Distributed Load Balancing in Key-Value Networked Caches

Sikder Huq, Zubair Shafiq, Sukumar Ghosh
The University of Iowa

Amir Khakpour, Harkeerat Bedi
Verizon Digital Media Services

***Abstract*—Modern web services rely on a network of distributed cache servers to efficiently deliver content to users. Load imbalance among cache servers can substantially degrade content delivery performance. Due to the skewed and dynamic nature of real-world workloads, cache servers that serve viral content experience higher load as compared to other cache servers. We propose a novel distributed load balancing protocol called Meezan to address the load imbalance among cache servers. Meezan replicates popular objects to mitigate skewness and adjusts hash space boundaries in response to load dynamics in a novel way. Our theoretical analysis shows that Meezan achieves near perfect load balancing for a wide range of operating parameters. Our trace driven simulations shows that Meezan reduces load imbalance by up to 52% as compared to prior solutions.**

I. Introduction

Background & Motivation. Distributed key-value networked caches (e.g., memcached [2], redis [3]) are widely used by modern web services. Networked cache systems decrease latencies by caching popular content and also mitigate performance bottlenecks at backend storage. Memcached is used by many large-scale web services such as Facebook, YouTube, Twitter, and Wikipedia. Facebook, for example, deploys more than ten thousand memcached servers which have enabled up to 10X performance improvement [23], [19].

A key challenge faced by large-scale key-value networked caches is *skewed* and *dynamic* workload which can result in significant load imbalance. For instance, viral content in online social media represents a disproportionate fraction of total cache workload [5], [27], [26]. Moreover, peak and trough workload of real-world caches often differ by several orders of magnitude [5], [27], [26]. The load imbalance caused by skewed and dynamic workload can nullify the performance benefit of using cache systems and may even result in performance degradation [14]. Prior research has demonstrated that load imbalance can result in more than 60% degradation in throughput and $3\times$ degradation in latency [8], [12].

Technical Challenges. Given that load imbalance caused by skewed and dynamic workload may severely degrade overall system performance, it is crucial to distribute load evenly among cache servers. There are three key technical challenges. First, the time-varying nature of workload requires addition or removal of servers to ensure high resource utilization. Second, such addition or removal of servers involves data migration among cache servers that also needs to be minimized. Finally, centralized solutions to sufficiently address the load imbalance problem are not scalable because the centralized point may cause performance bottlenecks.

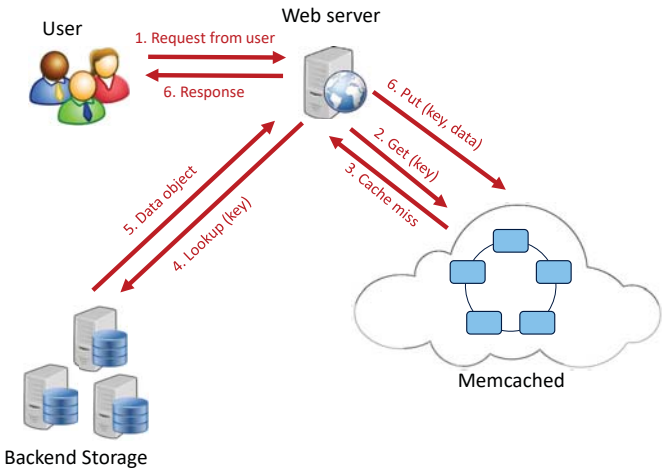


Fig. 1. Front-end web server requests for a data objects to the appropriate cache server. The cache server replies with object if the object is present in cache. Otherwise, the object is copied from the origin storage.

Limitations of Prior Art. Prior work commonly employs *consistent hashing* [16] for look-up operations, data partitioning, and load balancing. However, consistent hashing does not guarantee load balancing because it simply maps cache servers to random positions in the hash space and it is not sensitive to server load. Poor load balancing results in more than necessary server additions or removals, which in turn results in more data migration. Two techniques are typically used to mitigate this problem: *data replication* and *hash space adjustment*. Data replication aims to replicate popular objects at multiple cache servers [12]. Hash space adjustment aims to adaptively adjust the hash space boundaries between neighboring cache servers [15]. We show that existing data replication and hash space adjustment techniques do not achieve optimal load balancing. We also show that while data replication and hash space adjustment individually improve load balancing, they still leave much scope for further improvement when not carefully applied together. To the best of our knowledge, prior work lacks distributed data replication and hash space adjustment approaches to address the load imbalance problem.

Problem Statement. Figure 1 shows a typical network of cache servers. Front-end web servers forward requests to cache

servers by mapping the requests to a circular hash space using a random hash function. When a request maps to a specific position in the circular hash space, the nearest clockwise server becomes responsible for responding to that request. If the requested object is available at the cache server, it sends the object to the front-end web server. Otherwise, the cache server responds with a cache miss, following which, the web server fetches the object from backend storage and fills the cache. Our goal is to design a protocol that minimizes load imbalance among cache servers without requiring centralized coordination. To this end, cache servers can replicate objects in other cache servers, redistribute the hash space amongst themselves, and add or remove cache servers.

Proposed Approach. In this paper, we propose Meezan[1], a novel distributed protocol for optimizing load balancing in large-scale networked key-value caches. Meezan optimizes load balancing by addressing the skewed and dynamic workload using fully distributed techniques. Meezan addresses the skewness of workload by ensuring that the load associated with a distinct key is sufficiently small. To this end, Meezan replicates popular objects at multiple cache servers. More specifically, Meezan identifies popular objects and then adds a salt[2] to their object identifiers before computing the hash function. This ensures that the load associated with popular objects is divided among multiple cache servers.

Meezan addresses the dynamic nature of workload by enabling cache servers to efficiently adjust their hash space boundaries. To this end, Meezan forms random localities to divide cache servers into smaller groups. Each locality contains adjacent servers; which means each locality is a random chain cut from the ring. After localities are formed, each server shares its load with other servers in its locality. Once a server knows the load of all other servers in its locality, it conducts one of the following actions: (1) removes itself from the system and appropriately divides the hash space between the continuing surviving neighbors; (2) adds one or more new server(s) as neighbor(s) and migrate appropriate hash space to the continuing surviving or new neighbors; (3) migrates appropriate hash space to the continuing surviving or new neighbors. The procedure of locality formation, server addition/removal, and hash space migration is repeated after a fixed time interval. Note that locality formation and locality computation require only current local knowledge. The distributed cooperation of randomly chosen and independent localities to implement system-wide load balance is the cornerstone of our proposed approach.

