Miguel da Silva de Brito Cordeiro

Degree in Computer Science

Rethinking Distributed Caching Systems Design and Implementation

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Leitão, Assistant professor,
Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa

Co-adviser: Vitor Duarte, Assistant professor,
Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2018

Abstract

Distributed caching systems based on in-memory key-value stores have become a crucial aspect of fast and efficient content delivery in modern web-applications. However, due to the dynamic and skewed environments, to which they are typically subject to, several problems arise in the form of load imbalance.

This thesis addresses the sources of load imbalance in caching systems, which we classify as three principal source vectors: i) data placement, which relates to distribution of data items across servers; ii) data item access frequency, which affects the number of requests each server has to process; and iii) data item size, which impacts memory management.

Furthermore, we also provide several strategies to overcome the sources of imbalance in isolation. As a use case, we analyse Memcached, its variants, and propose a novel solution for distributed caching systems.

Keywords: Distributed Caching Systems , Memcached , Selective Replication , Key Placement , High-Availability , Cache-Coherency , Load Balance , Memory Management, Concurrency

Resumo

Os sistemas de cache distribuídos baseados em armazenamento de pares chave-valor em RAM, tornaram-se um aspecto crucial em aplicações web modernas para o fornecimento rápido e eficiente de conteúdo. No entanto, estes sistemas normalmente estão sujeitos a ambientes muito dinâmicos e irregulares. Este tipo de ambientes e irregularidades, causa vários problemas, que emergem sob a forma de desequilíbrios de carga.

Esta tese aborda as diferentes origens de desequilíbrio de carga em sistemas de caching distribuído, que classificamos em três vetores principais: i) colocação de dados, que se relaciona com a distribuição dos dados pelos servidores; ii) frequência de acesso aos dados, que afeta o número de pedidos que cada servidor deve processar; iii) tamanho dos dados, que afeta a gestão de memória.

Também demonstramos várias estratégias para reduzir o impacto proveniente das fontes de desequilíbrio quando analizadas em isolamento. Como caso de uso, analisamos o sistema Memcached, as suas variantes, e propomos uma nova solução para sistemas de caching distribuídos.

Palavras-chave: Systemas de caching distribuidos , Memcached, Replicação seletiva , Colocação de dados , Alta disponibilidade , Coherencia da cache , Equilibrio de carga , Gestão de memória , Concurrencia

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"What is considered slow depends on your requirements. But when something becomes slow it's a candidate for caching- High Scalability, A Bunch Of Great Strategies For Using Memcached And MySQL Better Together

As the evolution of the web progresses, so do their challenges, massive use of web services and applications leads to an increasing number of client requests with the need for a fast response, for instance in the case of Facebook, Youtube, Flickr, Twitter, or Wikipedia. This need for fast data retrieval from web services requires a more efficient design than simply fetching data from a potentially slow database or set of databases.

A study performed by Microsoft and Google [29] demonstrates that response time matters, revealing that server delays have a substantial negative impact. In particular, delays under half a second affect business metrics, with a direct translation regarding revenue. Moreover, they show that the cost of delays increases over time and persists, even when the source of the delay is removed.

A Web Application deployment is commonly divided into three main components: i) Load Balancer, ii) Web-Logic, and iii) Storage. The Load Balancer is usually seen as an ingress point for all client traffic; it has some internal policy (e.g. Round Robin) on how to distribute the incoming request across the multiple existing Web Servers that materialize the Web-Logic component.

The Web-Logic component, which in addition to web servers might also contain application servers, is responsible for handling the business logic for the application as well as handling the presentation of the application (and results of operations) for the end users. To enable multiple web/application servers to operate independently, servers materializing this component are usually stateless, which enables client operations to be handled by an active server (and also to quickly increase the number of such servers

when the system is under heavy load). Therefore, the applications state is managed by the storage component.

The Storage component might be composed from one to several storage systems, SQL, or NoSQL databases, or any other persistence storage (such as a distributed file system). Typically, a web/application server, to handle the client request accesses one or multiple storage services. Upon getting a reply from such storage services, the web/application server composes an answer to provide to the client and sends it.

When designing a web application, the storage is a crucial component of the application whose design has to be addressed carefully, due to its impact on the overall application performance when dealing with data-intensive applications. Some designs might include a single persistence store such as MySQL, DynamoDB, Cassandra, or Amazon S3, but a conventional approach is to store data over multiple systems, each of the systems optimized for a particular data type.

