Ultra-Fast Load Balancing of Distributed Key-Value Stores through Network-Assisted Lookups

Davide De Cesaris[1,2], Kostas Katrinis[1], Spyros Kotoulas[1], and Antonio Corradi[2]

[1] IBM Research, Dublin, Ireland
[2] DEIS, University of Bologna, Bologna, Italy

Abstract. Many systems rely on distributed caches with thousands of nodes to improve response times and off-load underlying systems. Large-scale caching presents challenges in terms of resource utilization, load balancing, robustness and flexibility of deployment. In this paper, we propose a novel distributed caching method based on dynamic IP address assignment. Keys are mapped to a large IP address space statically and each node is dynamically assigned multiple IP addresses. As a result, we have a system with minimal need for central coordination, while eliminating the single point of failure in competitive solutions. We evaluate our system in our datacenter and show that our approach localizes the effect of load-balancing to only loaded cache servers, while leaving cache clients unaffected and also providing for finely-granular rebalancing.

1 Introduction

Massive distributed caches play an important role in large-scale computing infrastructures. For example, Facebook has reported [1,2] that they store tens of TB in a modified implementation of memcached, distributed over hundreds of nodes in a cluster. A set of challenges emerge, when managing caches of this size:

- *Robustness* The system should be robust against node failure and should not have a single point of failure. On the other hand, distributed protocols should have low overhead and be able to respond to failures quickly.
- *Load-balancing* In most caches, key lookups would follow a very irregular pattern, presenting significant skew, as also described in [1]. Typically, HTTP requests and other lookups would follow a power law [3]. Key popularity may shift with time or rapidly change, due to unexpected events.
- *Scalability* Any distributed protocol should be able to scale to large numbers of nodes and have minimal performance impact.
- *Flexibility* It is common and highly desirable to tap into unused resources in a data center. We would like caching techniques that are flexible in terms of demand of computational resources.

This paper introduces Network-Assisted Lookups (NAL), a method to do rapid load-balancing of key-value stores by exploiting the existing IP infrastructure. The key points in our approach are:

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 294–305, 2014.

- *IP protocol as a distributed location registry.* Our system relies on a static mapping of keys to a large space, the static mapping of this space to IP addresses and the dynamic allocation of IP addresses to machines as a way to load balance the system.
- *Exploit existing resources.* The resources in the system need not be homogeneous or have similar throughput.
- *Scalability.* The proposed method scales linearly and is limited only by the number of available (private) IP addresses in the datacenter, which is not a problem in the foreseeable future.
- *Robustness.* Our method relies on IP address re-allocation on the network. As such, there is no central point of failure in the system.

We prototyped NAL in our lab datacenter and evaluated it using key access patterns that are characteristic for skewed data access in a range of trending and established workloads. Our results manifest that NAL manages to be as efficient as competitive approaches (consistent hashing), while achieving to reduce convergence time after load-balancing and also localizing the effect of block rebalancing to clients ”causing” the imbalance.

This paper is structured as follows. Section 2 presents related approaches and delves deeper into the most competitive one, namely consistent hashing, showing its scalability limit via experimental evaluation we conducted. Section 3 presents the architecture of NAL and discusses the workings of the various components, while Section 4 elaborates in our load-balancing algorithm for distributed caching in NAL. We present evaluation results we obtained using our NAL prototype implementation in Section 5 and conclude in Section 6.