Results. Our theoretical analysis shows that Meezan's replication policy mitigates the skewness in key distribution by limiting the expected number of requests mapped to a distinct key. We also derive an upper bound for load imbalance and show that Meezan achieves near perfect load balancing for a reasonable choice of parameters. We further conduct trace-driven simulations using content access logs from five geographically distributed data centers of a commercial Content Delivery Network (CDN). We compare Meezan with three prior approaches (CH [16], SPORE [12], APAC [15]) in terms of several performance metrics. In addition to load balancing, the performance metrics include number of servers, number of server additions/removals, data migration, and replication overhead. Our results demonstrate that Meezan significantly outperforms prior approaches. As compared to the prior approaches, Meezan reduces load imbalance by up to 52%, average number of servers up to 28%, number of server additions/removals up to 91%, data migration up to 81%, and replication overhead up to 40%.

[1]Meezan means *balance* in Arabic language.

[2]A small suffix to create a distinct identity.

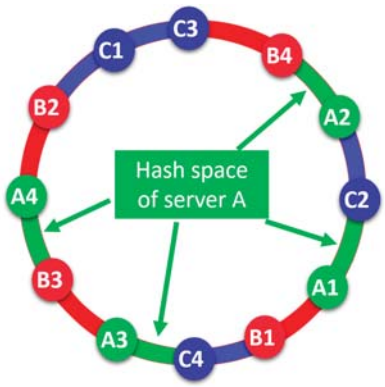


Fig. 2. A ring based hash space for 3 servers, each with 4 virtual nodes.

Paper Organization: The rest of the paper is organized as follows. Section II presents the problem statement and background. Section III introduces our proposed approach. Section IV evaluates and compares our proposed approach with prior work. We provide an overview of related work in Section V and conclude in Section VI.

II. Preliminaries

A. Design Goal

We consider a key-value networked cache architecture, illustrated in Figure 1, where web servers forward user requests to cache servers by using a hash function. If the cache server does not already have the requested object, it notifies the web server about cache miss. The web server then fetches the object from a backend storage to respond to user request, and the object may also be stored in the cache server depending on the cache replacement policy. Our goal is to design a distributed protocol for optimally balancing load among cache servers. Since cache servers can be added or removed (i.e., elastic resource allocation) in response to varying load, in addition to load balancing, we also aim to maximize utilization of cache servers. In sum, we desire all servers to be as evenly loaded as possible, and at the same time as much loaded as possible without exceeding the capacity of any server.

B. Background & Motivation

1) Consistent Hashing: Consistent hashing [16] is widely used for load balancing in many distributed systems. As illustrated in Figure 2, consistent hashing uses a circular hash

space, where the position of a server in the hash space is determined by computing the hash of the server's unique identifier. Note that a server can have multiple presence (i.e., virtual nodes) in the hash space. The positions of virtual nodes in the hash space are determined by adding incremental numbers to the server's unique identifier before hash computation. When a request arrives, the hash value of the requested object's identifier is computed and the server with closest clockwise virtual node becomes responsible for serving the request.

Consistent hashing can lead to severe load imbalance for skewed and dynamic workloads because it solely relies on randomization for resource allocation and is non-adaptive to change in access patterns. To illustrate this, we evaluate load balancing provided by consistent hashing using real-world content access logs from a commercial CDN. Our analysis suggests that some objects become very popular and cause load imbalance among cache servers. Figure 4(a) shows the skewed popularity distribution of objects in the CDN workload. The straight line on the log-log popularity distribution indicates that some objects are extremely popular whereas others are much less popular. Figure 4(b) shows the number of total requests and the number of distinct requests served by each cache server. We note that some servers are almost 5× more loaded than the average load (max/avg = 4.9).

2) Data Replication: The use of data replication in consistent hashing can improve load balancing by replicating popular objects across multiple cache servers. To replicate an object, a *salt* is concatenated with the object's identifier before passing to the hash function. For example, if an object's frequency exceeds a predefined *replication threshold* r, we replicate the object by adding a random salt picked from the range $[1, \lceil f/r \rceil]$, where f is the request frequency of the object. Figure 3 shows an example where an object is replicated in two additional servers (C and E) by using salts.

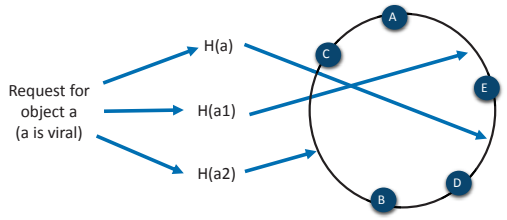


Fig. 3. An example of data replication. Incremental numbers are added to the object's identifier to distribute the load associated with popular objects across multiple servers.