In these cases, the response time is dependent on the slowest storage system. Still, there are several factors that weight in the response time to a web/application server such as the computational cost of executing a given query, the size of the receiver queues in the database, and network communication time. In particular relational databases tend to particularly difficult to scale when subjected to heavy loads.

Non-volatile storage is several orders of magnitude slower than RAM and as often seen in real workloads, both reads and writes follow a random access pattern for disk I/O where this random pattern wreaks havoc on performance. As a way to reduce the latency in such data-intensive environments, one can store the replies sent to clients for frequent read operations. This avoids the application servers to consult the storage component allowing to reuse the previously computed answer and hence, lower the latency of such requests. In general, this entails the use of caching systems.

A caching system is typically materialized by an in-memory data store that provides fast access to some previous computations or duplicates frequently accessed stored data item that usually resides in a durable way in one of the storage systems of the web application. The web/application servers typically exploit this Caching component. Caching systems usually follows a principle of non-interference, where they do not interact with other storage systems directly. This brings simplicity to their design and avoid increasing the complexity of the storage component.

Caching is commonly associated with the fact that the information, contained within it, is transient, as it will only reside in the cache for a given amount of time, as well as the fact that the cache is limited in space. This restriction in space demands a priority over items in the cache. Eventually, some item will have to be eliminated from the cache for some other data item to take its place. An eviction policy usually regulates this. The most common being Least Recently Used (LRU) and Least Frequently Used (LFU).

When clients make a request, the web/application servers will first try to access the cache, before turning to a slower data store. If the requested item can be found in the cache, it will be counted as a *cache hit*, if not, it will be a *cache miss*. Evidently, a higher

number of cache hits translate directly to faster response times, as it avoids resorting to slower storage systems.

When considering a single server working as a cache that is accessed by all web/application servers, one can obtain some improvements in terms of access time for an application, but this also implies some limitations, making it a single point of failure, and a bottleneck since all web/applications servers are simultaneously making requests to a single cache server.

Most systems try to overcome these limitations with distributed caching systems where multiple cache servers co-exist. In these deployments each server is responsible for a fraction of the data items of the whole system, allowing for requests to be divided across all cache servers, improving the scalability of the caching system. Even if a server fails, another will take over its responsibilities, which significantly improves fault-tolerance.

There are several examples of caching systems in the literature that are deployed in the world. Each of these systems has their own properties, such as: MICA [14] that achieves extremely high throughput; Aerospike [34] that is optimized for flash-memory, unlike most allows data persistence by design; Redis [25] which resorts to dynamic memory allocation, and Memcached [1] that employs static memory allocation.

Problem Statement

The access patterns displayed in most systems follow a Zipf distribution, for caching, this is relevant since it improves separation of content based on popularity, but inside the cache, at each node, it can be highly disruptive, causing bottlenecks and leading to unstable response times. Since response time is a crucial aspect that impacts the user experience while using a service, achieving faster response times is a concern, that can be mitigated by better cache usage regarding memory management and access patterns.

In particular, the goal of this thesis is to study, new mechanisms and combinations of techniques that allow to boost the performance (in terms of cache hits, throughput, and latency) of distributed caching architectures. We will focus in particular on the popular Memcached system which is highly deployed in real systems throughout the world.

Expected Contributions and Results

In this thesis we expect to produce the following main contributions and results:

- A novel design for building and managing distributed caching systems that integrates multiple aspects, in particular regarding the memory management for each individual server, improve load distribution, and selective replication.

- A prototype of our solution based on the highly popular Memcached system

- An experimental evaluation of our solution, compared with other alternatives, that focus on the benefits achieved by each design aspect and their combination.

Document Structure

The rest of this document is structured as follows:

- Chapter 2 - presents related work, focusing on the standard Memcached, variantes, eviction policies, and key sources of load imbalance in caching systems.
- Chapter 3 - discusses our plan to achieve the desired cache properties to mitigate the sources of load imbalance in caching systems, enabling an overall better performance.

Chapter 2

Related Work

This chapter discusses multiple aspects and existing solutions in the context of in-memory caching systems. For that purpose we describe possible caching architectures, with a step-by-step evolution in paradigm from a single node to distributed caching systems, we then discuss variants of currently existing systems (with a particular emphasis in Memcached which we use as a case study), key distribution techniques, and quality of service aspects. As well as other as aspects that are present in storage systems such as replication techniques and consistency.

2.1 Preliminaries

Throughout this chapter we discuss many solutions that resort to pseudo-random generators and cryptographic hash functions. For completeness in this section we provide a description of what is assumed regarding these abstractions.