2 Related Art and Motivation

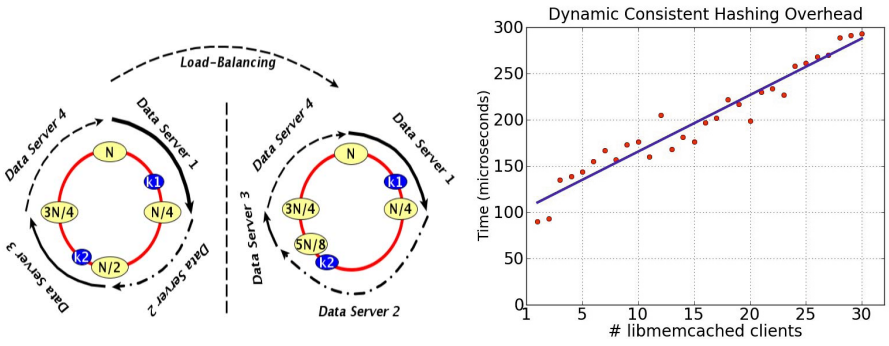
Distributed data-management/processing frameworks typically spread data block-s/records across two or more store locations (e.g. servers). For instance, distributed caching systems such as Memcached [4] and Redis [5] cache recently accessed key-value pairs (e.g. database records) across distributed cache server processes. The latter are typically started at servers, where spare compute, memory and network resources have been harvested. Further examples are parallel/distributed databases and distributed batch data-processing frameworks employing distributed filesystems (e.g. HDFS [6] in Hadoop). A precondition for these distributed stores is the existence of a lookup service entity that maps an identifier to the actual location of the data that needs to be retrieved. Typically, a data record/block is identified by a key. Then, by means of the mapping entity, the key is mapped to the identity (e.g. destination IP socket) of the physical/virtual host (server), where the corresponding data record/block resides. Unless otherwise qualified, we employ in the remainder of this paper the term ”key” to refer to data identifiers and the term ”block” (or ”data block”) to refer to the payload value (e.g. binary object or database record) that is the data unit uniquely identified by exactly one key and retrieved to a processing node as the result of a key lookup and data fetch action.

An additional functional requirement for a data lookup service is the ability to dynamically update the location of data blocks. The latter may change at runtime as the result of a load/store balancing action or to maintain a desired replication factor to

counter data node failures. At the occurrence of any amendment of a block's location (referred to as "block migration" hereafter), the key-to-location mapping entity needs to be updated accordingly to ensure non-intermittent access to distributed data blocks. Distributed stores relying on a centralized lookup service address this requirement by sending location updates to a nameserver. At large-scale, the centralized name service becomes quickly the bottleneck and typically replication (e.g. clustering) is employed to guarantee reasonable lookup latency. However, replication has scalability limitations due to cost and complexity proportionality between the data volume stored and the lookup service capacity required. Distributed hash-tables (DHTs) such as Chord [7]) constitute a scalable and resilient solution; still, they are harder to implement and may degrade performance due to operating at the application/session layer. In addition, having block location "encoded" among data nodes - as is the case with DHTs - can pose security concerns (e.g. in a public cloud environment multiple tenants share the same physical memory resources) and/or contradict the service model. A typical example of the latter case is when the service model mandates the service provider as the sole provider of a persistent and highly-available distributed filesystem service to multiple datacenter tenants. Offering a distributed file service using a DHT and with high availability guarantees may be hard to achieve within such a model, given that e.g. tenants may choose to reboot servers, where part of the block location information is stored.

Following the above discussion on limitations of alternative solutions, we narrow our attention to distributed caching as the best solution for being extended to provide for dynamic data re-balancing. Although various hashing techniques are possible, we focus here on *consistent hashing* [8], a technique that is known to significantly reduce the miss rate during cache server addition/removal. Figure 1a depicts the standard continuum-based implementation of consistent hashing, assuming in this example four data (cache) servers and an integer continuum set S ($S = \{0, 1, 2.., N\}, N = 2^{32} - 1$). Also, each data server is assigned a unique integer in the continuum (e.g. Data-Server-1 $\leftarrow N/4$). Server selection for a given key k occurs as follows: the key is first hashed to an integer value x in the integer set S. The server selected for retrieving the block identified by k is then the server that is assigned to the smallest integer in $\{x, x + 1, .., N\}$. For instance, on the left-hand side of Figure 1a, the block corresponding to $k1$ (resp. $k2$) will be fetched from data server-1 (resp. data server-3).