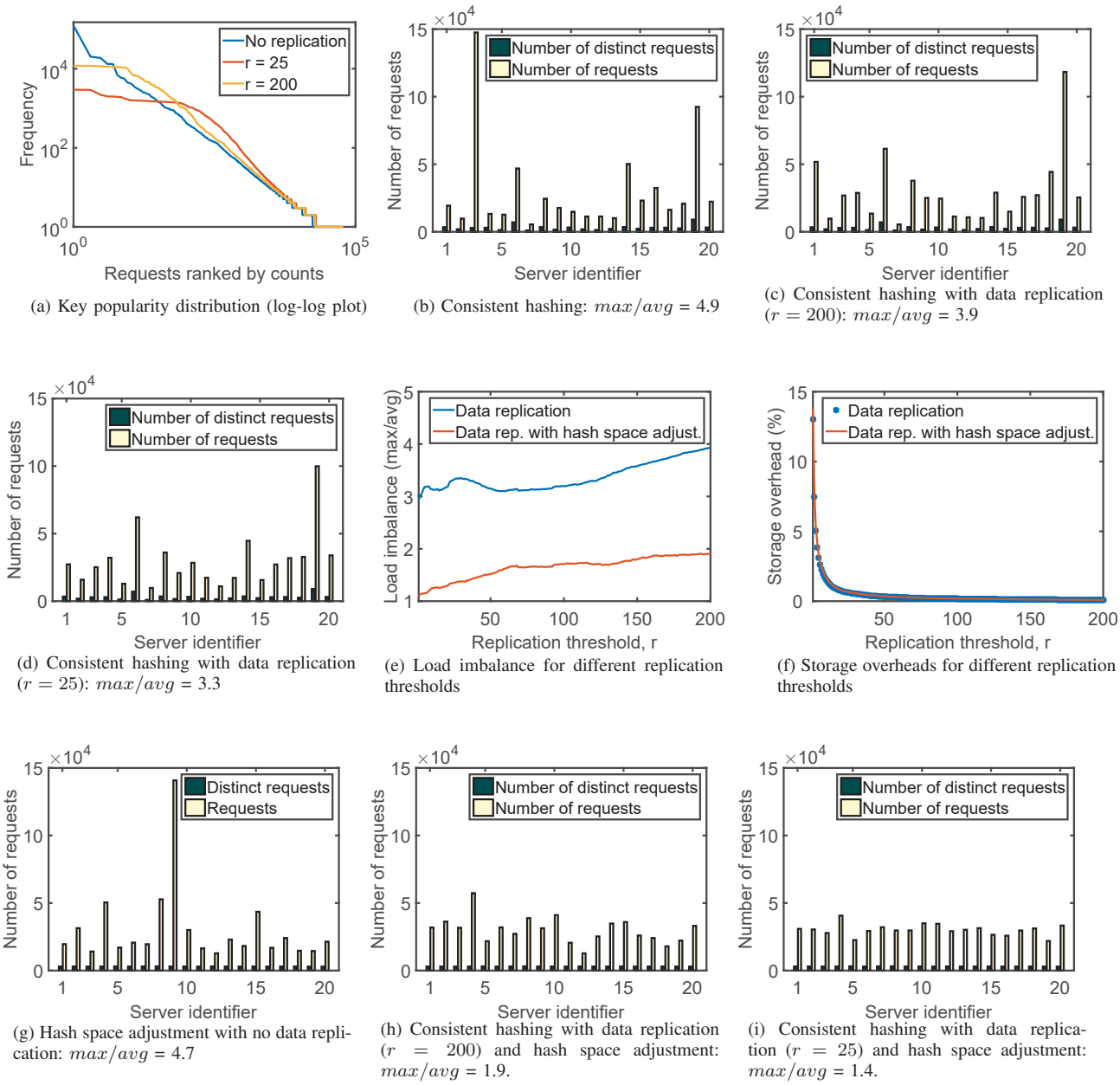
We evaluate load balancing provided by consistent hashing and data replication [12] using CDN access logs. Figures 4(c) and (d) show the number of total and distinct requests served by each server for r =200 and r =25, respectively. Comparing Figures 4(b) and (c), we observe that data replication improves load balancing in terms of max/avg from 4.9 to 3.9. Comparing Figures 4(c) and (d), we observe that reducing the replication threshold from 200 to 25 improves load balancing in terms of max/avg from 3.9 to 3.3.

To analyze the impact of data replication on load balancing, we compare key popularity distributions in Figure 4(a). We observe that data replication reduces the skewness for keys with request frequency greater than the replication threshold. Data replication improves load balancing because the skewness of key popularity distribution is reduced. However, load imbalance persists because the skewness is not completely eliminated. In Figure 4(e), we show that more aggressive data replication does not result in considerably reducing load imbalance. Further, in Figure 4(f), we show that reduction of replication threshold results in increased storage overhead. Thus, as also reported in prior work [14], we conclude that data replication alone is not sufficient for load balancing.

3) Hash Space Adjustment: Transferring a portion of hash space from a highly loaded server to a server with relatively lower load can reduce load imbalance. Figure 5 shows an example of hash space adjustment, where parts of hash space from servers E and C are transferred to server A. We implement and evaluate load balancing provided by consistent hashing and hash space adjustment [15]. We adjust the hash space by shifting the hash space boundaries between top five most load disparity neighboring server pairs. Figure 4(g) shows that hash space adjustment performs worse (max/avg = 4.7) than data replication in terms of load balancing. The reason is that some objects are extremely popular and any server caching them becomes much more loaded than other servers. In fact, the load associated with a viral object may even exceed a server's capacity. Hence, we conclude that hash space adjustment alone is not sufficient for load balancing.

We observe a substantial improvement in load balancing when we combine data replication and hash space adjustment. Figures 4(g), (h), and (i) show that hash space adjustment with data replication significantly improves load balancing over the approaches that use only data replication or only hash space adjustment. We observe that hash space adjustment with data replication for $r = 25$ improves load balancing in terms of max/avg to 1.4. Figure 4(e) shows that hash space adjustment with data replication leads to substantially better load balancing over data replication alone for a wide range of replication thresholds. As shown in Figure 4(f), hash space adjustment with data replication incurs at most 0.2% more storage overhead as compared to data replication without hash space adjustment for any $r \geq 4$. Comparing Figures 4(e) and (f), we conclude that hash space adjustment reduces storage overhead by allowing us to choose a relatively higher replication threshold, yet achieving a significant improvement in load balancing.

4) Dynamic server addition/removal: Most web applications deploy elastic cache systems [30] to address large variations in workload. While under-provisioning causes performance bottlenecks, over-provisioning results in waste of resources. A practical approach to address this issue is to dynamically add or remove servers on an on-demand basis. When the overall system load approaches the overall capacity,



(a) Key popularity distribution (log-log plot)

(b) Consistent hashing: max/avg = 4.9

(c) Consistent hashing with data replication ($r = 200$): max/avg = 3.9

(d) Consistent hashing with data replication ($r = 25$): max/avg = 3.3

(e) Load imbalance for different replication thresholds

(f) Storage overheads for different replication thresholds

(g) Hash space adjustment with no data replication: max/avg = 4.7

(h) Consistent hashing with data replication ($r = 200$) and hash space adjustment: max/avg = 1.9.

(i) Consistent hashing with data replication ($r = 25$) and hash space adjustment: max/avg = 1.4.

Fig. 4. Performance of different consistent hashing based approaches for an one hour long real-world access log.

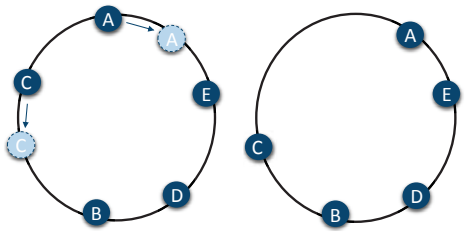
we need to add more servers to ensure that the load on any server does not exceed its capacity. Similarly, to optimize resource utilization, we need to remove servers when the overall system load decreases sufficiently. Figure 6 shows an example where two servers are added and one server is removed from the hash space.