A Pseudo Random Number Generators (PRNG) is a class of deterministic functions that take as input a seed and generates sequences of outputs that try to approximate a true random number generator, most guarantee a certain level of randomness within a sequence of generated numbers but with few guarantees regarding the correlation between parallel sequences generated using different seeds, which can lead to more predictable distributions and less fairness [20].

A cryptographic hash function takes an arbitrary sized input and generates a fixed size output with several properties:

Deterministic: The same input always results in the same output.

Uniformity: Every output should have the same generation probability for the output values range of the function.

Fast Computation: The operation should be inexpensive in terms of CPU cycles.

Avalanche Effect: Small changes in the key cause the output to change significantly.

Collision Resistance: It should be difficult for two different inputs $m1$ and $m2$ to have $hash(m1) = hash(m2)$.

2.2 Cache Overview

A cache is responsible for storing limited amounts of frequently accessed data, usually in faster memory (RAM), to minimize the access times for that data. When talking about main memory it implies that it is normal RAM, not focusing on the inner layer of caching such as L1, L2, and L3, which is fully managed by the hardware. From the point of view of caching systems, we are only concerned if the intended data is present in the process (dynamic or static allocated) main memory.

Data items in a caching system, usually also exist in a persistence (and slower) data store, or they can be the result of performing computations over multiple inputs stored in one or multiple persistence data stores. This technique offers the potential to reduce the overall latency of requests, particularly when applied to systems with workloads composed mostly of read operations. This happens because caching avoids accessing these slower storage systems, such as databases or hard disks. As caches have limited space (as RAM is limited), not all data items can be maintained by them. Hence, when cache space is exhausted, eviction policies are required to accommodate new data items.

Eviction policies define which data items in the cache should be removed in order to accommodate new elements. These policies strive to maximize the probability of having in the cache items that will be requested in future accesses.

If a requested element exists in the cache, it will result in a *cache hit*, otherwise it will result in a *cache miss*. The fraction of cache hits over all the requests translates to a hit-ratio, and the complementary metric is called miss-ratio. A key performance indication for caching systems is the hit-ratio, where higher values imply better performance.

2.3 RAM Key-Value Stores

There are several layers of caching through out most systems, that can divided in two groups: (1) proxy cache, and (2) service cache.

Proxy caches are mainly accessed by clients and employs a philosophy of caching data closest to the clients. In order to achieve it, a common approach is to have hierarchical caching where the root cache is the only one that interacts with a persistence storage system [8] and all other levels cache contents that are present in caching layers above it.

Service caches are the caching systems in which we will be focusing. These are for exclusive use of an application on the server side. Most of theses caching systems present an interface based on key-value stores, in which a data item is defined as an entry indexed

by an unique identifier. Typically, some meta-data is associated with each data item stored in the caching system. This meta-data can include control information such as time stamps, vector clocks, locks and marked bits

2.3.1 Single Cache Instance

A cache system can be deployed as a single instance. In this case all data objects currently cached will be in the same machine. The benefit of such deployments is in their simplicity, where there is no replication, which inherently ensures a high degree of data consistency. On the other hand, a single instance deployment is only suitable for small application deployments, as machines have limited RAM. Additionally, in this case the cache can become a single point of failure (SPOF), that might impact user perceived latency.

In this setting, to accommodate data items, one has to scale up the machine supporting the caching system, by adding more resources (such as RAM and CPU capacity). Additionally, the cache can also become a bottleneck, which can lead to increased latency on answering to user requests.

2.3.1.1 Local Eviction Policies

Considering the deployment of a single instance with a limited amount of RAM, the management of the cache contents becomes a crucial aspect as to ensure high hit-ratio. Since pushing new data items into the cache, will eventually require evicting some of the old ones still present there.

The problem resides on the automatic selection of which data items to evict. For that, there are several algorithms [38] that can be used and have demonstrated good performance at least for some workloads: Least Recently Used (LRU), Least Frequently Used (LFU), Random Marking (RMARK), Reverse RMARK (RRMARK), LRU-K, 2Q policy, and GAVISH among others. Most of these algorithms strive for separating data items either in classes, or to classify data items in relation to each other, typically taking three factors into account to guide the selection of the data items to evict :

Access Frequency Rate at which accesses to a data item occur in a bounded period of time

Access Recency An indicator of when was the last access to a data item

Overhead Both memory usage and computational cost for management.