In commodity distributed caching deployments (e.g. memcached [4]), consistent hashing is static, i.e. servers are statically assigned to the continuum, albeit not necessarily following a uniform key load per server (e.g. to cater for heterogeneity of available memory at each server). Here, we exercise the scalability of extending consistent hashing with load-balancing capabilities, specifically via dynamic re-hashing. The approach has been proposed before in different contexts (e.g. processor memory hierarchy [9]), albeit with scalability requirements not to the level required in this use-case. Figure 1a exemplifies dynamic load-balancing via re-hashing in our toy four-server setup. Initially (left-hand side sub-figure), each server is assigned (for brevity) an equal number of keys along the continuum. Due to data skewness, a fraction of keys served by *Data-server-3* becomes hot (relative to average key popularity), leading to sub-optimal cache performance. To remedy this, the continuum is rebuilt, shifting part of the hot key range to *Data-Server-2* (right-hand side in Figure 1a) and thus providing for a balanced



(a) (b)

Fig. 1. Consistent Hashing: (a) Load-balancing through dynamic continuum rebuilding on four cache servers and (b) evaluation of time overhead due to load-balancing

caching load, adaptive to recent key access patterns. This approach assumes the existence of a centralized key access pattern monitoring and adaptation entity that is though off the lookup and retrieval path.

To evaluate the performance of dynamic consistent hashing to provide for dynamic load-balancing at scale, we created a prototype setup in memcached, using the provided consistent hashing implementation in libmemcached (*libketama* [10]). The latter provides for the ability to dynamically re-assign keys to memcached servers through a weighting mechanism. We also implemented a baseline centralized controller with the sole functionality of notifying memcached clients to rebuild their continua (dynamic re-hashing), together with communicating the set of weights that each client should use as input during each re-hashing cycle to assign memcached server in its continuum. The latter occurs via a simple application-level UDP protocol between the controller and the memcached client.

Using this setup, we measured the time overhead of completing a load-balancing action, specifically by measuring the time that each memcached client takes to finish rebuilding its continuum data structure and reporting the maximum value over the entire set of clients. For each client set size, we repeat the experiment for 5000 times and report the average time overhead over all 5000 repetitions. Figure 1b depicts the results of this experiment with a memcached client set size (actual servers) ranging from 1 to 30. By applying linear regression to the measurements, we obtain the following expression for estimating the time overhead T_{dch} of dynamic consistent hashing (in microseconds) as a function of the number of cache clients x:

$$T_{dch}(x) = 100 + 6.6 \cdot x \tag{1}$$

In extrapolation for a conservative size of a web application comprising 1000 clients, the last equation yields a load-balancing overhead of approximately 7ms just for rebuilding the hashing data structures (i.e. not accounting for the cache misses that will inevitably occur during any cache re-balancing action, regardless of the approach). Obviously, this is a significant penalty, when sub-10ms queries is the desired operating range of target

applications. Even worse, the re-balancing penalty is uniform to **all clients**, e.g. the key lookup has to be disrupted at all clients, even if the load-balancing adaptation is performed to address a hot key range accessed by a small fraction of the client set.

All the above limitations of state-of-the art motivated the Network-Assisted Lookup (NAL) approach presented in the following. Among others, NAL localizes the penalty of load-balancing only to clients that access keys in hot key ranges, while also minimizing the overhead of location resolution during lookup.

3 NAL Architecture

The key paradigm shift introduced by NAL is the following: instead of having fixed network service identifiers attached to data nodes and have these identifiers updated at the lookup service, whenever the location of a data block is changed due to migration, NAL employs a static key-to-location mapping created once and for all at key hash generation time and provides for accurate lookup of arbitrarily migrated data blocks by updating the network identity of the actual location of a block.