III. Proposed Approach

A. Architectural Overview

Cache servers are placed in a ring overlay in the order of their hash space placement. The position of a server in the hash space is determined by computing the hash of the server's unique identifier. Each server has a clockwise neighbor and an anticlockwise neighbor in the ring overlay which hold clockwise and anticlockwise adjacent portions in hash space, respectively. Web servers use consistent hashing to send a request to the appropriate cache server. Specifically, a request's identifier is hashed to compute its key and the request is sent to the cache server responsible for that portion of the hash space. The cache server that receives a request is assigned as the *home server*[3] for the request. Upon receiving a request, the home server performs one of the following actions. (1) It responds with a cache miss if the requested object is not present in the cache; (2) it responds with the requested object if the requested

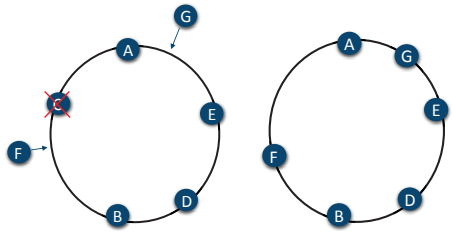
[3]The term *home server* is borrowed from [12].



(a) A ring based hash space with 5 servers before hash space adjustment.

(b) Hash space adjustment by changing hash space boundaries allocated to the servers

Fig. 5. An example of hash space adjustment.



(a) A ring based hash space for 5 servers before addition/removal

(b) Addition of 2 servers (E, F) and removal of server C in the hash space shown in (a)

Fig. 6. An example of server addition/removal.

object is present in the cache; or (3) it forwards the request to another cache server that contains a replica of the requested object. When a request is forwarded to another server by the home server, the recipient server can either respond with the requested object or a cache miss. In the latter case, the home server sends the requested object to the client (from its cache or backend storage) and also replicates the requested object at the server that reported the cache miss.

B. Solution Overview

Our solution for load balancing relies on two techniques: (1) identification and replication of popular objects, and (2) adjustment of hash space boundaries. Home servers are responsible for identifying and replicating popular objects which are mapped to them through consistent hashing. Hash space adjustment takes place at the beginning of each time interval. In hash space adjustment, servers form random *localities* and broadcast their load counts to other servers in their localities. Based on this information, servers adjust their hash space boundaries within the locality to minimize load imbalance. One or multiple servers may be added or removed from the locality depending upon the overall load of a locality. Servers transfer keys to appropriate servers so that web servers can continue using consistent hashing.

C. Popular Object Replication

We propose a replication policy to limit the maximum number of requests within a time interval associated with a key. To identify popular objects, home servers maintain an exponentially weighted moving average (EWMA) of request counts for each object that is mapped to them by consistent hashing. Since the request for an object always goes to the same home server, only one cache server is responsible for maintaining EWMA for the object.

Upon a request for an object a, a popularity weight $w = \frac{max(C,M)}{r}$ is computed, where C is the request count of an object so far in the current time interval, M is the EWMA of request count for the object, and r is the replication threshold. If $w < 1$, the original request is hashed without adding any salt. Otherwise, if $w \geq 1$, the salt is chosen as follows:

$$salt = \begin{cases} \text{random integer from } [1, \lceil w \rceil] & \text{if } C \leq M \\ \lceil w \rceil + 1 & \text{if } C > M \end{cases} \quad (1)$$

We use random salt to ensure that roughly the same number of requests are mapped to each distinct key. However, the expected number of requests mapped to a distinct key is greater than r when $C > M$. Thus, to limit the severity of skewness in key distribution, we use a deterministic salt when $C > M$ to ensure that the expected number of requests mapped to a distinct key in a time interval is at most r.

Theorem 1. ***(Replication Theorem)*** *Considering the hash function to be perfect (i.e., no collisions), for a given time interval, the expected number of requests mapped to a distinct key is at most r.*

Proof. From Equation 1, salt = $\lceil \frac{C}{r} \rceil$ when $C > M$. Thus, a unique salt is used for each r requests of object a after C exceeds M. Now, consider the case when $C \leq M$. According to Equation 1, salt is an integer that is chosen randomly from $[1, \lceil \frac{M}{r} \rceil]$. Let j be an integer s.t. $1 \leq j \leq \lceil \frac{M}{r} \rceil$, and X_i be an indicator random variable s.t. for the given time interval:

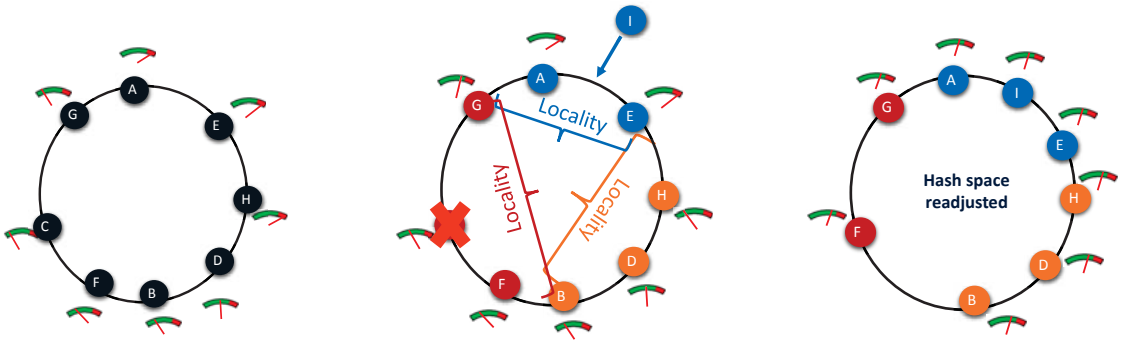
$$X_i = \begin{cases} 0 & \text{if } salt \neq j \text{ for } i^{th} \text{ request of } a \\ 1 & \text{if } salt = j \text{ for } i^{th} \text{ request of } a \end{cases}$$

Let c denotes the number of times a random salt was used for object a in the previous time interval. Obviously $c \leq M$. Let object a map to key k when $salt = j$, and the number of requests that map to key k is X. Clearly, $X = \sum_{i=1}^{c} X_i$. Using *linearity of expectation*, the expectation of X:
$\mathbb{E}[X] = \sum_{i=1}^{f} \mathbb{E}[X_i] = \sum_{i=1}^{c} Pr[X_i = 1] = c \lceil \frac{r}{M} \rceil \leq r.$

□

D. Hash Space Adjustment

We propose a novel distributed implementation of hash space adjustment policy where servers rely on local knowledge to adjust their position in the hash space. Each cache server maintains request counts for all distinct keys requested in a time interval. Each server also maintains a *load counter* to keep track of the total number of requests received in the



(a) Cache servers; their positions in the ring based hash space; and their loads before adjustment

(b) Random localities are formed.

(c) servers are repositioned in the hash space within their locality; one server is added and one server is removed.

Fig. 7. Illustration of hash space adjustment

current time interval. As we discus next, hash space adjustment is performed in three steps: (1) form random localities and share load counters; (2) add or remove servers, if necessary; and (3) adjust hash space boundaries. Figure 7 illustrates these three steps.

