in *IEEE INFOCOM*, 2016, pp. 1–9.

[24] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *EuroSys*, 2017, pp. 620–634.

[25] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman, "Adaptive ttl-based caching for content delivery," in *ACM SIGMETRICS*, 2017, pp. 45–46.

[26] Y. Zhong et al., "Program locality analysis using reuse distance," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, pp. 1–39, 2009.

[27] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data, "Characterizing storage workloads with counter stacks." in *OSDI*, 2014.

[28] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *USENIX ATC*, 2017, pp. 487–498.

[29] H. Che, Y. Tung, and Z. Wang, "Hierarchical Web caching systems: modeling, design and experimental results," *IEEE JSAC*, vol. 20, no. 7, pp. 1305–1314, Sep 2002.

[30] M. Dehghan, L. Massoulié, D. Towsley, D. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," in *IEEE INFOCOM*, April 2016, pp. 1–9.

[31] E. G. Coffman and P. J. Denning, *Operating systems theory*. Prentice-Hall Englewood Cliffs, NJ, 1973, vol. 973.

[32] H. Kushner and G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*. Springer New York, 2003.

[33] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[34] S. Hosseini-Khayat, "On optimal replacement of nonuniform cache objects," *IEEE Tran. on Computers*, vol. 49, no. 8, pp. 769–778, 2000.

[35] J. Erman and K. K. Ramakrishnan, "Understanding the super-sized traffic of the super bowl," in *ACM IMC*, 2013, pp. 353–360.