For completion, we now briefly describe some of the more popular eviction policies:

FIFO Evicts the oldest data item residing in the queue, without considering either recency or frequency.

LRU Evicts the least recently used items first, which causes it to only take into account recency, leading to considering frequently accessed and recently accessed data items similar.

LFU Keeps track of all the accesses made to a data item, evicting the data item with a lower number of accesses. This causes it to only take access frequency into account which leads to the frequency values associated with data items to lower slowly, leading to a poor capacity of the eviction policy to adapt to changing workloads. This can lead to the eviction of newly created data items due to low number of accesses.

LRU-K combines recency and frequency. This strategy takes into consideration the K most recent accesses of a data item, storing K-1 references for each data item access. It takes into account an estimate of access arrivals for eviction priority. If a data item was accessed less than K times it will have higher eviction priority. LRU-2 can be used storing the penultimate access time in a priority queue achieving logarithmic complexity. This method improves LRU by using a more aggressive exit rule to quickly remove cold data items from the cache that become unpopular (i.e,less frequently accessed).

2Q policy [12] It is composed by three queues: A1, A2, and the shadow queue. The A1 queue stores cold data items eligible for eviction using a FIFO policy, and uses the shadow queue as a complementary side, which stores only the identifiers of data items that have been accessed more than once while residing in the A1 queue. Queue A2 is managed using a LRU policy and stores data items that have received hits while referenced in the shadow queue.

RMARK combines both recency and frequency, this is achieved by separating the data items in two sets, marked and unmarked, where only unmarked data items are eligible for eviction. New data items and accessed data items are moved to the marked set. When the unmarked set becomes empty, the marked data items become unmarked. This method allows some distinction between "hot"and recent data items taking frequency into account, without the overhead introduced by LFU.

RRMARK is very similar to RMARK, with the key difference that a new data item starts as unmarked. RRMARK, in relation to RMARK, is more biased towards frequency.

GAVISH takes an interesting approach, where unlike the previously discussed policies, it uses four hierarchical lists, where new items enter the bottom of the hierarchy. With each access a data item moves to the head of the list above, and each time the list exceeds its size, the tail data item is moved to the middle of the list bellow. If this happens at the bottom list, the data item is evicted from the cache. This solution allows for a clear separation of access frequency by levels with low overhead, where high access frequency data items will mostly reside in top levels, and the lower

access frequency data items will stay in the bottom levels of the cache. Warm data items (that lie between very popular and rarely accessed data items), will move cyclically between the middle queues.

These eviction policies are local decisions that aim at increasing the hit-ratio of a single node. However, these decisions do not take into account the global system, where emergent behaviors might impact negatively the existence of other cache servers making their own decisions.

2.3.2 Distributed Caching Systems

A distributed cache is typically deployed in a cluster of nodes that offers a logical view of the cache as if it were single node. This is commonly accomplished through horizontal partitioning using data sharding, where data items are split among the nodes using a deterministic algorithm, usually consistent hashing. This allows load distribution across nodes, where each access targets the node that holds the data item requested. Also, it allows a simple way of scaling-out when in need to accommodate more data items, simply by adding more nodes to the system. This comes at a great advantage since scaling-up is not cost-effective, where the cost of adding more resources is not linear with the improvements achieved by it.

When in the presence of a node failure, it does not result in complete outage of the cache, as the system will still continue the normal mode of operation, taking only a small overhead as another node takes over the responsibilities of the failure node.

This scheme allows for the system to support high amounts of cached data, but as complexity grows, so do the challenges when trying to extract the full capacity of the system. we now study the sources of imbalance on the system that force some nodes to struggle with the load they are subject to, resulting in a decay in performance.

2.4 Load Imbalance

Nowadays websites can store and process very large quantities of data, this leads to a large number of requests being processed. In such scenarios, any single cache instance would have its capacity easily overwhelmed, and so, distributed caching systems are commonly seen as a more suitabe solution, since it can easily scale out, allowing it to better adapt to the applications needs [37].

Some core features that are desirable in caching system are high hit-ratio, fast response times with low variance. But typically applications are subject to accesses that can be described by a Zipf distribution with varying data item sizes that change their popularity over time. All these aspects cause some instability across caching servers that might become saturated with requests, leading to degrade performance when accessing data items in those servers.

This imbalance therefore should be tackled as to normalize the load imposed over each caching server, instead of allowing that by chance some nodes become clogged with requests. Redistribution of load across several nodes provides a solution to deal with the skewed workloads that the system might be subject to, while allowing it to adapt and change accordingly.