1) Formation of Random Localities: To divide the hash space into random localities, each server is declared a locality separator with probability $1/p$, where p is the *locality threshold*, and $p > 1$. Servers become part of the locality of their clockwise nearest locality separator. We refer the number of servers in a locality as the *size* of the locality. Note that the expected size of a locality is p. After formation of random localities, all servers broadcast their load counters (one count per server) to other servers within their localities.

2) Addition or Removal of Servers: Each server computes the total load of the locality by adding the load counters it receives from other servers in the locality. The server then divides the total load by the *load threshold* (also called server capacity) to compute the adjusted number of servers for the locality. Let n_0 and n be the number of servers in a locality before and after adjustment, respectively. If $n_0 = n$, there is no need for addition or removal of servers. Otherwise, we add (or remove) $|n - n_0|$ servers if $n > n_0$ (or $n < n_0$), respectively. We remove the least loaded servers and add servers neighboring to the most loaded servers.

3) Server Positioning and Key Transfer: After addition or removal of servers in each locality, we determine the adjusted server positions in the hash space as follows. Each server computes the *average load* ($\mathcal{A}$) by dividing the total load by the adjusted number of servers. Consider the locality of server s as a linked list consisting of the servers of the locality, where servers are placed in the linked list from left to right in their clockwise order in hash space after server addition/removal. Let $Load_{left}$ be the sum of load counts of all the servers positioned left to s in the linked list. Each server maintains a list of keys in its cache sorted [4] in clockwise order. Let $k_1, k_2, \ldots k_m$ be the sorted list of keys maintained by server s. Let $count(k)$ be request count of key k in the previous time interval. Then we define the rank of a key k_i as $rank(k_i) = Load_{left} + \sum_{j=1}^{i} count(k_j)$. Now server s transfers any key k_i to the y-th server (from left in the linked list) if $(y-1)\mathcal{A} < rank(k_i) \leq y\mathcal{A}$. To do so, servers need to check the first $m^{'}$ keys such that $Load_{left} + \sum_{j=1}^{m^{'}} count(k_j) \leq (y-1)\mathcal{A}$ and the last $m^{''}$ keys such that $Load_{left} + \mathcal{A} - \sum_{j=m^{''}}^{m} count(k_j) > y\mathcal{A}$. Note that the right most server in the linked list does not change its position in the hash space. All other servers adjust their positions in the hash space as a result of key transfer.

Definitions. We define *adjusted load* of a server as the number of requests that map to the adjusted hash space of the server during the previous time interval. The *adjusted* max/avg is defined as the ratio of the maximum adjusted load to the average adjusted load of servers. Finally, we define *average load* as the average load of servers within a locality after server addition/removal.

Theorem 2. ***(Local Load Balance Theorem)*** *Let H be the hash space covered by the servers of a locality. If the average load for the locality is $\mathcal{A}$ and the maximum number of requests mapped in the previous time interval to a distinct key $k \in H$ is R, the adjusted max/avg for the locality is at most $1 + \frac{R-1}{\mathcal{A}}$.*

Proof. Let $k_1, k_2, \ldots, k_m$ be the keys, sorted in the clockwise order based on their positions in the hash space H, requested to all servers of a locality in the previous time interval. We rank each key k_i as:

$$rank(k_i) = \sum_{j=1}^{i} count(k_j) \tag{2}$$

Figure 8 illustrates the hash space of a locality where m keys are distributed among n servers. After adjustment, if $H(y)$ is the hash space of the y-th server of the locality, according to Meezan, $\mathcal{A} \cdot (y-1) < rank(k) \leq \mathcal{A}y$ for all keys $k \in H(y)$. Now, load balancing is perfect if the adjusted load for $\forall y$ is exactly $\mathcal{A}$. However, this may not be possible if there exists a key k_i, $1 \leq i \leq m$, such

[4]Meezan does not sort keys, it simply maintains a sorted list.

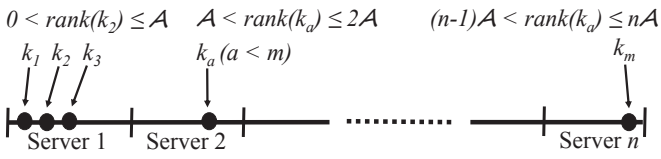


Fig. 8. Adjusted hash space in a locality

that $rank(k_{i-1}) < \mathcal{A} \cdot (y-1) < rank(k_i)$. In such a scenario, assuming $k_{i-1} \notin H(y)$ and $k_i \in H(x)$, we get $rank(k_i) - \mathcal{A} \cdot (y-1) \leq count(k_i)$. Since $count(k_i) \leq R$, adjusted load for the y-th server is at most $\mathcal{A} + R - 1$. This follows $max/avg \leq \frac{\mathcal{A}+R-1}{\mathcal{A}} = 1 + \frac{R-1}{\mathcal{A}}$. □

Now we analyze the load balance performance of Meezan for the entire system.

Lemma 1. *For any time interval, the average load of the entire system is lower bounded by $\frac{l(n-1)+1}{n}$, where n is the size of the smallest locality after server addition/removal.*

Proof. According to the design of Meezan, the average load for the smallest locality is at least $\frac{l(n-1)+1}{n}$. Now, the lemma follows because the average load of the entire system must be lower bounded by that of the smallest locality.

□

Theorem 3. ***(Global Load Balance Theorem)*** *The adjusted max/avg for the entire system is upper bounded by $\frac{n}{n-1}$, where n is the size of the smallest locality after server addition/removal.*

Proof. Using the argument we used to prove Theorem 2, the maximum adjusted load to any server is $l + R - 1$, where R is the maximum number of requests mapped to a distinct key. Using Lemma 1, we get:

$$\frac{max}{avg} \leq \frac{l+R-1}{\frac{l(n-1)+1}{n}} < \frac{n(l+R-1)}{l(n-1)} = \frac{n}{n-1}\left(1 + \frac{R-1}{l}\right)$$

$$\approx \frac{n}{n-1} \text{ [} R << l \text{, when } r \text{ small enough]}$$

Thus, $max/avg \to 1$ when n is sufficiently large. □

E. Implementation Details

Key transfer. Servers transfer keys in the sorted order so that the recipient servers do not need to sort them again. Since all servers use the same deterministic protocol to compute adjusted load and positions of new servers, servers can transfer keys in parallel without ambiguity. While transferring keys, servers also transfer the corresponding objects and their EWMA values (if available). Key transfer should include objects because the recipient servers would otherwise need to fetch objects from the backend storage. Since all servers in a locality are in geographical proximity (perhaps the same rack in a data center), object transfer to other servers in a locality is more efficient than fetching from backend storage. Further note that multiple keys, which are to be transferred, may map to the same object due to replication. To avoid redundant data transfer, servers transfer such objects only once.