Figure 2 materializes the above abstract statement, depicting the architectural amendments to a distributed application (e.g. web application) employing memcached with Network-Assisted Lookups. The embodiment assumes deployment on an IT infrastructure employing an Ethernet 802.3/IPv4 network stack; due to this setup being ”standard” in commodity datacenters, we assume it in the rest of this work, whereby a generalization of the approach to alternative network technologies is beyond the scope of this paper. One of the many application servers comprising the distributed application is shown at the top of Figure 2, where a memcached client is running. Although the memcached client part is typically embedded into the application logic, we depict it for the sake of presentation as a standalone service termed ”lookup service”. Unlike the ”standard” memcached practice of having each key hashed to the single network identifier (hostname or IP address) of a cache server process that potentially holds the data block identified by the respective key, the NAL architecture takes a **static** hashing

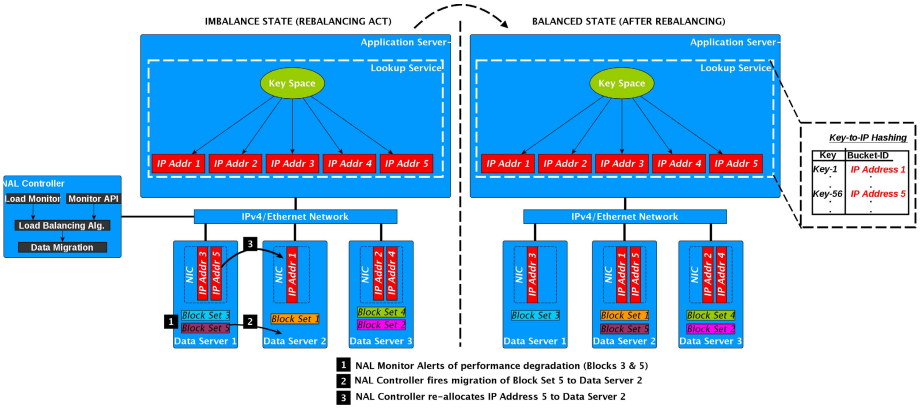


Fig. 2. Network-assisted Lookup Architecture and Data Migration Example

approach of keys to IP addresses, the latter not being bound to a specific cache server process.

The NAL Controller facilitates dynamic binding of IP addresses (that buckets are mapped to) to memcached cache servers, whereby each IP address (and thus hash bucket) corresponds to a set of data blocks ("Block Set"). Leveraging on the feature provided by modern operating systems to have Network Interface Cards (NICs) identified by a plurality of IP addresses, the controller is capable of assigning multiple IP addresses to the NIC of a memcached server and thus mapping multiple buckets to the server. Dynamic address binding is in fact the last step taken by the controller at the event of a block set migration event. Additional functionality implemented at the controller is the continuous monitoring of cache servers' load - either embedded as part of the controller implementation or by interfacing existing memcached monitoring tools via a dedicated API - and the execution of a load-balancing algorithm (cf. section 4) at the detection of caching performance degradation as the result of one or more server overload.

Figure 2 depicts also an example showcasing how the NAL approach achieves dynamic load-balancing without breaking the linear scalability of the distributed caching service. At the event of the NAL controller deducing a cache server overload incident (e.g. Data Server 1 being overloaded in the example of Figure 2), the controller takes a re-balancing action by picking a block set and migrating it from the overloaded (Data Server 1) to a less utilized cache server (Data Server 2). For this, the controller maintains a data structure that maps a bucket identifier to the list of keys that hash to the specified bucket. The re-balancing action completes by also migrating the IP address that the migrated block set statically maps to: in the example of Figure 2, this occurs by de-allocating IP Address 5 from Data Server 1 and allocating it via a newly created aliased interface to Data Server 2. Deriving from this example, it follows that re-balancing in NAL involves interaction **only between a constant number of servers, leaving application/frontend servers untouched** and providing for seamless key lookup while driving the system to a more balanced state and thus yielding better end-user performance.