There are three main vectors as a source of load imbalance in a caching system: (1) Data Placement, (2) Data Access Frequency, and (3) Data Object Size. Any of these factors, even in isolation has the potential to degrade the cache performance and waste system resources.

We now discuss each one in turn.

2.4.1 Data Placement

Achieving uniform distribution in data placement comes as a key aspect in distributed storage systems. Distributed storage systems are composed by a set of nodes with limited amount of RAM, where the aim is typically to fully utilize the available RAM before resorting to evictions. Cases of non-uniform data placement distributions lead to memory capacity under usage, that requires more nodes to accommodate the same global amount of data items.

Previous work [11] has shown that, non-uniformity data placement distributions with a maximum variability of 10% requires up to 9,1% more machines in order to accommodate the same global amount of data items compared with a scenario with uniform data placement distribution.

Therefore, achieving uniform key distribution is a very important feature in order to achieve near optimal data placement, which allows to mitigate local sources of imbalance. However, there is set of properties that are essential for any key distribution scheme to ensure that it is practical:

Low Overhead In order to achieve low overhead decisions should be made locally with the minimal amount of information needed (e.g, server and object names).

Load Balancing Each node should have equal probability of receiving an object as to uniformly distribute the load between nodes.

High Hit Ratio All clients must agree on key placement, ensuring that if a data item is present, it will be retrieved. Hence, retrieval of data items will yield the maximum utility, in terms of hit ratio.

Minimal Disruption When a node fails, only objects which were mapped to that node must be reassigned.

Distributed K-Agreement Each node, must select the same K nodes to place the same data items, based on local information.

2.4.1.1 Simple Approach

A naive approach for key distribution is having all the keys stored in the clients main memory with server mappings. This is an expensive and unpractical way of storing key-server mapping, as the number of keys will probably take a large toll on the client memory. This also requires clients to coordinate among them to agree on the assignment of each key. Furthermore, if added the possibility of server failing and joining the current pool of servers, each time this happens there is a need to re-map a large set of keys. To avoid this computationally expensive process, there are more clean and efficient methods that take advantage of some properties of a hash functions.

The $HashTable$, a well known data structure, recurs to a simple hash function $hash(O)$ $mod(N)$, that allows to evenly distribute a given number of data item identifiers (O) through the positions of the $HashTable$ array of size N. Or in this case, N servers by assigning a distinct value to each server between $[0, N[$ similar to an array position. This solution achieves close to a uniform distribution of objects per server.

There is however, an issue when we take into account the possibility of server failures and servers joining the system, as the resize operation in the $HashTable$ example.This demands a remapping of all to keys to servers, turning it into a very expensive operation.

2.4.1.2 Consistent Hashing

This method, described in [13], is based in a single computation of a hash function $hash(O)mod(U)$, where O is the object name, or some unique identifier, and U the range of acceptable values that map into the range used by servers as their own identifiers.

To solve the problem of servers joining or leaving the system, this method instead of considering N, the number of servers, considers U as the upper bound on the range of values the solution is considering, being $U >> N$. For this to work properly there is a need for a ring-like structure where each server has a given bucket $[a, b[$, there are no two servers where buckets intersect each other, and the union of all buckets is $[0, U]$.

The hash function simply points to a value θ in $[0, U[$, the object will fall in the bucket where $\theta \subset [a, b[$. A server joining membership simply means that a bucket will be split and server leaving means two existing buckets will be merged.

However, simply assigning a bucket to a server is not enough, since if buckets are not all the same size The number of keys assigned to each server will be very distinct.

To overcome this aspect, a common solution is to rely on the techinique popularized by *Dynamo* [6, 7], which is notion of virtual nodes. The key intuition is to simply increase the number of buckets assigned to each server in order to have a better approximation of a uniform distribution. These buckets have an arbitrary position in the identifier space (i.e, the ring), meaning that a server is unlikely to have two buckets where $[a, b[\cup[b, c[= [a, c[$. Some variants of this algorithm achieve $Distributed\ k-agreement$ either by following the ring structure clock-wise k-1 hops from the initial node to where the key of the object is mapped, or by concatenating the data item identifier with a salt value in the range $[0, K[$.