Distributed implementation. Our proposed load balancing approach is distributed and does not require any central coordination or global knowledge. Servers form localities only by talking to their neighbors in the ring overlay. Locality separators forward their load counters to their anti-clockwise neighbors, while other servers forward their load counters to both the clockwise and anti-clockwise neighbors. Any server that is not a locality separator also forward load counters received from the anti-clockwise neighbor to the clockwise neighbor. It is noteworthy that we do not require any inter-locality communication since key transfers take place only within localities.

Bookkeeping overhead. We require servers to maintain a counter for all distinct keys that are requested in the current time interval. Servers delete these counters after hash space adjustment. Although the number of distinct keys in a time interval is manageable, we can further reduce the bookkeeping overhead by using sampling. Since there are typically only a few popular objects, a reasonable sampling rate will ensure that we maintain counters for most popular objects. We can also use sampling to reduce the bookkeeping overhead of EWMA counters that are maintained by home servers.

Heterogeneous servers. Note that servers may have different computational and storage abilities. We can generalize our approach by assigning different load thresholds to severs and use weighted averages to compute their adjusted load. Other modifications in our approach for adaptation to heterogenous servers are trivial.

Fixed number of servers. Our approach can also be used when the number of servers is fixed. To this end, we can bypass the server addition/removal step during hash space adjustment. While we can still guarantee load balancing in this case, we cannot provide any guarantees for resource utilization.

IV. Experimental Results

A. Data

For empirical evaluation, we use content access logs from five different geographically distributed data centers (Los Angeles, Singapore, Sydney, Amsterdam, and Atlanta) of a commercial CDN. Content access logs contain detailed information for each request, including timestamp, URL containing object's name, size of the object, and client information (e.g., hashed IP address). Overall, content access logs contain more than 212 million requests for 5.6 million distinct objects over the duration of one week. We use these content access logs to conduct trace-driven simulations for different load balancing approaches.

B. Performance Metrics

We use the following performance metrics.

1. $\boldsymbol{max/avg}$**:** The max/avg metric quantifies the load imbalance of a networked cache system. It is defined as the ratio of the number of requests received by the most loaded server (max) to the average load of all servers (avg). The minimum

value of max/avg is 1, which indicates perfect load balancing. The values of max/avg larger than 1 represent increasing load imbalance.

2. **Number of servers:** We measure the average number of servers to quantify the resource consumption of a networked cache system. If N_i is the number of servers used in the time interval i, the average number of servers for T time intervals is $\frac{\sum_{i=1}^{T} N_i}{T}$.

4. **Data Migration:** We quantify data migration as the amount of data transferred among servers due to server addition/removal, hash space adjustment and data replication. If D_i is the amount of data transferred in the time interval i, the average data migration for T time intervals is $\frac{\sum_{i=1}^{T} D_i}{T}$.

5. **Replication Overhead:** Replication overhead quantifies the number of replicas (copies) of objects used to serve requests. Let X be the number of distinct objects requested in a time interval, and Y be the number of total replicas used to serve the X requested objects, replication overhead is defined as $\frac{Y-X}{X}$.

C. Baselines for Comparison

We compare our approach with the following baseline approaches for a range of operating parameters. We briefly summarize the baselines as follows.

1. **Consistent Hashing (CH).** Our first baseline is a dynamic version of consistent hashing [16] that adds or removes servers on demand in every time interval. To dynamically add or remove servers, we use two thresholds referred to as *load threshold* and *low threshold*. A server is removed if its load falls below the low threshold and a server is added if the load of an existing server exceeds the load threshold. The servers are mapped to the hash space by hashing server identifiers by using a random hash function.

2. **Self-adaptive POpularity based REplication (SPORE):** SPORE aims to identify and replicate popular objects for minimizing load imbalance [12]. We use a dynamic version of SPORE as our baseline. To dynamically add or remove servers, we again use the *load threshold* and *low threshold*.

3. **Adaptive Performance-Aware Caching (APAC):** APAC performs hash space adjustment and dynamic server addition/removal to reduce load imbalance [15]. This approach requires all servers to have multiple replicas (positions) in the hash space, where each replica is called a *virtual node*. As we do for other baselines, we remove a server when its load falls below the *low threshold*. However, unlike other baselines, APAC always adds new servers to support the overloaded servers. Hash space adjustment is performed in each time interval by finding the most load disparity pair of neighboring servers and adjusting the hash space boundary between them. We used the data replication policy from SPORE and the hash space adjustment technique from APAC in this baseline.

D. Experimental Setup

We conduct trace-driven simulations to compare and contrast our approach and the aforementioned baselines. We use Least Recently Used (LRU) as the underlying cache eviction policy in our experiments. Our simulations for CH and SPORE start with 25 servers each with 10 replicas (or *virtual nodes*) in the hash space. We initialize our experiments for APAC with 25 servers each with 100 (the system default in [15]) virtual nodes. APAC requires two more parameters, α and β, to control adjustments in the hash space boundaries. We set $\alpha = 0$ to ensure that the protocol intends to only optimize load balancing and does not consider hit rate as a metric of cost. After pilot analysis, we set $\beta = 0.95$, which means the load balancer works aggressively by moving up to 95% of the hash space during hash space adjustment. Same as all the baselines, we initialize simulation for Meezan with 25 servers. We do not use virtual nodes for our approach as we require every server to have a single position in the hash space. We use 1 minute time intervals for all our simulations. To evaluate Meezan and baselines, we set the replication threshold r = 25 and we set the locality threshold p = 15 for Meezan.

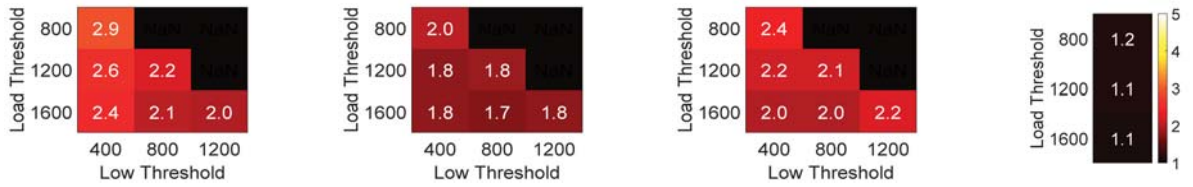
E. Performance Comparison

Figures 9, 10, 11, and 12 show the performance of baselines for different combinations of load and low thresholds, and the performance of Meezan for different load thresholds.