Load-Balancing Granularity Analysis at Scale. Based on the above specification, it is straightforward that the bucket size used by a NAL deployment drives the granularity of load-balancing, since a bucket of keys (i.e. a block set) is the minimum unit of migration. Let C_{size} be the total cache size (in bytes) available in the system (i.e. the sum of memory allocated to each cache process in the system) and $addr$ be the number of IP addresses dedicated within the datacenter to NAL use. Then, assuming uniform block set size, the *block set size* BS_{size} (in bytes) is lower bounded by

$$BS_{size} \geq \frac{C_{size}}{addr} \tag{2}$$

Due to NAL statically binding an IP address to a bucket and thus to a block set, it follows that the maximum number of block sets that a cache server can hold equals the maximum number of IP addresses $addr_{node}$ that the cache server's network interface can be identified by. Assuming a uniform distribution of total cache size to cache servers, let r be the average cache size ratio of the total cache size C_{size} that is allocated to a cache server. Then, the *block set size* BS_{size} (in bytes) is also lower bounded by

Table 1. NAL dimensioning examples

Cache Size (Total)	Cache Size (Server)	Minimum Bucket Size
64 GBytes	0.5 GBytes	0.125 MBytes
64 GBytes	6.4 GBytes	1.6 MBytes
256 GBytes	0.5 GBytes	0.125 MBytes
256 GBytes	25.6 GBytes	6.4 MBytes
1 TBytes	1.024 GBytes	0.256 MBytes
1 TBytes	10.24 GBytes	2.56 MBytes
1 TBytes	102.4 GBytes	25.6 MBytes
32 TBytes	**32.768 GBytes**	**8.192 MBytes**
32 TBytes	327.68 GBytes	81.92 MBytes
32 TBytes	2 TBytes	512.0 MBytes
132 TBytes	131.072 GBytes	32.768 MBytes
132 TBytes	2 TBytes	512.0 MBytes
512 TBytes	524.288 GBytes	131.072 MBytes
512 TBytes	2 TBytes	512.0 MBytes

$$BS_{size} \geq \frac{C_{size} \cdot r}{addr_{node}} \tag{3}$$

and by combining Equations 2 and 3 we get the feasible set of block set sizes

$$max(\frac{C_{size}}{addr}, \frac{C_{size} \cdot r}{addr_{node}}) \tag{4}$$

Equation 4 governs the finest granularity of load-balancing and is obviously dependendent on system dimensioning. To develop a practical feeling of this bound, we first seek to specify the maximum number of IP addresses we could assign to a network interface ($addr_{node}$ in Eq. 4). Experimentation on a modern albeit "commodity" server running Redhat Enterprise Linux 6.3 revealed that we could assign up to 8192 addresses to a network interface and seamlessly communicate with the server under test on the various IP addresses assigned to it. In the rest, we assume a maximum bound of $addr_{node} = 4096$ IP addresses per cache server. We further set the total number of IP addresses available to NAL to $addr = 16M$, thus allocating an entire /8 IPv4 subnet to NAL. Given the abundance of private IP addresses in a datacenter, the latter choice is not expensive. Using the above setup, we plug target total cache size values to Equation 4, covering a range of systems, from small-scale campus hosting to the largest social networking websites (Facebook reported [2] 800 servers offering at least 28 Tbytes of in-memory cache in 2008). We also assume a broad range of average cache memory size allocated to each per cache server, with the ratio r alternating among following values: $0.1\%, 1\%, 5\%$ and 10%. Table 1 lists the minimum bucket sizes supported by the various setups generated by the process previously described. Evidently, the minimum bucket size supported is always by multiple orders of magnitude smaller compared to the per server average cache memory size. Taking the 2008 Facebook memcached setup [2] as an example (shown in bold font in Table 1), each cache server has 35 Gbytes of memory allocated to each memcached server (assuming uniform distribution of total

cache size to servers). In this example, an approximately 8 Mbytes bucket size is possible, yielding the possibility to do load-balancing at more than 1/1000th of cache size on a per server basis.