Load balancing. As shown in Figure 9, Meezan outperforms all baselines in terms of load balancing (max/avg). Meezan outperforms CH and SPORE because new servers are placed at the random positions in the hash space by CH and SPORE. Thus, overloaded servers are not guaranteed to be supported by the new servers by these approaches. As a consequence, overloaded servers may continue with their high load for an extended period of time. Meezan outperforms APAC due to the following reasons. First, Meezan conducts more aggressive hash space adjustment as compared to APAC. APAC adjusts the hash space boundaries only between the neighboring virtual nodes at the most load disparity server pairs. In contrast, Meezan allows all servers to adjust their hash space boundaries simultaneously. Second, APAC transfers exactly half of the hash space from an overloaded server to a new server. This approach does not guarantee optimal hash space adjustment. In contrast, Meezan identifies optimal positions and adjusts hash space for all servers when adding/removing servers.

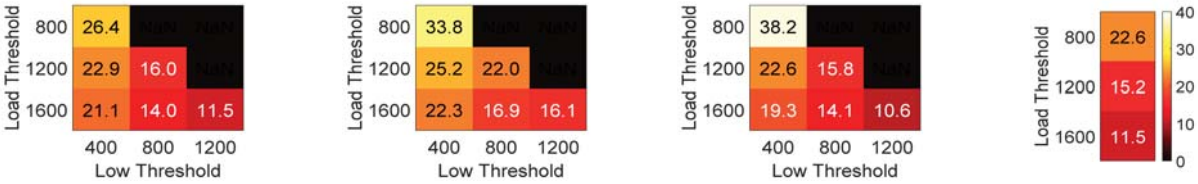
Average number of servers. Figure 10 shows that Meezan uses fewer servers on average as compared to baselines. The reason is that Meezan aggressively adjusts hash space boundaries and adds/removes servers on demand to ensure that the average load is as close as possible to the load threshold. Furthermore, we note that, for a wide range of threshold combinations, Meezan reduces average number of server additions/removals by 91% as compared to baselines.

Data migration. Figure 11 reports less average data migration for Meezan as compared to the baselines. The reason is that Meezan adds/removes fewer servers as compared to baselines. We note that data migrated due to server addition/removal is much more than that due to hash space adjustment. Thus,



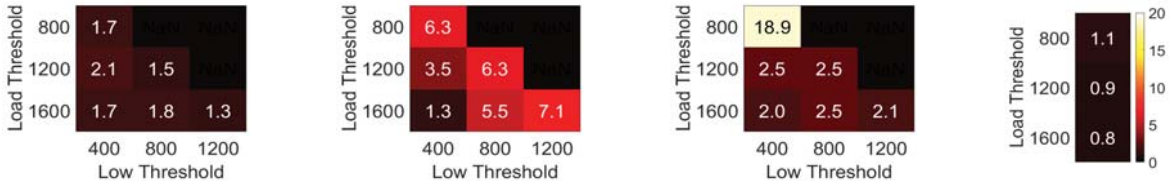
(a) CH (b) SPORE (c) APAC (d) Meezan

Fig. 9. Average max/avg



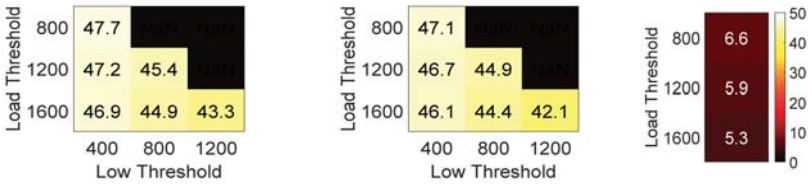
(a) CH (b) SPORE (c) APAC (d) Meezan

Fig. 10. Average number of servers



(a) CH (b) SPORE (c) APAC (d) Meezan

Fig. 11. Average data migration (in gigabytes)



(a) SPORE (b) APAC (c) Meezan

Fig. 12. Average replication overhead (%)

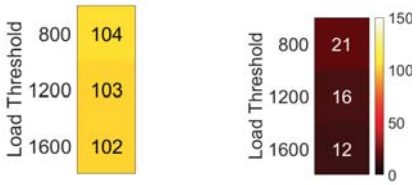
aggressive hash space adjustment conducted by Meezan has a little impact on data migration.

Replication overhead. Figure 12 shows that Meezan incurs 40% less replication overhead compared to SPORE and APAC. The reason is Meezan minimizes the average number of servers 10. Since a server can have at most one replica of an object, Meezan requires less replicas to serve popular objects compared to SPORE and APAC. Moreover, SPORE and APAC add/remove more servers to respond to load dynamics as compared to Meezan, resulting in creation of more replicas for popular objects.

F. Discussions

Bookkeeping Overhead. Figure 13 shows average number of counters maintained and transferred by a server in a time interval (1 minute time interval is used). We observe from Figure 13 that the average number of total counters maintained by a server is roughly equal to the load threshold, and the percentage of counters that need to be migrated due to load balancing decreases as we increase the load threshold.

Parameter Selection. To understand parameter selection for Meezan, we conduct a series of experiments for varying locality thresholds p, load thresholds $\{400, 1600, 4000, 8000\}$, and replication thresholds $\{5, 15, 25, 100\}$. Our experiments show that load balancing improves as we decrease replication threshold for a fixed load and locality threshold. For a fixed replication and load threshold, increasing the locality threshold has no impact on load balancing given that we chose a reasonable value (e.g., at least 5) for locality threshold. Replication and locality thresholds have a little impact in load balancing when we choose a sufficiently large load threshold. For large load thresholds, other parameters have almost no impact on load balancing and we achieve near perfect load balancing. We conclude that Meezan achieves near perfect load balancing when the load threshold is sufficiently larger than the replication threshold, which corroborates our analysis presented in Theorems 2 and 3.



(a) Counters maintained

(b) Counter migration

Fig. 13. Average number of counters (% of load threshold) per server per time interval for Meezan.

G. Summary

Meezan outperforms all baselines in terms of different performance metrics for a wide range of parameters. We find that Meezan reduces load imbalance by 52%, average number of servers by 12%, and average data migration by 43%, on average, as compared to CH. We find that Meezan reduces load imbalance by 38%, average number of servers by 28%, average data migration by 81%, and replication overhead by 40%, on average, as compared to SPORE. We find that Meezan reduces load imbalance by 0.47%, average number of servers by 18%, average data migration by 81%, and replication overhead by 39%, on average, as compared to APAC.