It follows from the above that **NAL provides for very fine-grained dynamic load-balancing, even when dimensioned for the most aggressive distributed caching setups that can be thought of today**. Obviously, there is a trade-off between the selected bucket size and the number of buckets in terms of bucket management overhead. It must be noted that the numbers shown in Table 1 are only indicative of the minimum bucket size, while it is still possible to dimension the system with a much larger bucket size to strike a good balance between load-balancing granularity and management overhead.

4 Load Balancing Algorithm

This section presents the load-balancing algorithm we implemented within our proof of concept prototype for the sake of evaluating the NAL approach. It must be though noted that due to the modularity of the NAL Controller (cf. Figure 2), our algorithm can be replaced by any alternative algorithm that interfaces with the rest of the controller modules.

It is straightforward that a well balanced distributed caching system should strive to minimize data access latency and thus global rate of cache evictions, obviously bounded by the caching system dimensions and constrained on the actual key access pattern. Around this intuition, our algorithm shown in Listing 1.1 works as follows:

- At the beginning of each load-balancing iteration, the algorithm reads via the Controller monitor API the eviction rates of all cache servers and computes the average eviction rate across the system.
- In case no outlier is identified, specifically no cache server with eviction rate diverging more than a predefined threshold from the average eviction rate, then no load-balacing action is taken,
- or else the algorithm enters the core of its load-balancing logic towards deciding the list of block sets (sets of key-value stores) that are to be migrated from the most overloaded to the least loaded server.
- The choice of the destination server depends on the state of the caches: if there are nodes without evictions, the algorithm chooses the server with the lower number of requests per second, otherwise it picks the server with the lowest number of evictions per second. The last critical decision taken by the load-balancing algorithms deals with the how many and which block sets to migrate.
- In terms of which block sets to migrate, the algorithm picks the block sets with the lowest number of requests per second, thus keeping the hot blocks at the overloaded data server. The intuition here is that, despite keeping all hot blocks in the overloaded server, the migration of blocks from the latter will free up cache memory and thus decrease the eviction rate due to more hot blocks being able to be kept in cache. An additional advantage of this approach is the continuity of service from cache for a higher number of clients, for the hot block sets are highly demanded. Would it be the hot block sets that were migrated, this would lead to an eviction storm that would impact a larger number of client requests (hot requests).

– The number of block sets to migrate is driven by the parameter N. This allows for adopting a coarse-grained approach when the system becomes very imbalanced, while employing a more fine-grained adaptation (by dynamically setting the parameter N) when the skew between eviction rates is reduced and the system is more stable.

Listing 1.1. Pseudo-code of Load-balancing Algorithm with NAL

```
Set M // overload threshold (eviction rate)
Set N // normalization factor for deciding popularity of blocks to be migrated
LOOP
{
  FOR all the nodes in cache server list
  {
    READ #evictions/sec into array E
  }
  AvgEvcts=Average(E)
  FOR each item e in E
  {
    Max_Evcts = Max(Abs(e-AvgEvcts),Max_Evcts)
  }
  IF Max_Evcts < M // system is balanced
  {
    WAIT T_check seconds
  }
  ELSE // start load-balancing/block migration
  {
    S = node with max #evictions/sec
    Max_reqs = #requests/sec of S
    L = list of x nodes with least evictions
    IF L is not empty
    {
      D = node with min #evictions/sec
    }
    ELSE
    {
      D = node of L with min #requests/sec
    }
    Min_reqs = #requests/sec of D
    Reqs_to_move = (Max_reqs - Min_reqs) / N
    B = list of less accessed Block Sets of S with sum of #requests/sec greater
        than or equal to Reqs_to_move
    MOVE B from S to D
    WAIT T_check seconds
  }
}
```

Our algorithm scales with the number of cache servers and is highly responsive within the dimensions of the most aggressive setups (O(100) servers). As part of our ongoing work, we are currently formalizing its efficacy against equivalent standard algorithms (e.g. bin packing), while also experimenting with hierarchical monitoring so as to relieve the strain on the controller from having to poll all cache servers for cache performance statistics.

5 Evaluation

We deployed our proof of concept prototype implementation of NAL in our lab datacenter using 30 servers in total. All servers comprise two Intel Xeon X5670 6C processors,

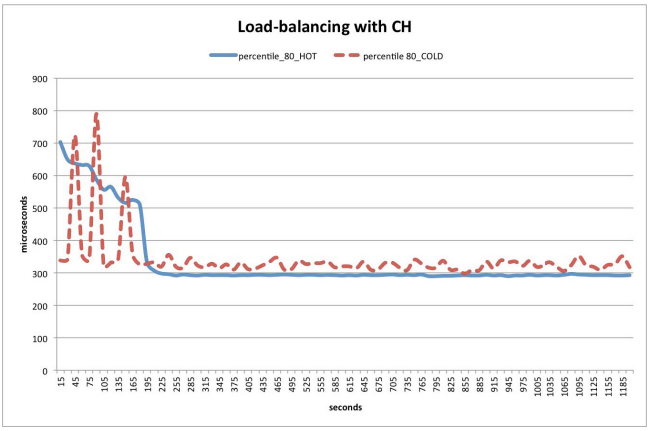


Fig. 3. Evolution of 80th-percentile of key-value pair retrieval latency when load-balacing with dynamic consistent hashing

128GB DDR3 RAM and an 1TB SATA disk, while interconnecting via a full-bisection 10GigE network. Eight (8) of the servers are used as cache servers (memcached), one (1) server is running the persistent data store (MySQL database with 10M entries of average size 100K each) and the rest of the servers are running client access code, i.e. code that fetches key-value pairs randomly from the hierarchical data store.

Each client initially attempts to fetch a key-value pair from the cache (using libmemcached) and only in the case of a cache miss, it then fetches the requested key-value pair from the database server. To drive the system to an imbalanced state for the purpose of evaluating the efficacy and performance of NAL, a fraction of the client set requests keys following a Zipf distribution, while the rest of the clients request keys uniformly. This is a known key access pattern evident in various applications, e.g. web applications requesting a popular web page or object, batch document/chunk processing frameworks etc., and is exactly the root cause of imbalance in distributed data stores. We initially run the system with uniform key access across all clients until the system stabilizes to a constant average eviction rate (steady state) and then a fraction of the clients enter the Zipf access pattern mode. Figures 3 and 4 depict the 80th percentile of key-value retrieval latency across all clients in the system for dynamic consistent hashing and NAL respectively [1], whereby we have dissected the client set to a hot (clients using Zipf access pattern) and a cold (clients following uniform access pattern) set. By inspecting the two charts, one can easily observe the anticipated superiority of NAL over dynamic consistent hashing: a) while consistent hashing causes a latency fluctuation to the cold set of clients due to dynamic continuum rebuilding regardless of the type of client, **NAL leaves cold clients unaffected, while halving the latency of hot clients** and b) NAL manages to achieve a balanced state by almost 2x faster than consistent hashing, due to the localized nature of the balancing act, as opposed to consistent hashing convergence time, which is proportional to the number of clients.

[1] We start reporting results at the time Zipf distribution mode comes into effect,i.e. steady state latency is not shown

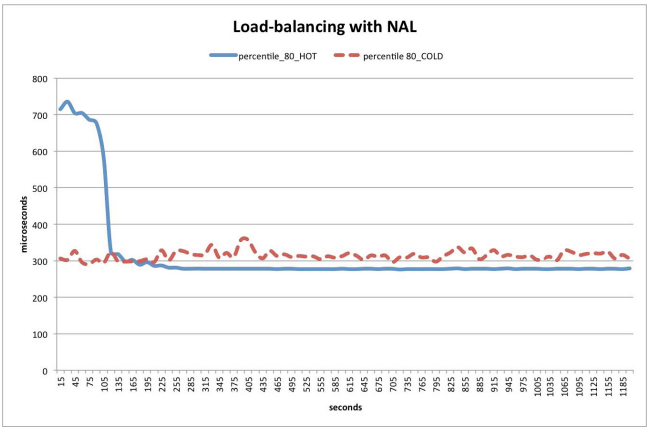


Fig. 4. Evolution of 80th-percentile of key-value pair retrieval latency when load-balacing with NAL