V. Related Work

Consistent Hashing. Distributed hash tables (DHT) have been widely used in peer-to-peer (P2P) systems (e.g. [29], [18], [25], [10], [11]) for efficient lookup and load balancing. These systems use *consistent hashing* [16], [17] for data partition. Prior work [14], [5], [15], [12] as well as this paper, show that consistent hashing is not effective in optimizing load balancing for skewed and dynamic workloads. Although our work uses a ring based hash space similar to consistent hashing, the focus of this work is to study additional data replication and hash space adjustment techniques to mitigate load imbalance in networked caches.

Data Republication. Data replication [23], [6], [22], [31] is widely used to mitigate the impact of workload skewness. Hong et al. [12] proposed an adaptive data replication technique, which is, to the our best knowledge, the most recent work on adaptive data replication designed for distributed key-value cache systems. In contrast to [12], our data replication policy is different because we consider the request frequency of current time interval along with EWMA count to compute salt. Huang et al. [14] investigated the reasons of load imbalance across cache servers using real world traces from Facebook's Tao cluster [7]. They concluded that existing techniques such as consistent hashing and replication approaches do not completely address the load skewness. Our work attempts to fill this gap by combining data replication with dynamic hash space adjustment.

Hash Space Adjustment. Hwang et al. [15] proposed an adaptive load balancer that allows adjustment in hash space boundaries. Their approach relies on initial positioning of virtual nodes and centralized coordination. In contrast, our hash space adjustment technique is distributed, more aggressive compared to [15], and when combined with our data replication policy outperforms both [12] and [15]. Li et al. [20] proposed a dynamic server provisioning based protocol for memory cache clusters to optimize load balancing and minimize data migration under dynamics. Our approach is different because: (1) we do not use virtual nodes; (2) we allow dynamic shift of hash space boundaries in addition to adding or removing servers; and (3) our approach is fully distributed whereas Proteus is not. Cheng et al. [8] proposed a protocol, called MBal, for multi-threaded servers. MBal uses a configurable resource container called *cachelet* that encapsulates multiple virtual nodes. For server level load balancing, MBal migrates cachelets from one server to another along with other load balancing techniques. Our work differs as we dynamically adjust the hash space boundaries and always migrate data from a server to its neighboring servers. Blink [28] and Couchbase [1] use centralized coordination for remapping keys or buckets to the servers. In contrast, our approach adjusts hash space boundaries in a distributed manner without requiring global coordination.

Miscellaneous. LAMA [13], CircularCache [21], and DynaCache [9] proposed adaptive cache controllers that focus on cache eviction policy and/or memory management. Although Meezan results in efficient resource management by reducing replication overhead and average number of servers, our focus is on reducing load imbalance rather than improving cache hit ratio. AdaptCache [4] proposed an adaptive load balancing technique for distributed caches, however their work is designed for a different architecture, and not applicable to networked caches such as Memcached. Finally, RackOut [24] proposes a rack-scale memory pooling technique to mitigate load imbalance. In contrast, our approach targets architectures where servers cannot directly access other servers' memory.

VI. Conclusion

Load imbalance in large-scale networked cache systems can substantially degrade overall system performance. Achieving optimal load balancing is challenging due to the dynamic and skewed nature of real-world workloads. To this end, we presented a distributed load balancing protocol (Meezan) for key-value networked caches. Meezan addresses workload skewness by ensuring that the load associated with popular objects is divided among multiple cache servers. Meezan addresses the dynamic nature of workload by enabling cache servers to efficiently adjust their hash space boundaries. Our theoretical analysis shows that Meezan achieves nearly perfect load balancing for a wide range of operating parameters. Our empirical results show that, as compared to baselines, Meezan reduces load imbalance by up to 52%, average number of servers up to 28%, number of server additions/removals up to 91%, data migration up to 81% and replication overhead up to 40%.

REFERENCES

[1] Couchbase: a NoSQL database, http://www.couchbase.com.

[2] Memcached: distributed memory objet caching system, http://memcached.org.

[3] Redis: in-memory data structure store, http://redis.io.

[4] O. Asad and B. Kemme. Adaptcache: Adaptive data partitioning and migration for distributed object caches. In *ACM Middleware*, 2016.

[5] 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.

[6] G. Barish and K. Obraczke. World wide web caching: Trends and techniques. *Comm. Mag.*, 38(5):178–184, May 2000.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[8] Y. Cheng, A. Gupta, and A. R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *European Conference on Computer Systems (EuroSys)*, 2015.

[9] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2015.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *ACM SOSP*, 2001.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, 2007.

[12] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Annual ACM Symposium on Cloud Computing (SOCC)*, 2013.

[13] 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 *2015 USENIX Annual Technical Conference*, 2015.

[14] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *ACM Workshop on Hot Topics in Networks*, 2014.

[15] J. Hwang and T. Wood. Adaptive Performance-Aware Distributed Memory Caching. In *USENIX International Conference on Autonomic Computing (ICAC)*, 2013.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing (STOC)*, 1997.

[17] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *WWW*, 1999.

[18] D. R. Karger and M. Ruhl. Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-peer Networks. In *International Conference on Peer-to-Peer Systems*, 2004.

[19] S. Kumar. Social networking at scale, http://www.ece.lsu.edu/hpca-18/files/HPCA2012_Facebook_Keynote.pdf. In *Keynote Talk, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[20] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. F. Abdelzaher. Proteus: Power Proportional Memory Cache Cluster in Data Centers. In *ICDCS*, pages 73–82. IEEE Computer Society, 2013.

[21] Y. Z. X. Y. S. Z. J. W. C. X. Liqiong Liu, Xiaoyang Qu. Circularcache: Scalable and adaptive cache management for massive storage systems. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2016.

[22] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. *Comput. Netw. ISDN Syst.*, 30(22-23):2169–2177, Nov. 1998.

[23] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *USENIX Conference on Networked Systems Design and Implementation*, 2013.

[24] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the 2016 ACM Symposium on Cloud Computing*, 2016.

[25] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, Oct. 2001.

[26] M. Z. Shafiq, A. R. Khakpour, and A. X. Liu. Characterizing Caching Workload of a Large Commercial Content Delivery Network. In *IEEE INFOCOM*, 2016.

[27] M. Z. Shafiq, A. X. Liu, and A. Khakpour. Revisiting Caching in Content Delivery Networks. In *ACM SIGMETRICS*, 2014.

[28] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing Server Clusters on Intermittent Power. *SIGARCH Comput. Archit. News*, 39(1):185–198, Mar. 2011.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, 2001.

[30] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 45–52, 2011.

[31] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.