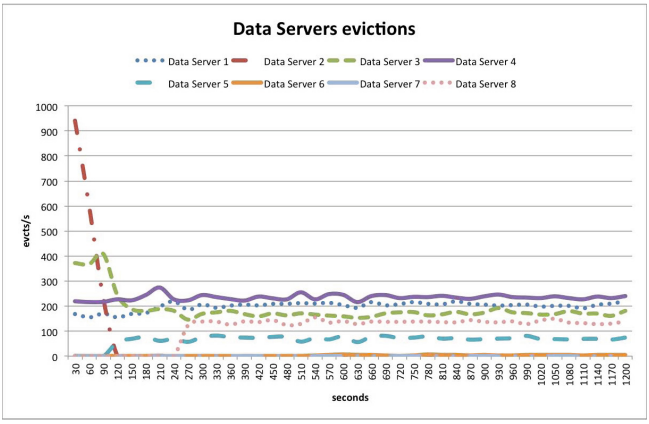


Fig. 5. Time series of eviction rates across cache servers when load-balancing with NAL

To showcase the granular adaptation brought by our load-balancing algorithm, we also depict in Figure 5 the time series of eviction rates across the eight cache servers when using NAL. We observe that the algorithm reduces initially the eviction rate of the most overloaded server (Data Server 2), then proceeding with the newly most overloaded server (Data Server 3) and finally maintaining a fairly balanced system by fine-tuning among all servers. We note here that throughout our evaluation, CPU load due to the NAL Controller has never exceed 1%, memory footprint was only 4.25MB and control network throughput in-and-out of the controller was always less than 4 Kbytes/s.

6 Conclusions

This paper presented a novel scheme that facilitates scalable, fast and fine-granular load-balancing in distributed key-value data management systems. Leveraging on the ability of modern servers and operating systems to alias network interfaces, we disconnect key lookup from actual key-value pair location and thus cancel the need for dynamically updating the location of a key, whenever the key-value pair location changes. We have presented a comprehensive architecture that in turn we prototyped in our lab datacenter, showing through experimentation that our approach achieves equal efficacy to state-of-the-art, while being 2x faster for the setup size tested. In larger setups, we expect the convergence gain of our approach to be proportional to cache system size, when compared to competitive approaches.

References

1. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al.: Scaling memcache at facebook. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, pp. 385–398. USENIX Association (2013)
2. Facebook Note: Scaling memcached in facebook (2012), `https://www.facebook.com/note.php?note_id=39391378919`
3. Adamic, L.A., Huberman, B.A.: Zipfs law and the internet. Glottometrics 3, 143–150 (2002)
4. Fitzpatrick, B.: Distributed caching with memcached. Linux J. 2004, 5 (2004)
5. Redis: Redis website (2014), `http://redis.io/`
6. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, pp. 1–10. IEEE Computer Society, Washington, DC (2010)
7. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 149–160 (2001)
8. Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y.: Web caching with consistent hashing. Computer Networks 31, 1203–1213 (1999)
9. Chang, K., Loh, G.H., Thottethodi, M., Eckert, Y., Connor, M.O., Subramanian, L., Mutlu, O.: Enabling efficient dynamic resizing of large dram caches via a hardware consistent hashing mechanism. Technical Report 2013-001, Electrical and Computer Engineering Department,Carnegie Mellon University (2013)
10. Libmemcached: Libmemcached website (2014), `http://libmemcached.org`