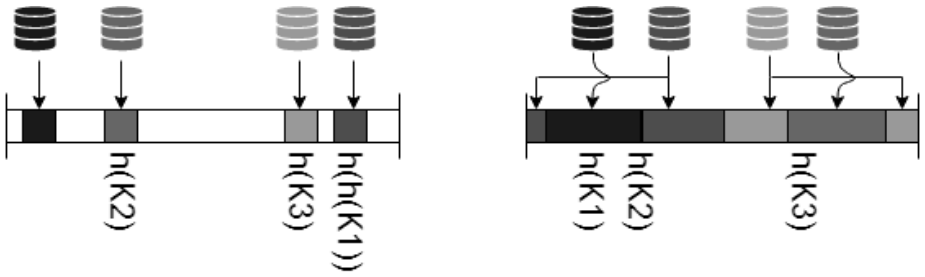
2.4.1.3 Highest Random Weight

This method described in [33] exploits the properties of a hash function to achieve global server ordering for a given object, using $hash(O||S_i)$, where O is the data item identifier, $||$ is concatenation operation, and S_i the server identifier.

The algorithm is simple and intuitive: for a given object, the $hash(O||S_i)$ has to be computed for each server identifier. This output can be displayed in a ordered list, where the first position represents the home server for the data item. The uniform distribution property is expressed at each position of the list, as long as all clients choose the same positions. *Distributed k-agreement* can be achieved by simply selecting the first K positions of the ordered list.

2.4.1.4 In Line Data Placement Algorithms

These class of algorithms typically represents a node as a segment over a finite space defined as a line, the segment size assigned to a server is proportional to the capacity of that server (in our case all segments have the same size). The segment placement over the line is deterministic and do not overlap, so all clients can compute it independently.



a) Random placement with gaps b) Random placement with full line

Figure 2.1: In line Random placement strategies

There are two major segment placement policies: with gaps (fig.2.1a) or, using the full identifier space (fig.2.1b). Segment placement with gaps is used in SPOCA [5] , ASURA [11], and SOURA [39], but require a retry mechanism in order to pin point the target segment, where this can be accomplished by iteratively generating random number using a PRNGs with the data item identifier used as the seed, until the output value is contained within a segment, or by using a hash function iteratively with its previous output as input.

Segment placement using the full line space such as Random Slicing [20] requires that the client to have a full causal history of all the changes in the membership to compute the current range of each segment. This is only needed for bootstrap, and to accomplish it, this scheme requires a persistence store to supply the initial view to new clients. Based on these mechanism, every client can compute the optimal segment separation in order to have all segments with the same size in the identifier interval. Compared with solutions

Table 2.1: Comparison between key distribution algorithms

Strategy	Fairness	Memory Usage	Lookup Time
Consistent Hashing (fixed)	Poor *	High	Low
Consistent Hashing (adapt.)	Moderate	High	High
Highest Random Weight	Good	Low	Very High**
Random Slicing	Good	Low	Low

* Fairness increases with the amount of nodes/ virtual nodes.
** Lookup time increases with the amount of nodes.

that allow gaps, these scheme only requires one computation to find the target segment for an object.

Most of theses algorithms achieve *Distributed K – agreement* by iteratively executing the same deterministic function using the previous value as input, until it hits K different segments.

2.4.1.5 Discussion

In caching systems, response time is a fundamental concern, and since network latency contributes to increase response times. Most distributed caching systems avoid the use of Distributed Hash Tables (DHT's) such as Chord [30, 31], to avoid multi-hop routing. Instead, solutions where each node knows the global membership of the system are favored.

All algorithms described here allow weighted key distributions, but avoid it since most systems are homogeneous in terms of capacity of each server.

The key feature in these algorithms is to always use a function that, a priori, ensures uniform distribution [19], most algorithms use this factor so that only the size of a segment (relative to the total identifier space) will have an influence in the amount of data items that each server becomes responsible for. If each server is responsible for a segment of equal size, and assuming uniform distribution of objects across segments, the distribution among servers tends to be similar. This can be seen in most of these algorithms, and is the main cause of load skew in consistent hashing, that tries to minimize it through the use of virtual nodes.

As a cross reference between work provided in [20] and [11], we can establish a base comparison between several of these algorithms in terms of fairness, memory usage, and lookup time:

Furthermore, Consistent Hashing with a high amount of virtual nodes incurs in high space overheads in order to maintain the logical ring between all servers, when compared with the other algorithms. Highest Random Weight, even though it is the algorithm that achieves the most uniform distribution, requires the computation of N hash functions for each data item identifier in order to map it to a node (N is the total number of servers). This can have a non-negligible overhead on clients. In line data placement algorithms

with gaps, there is no bound on the number of iterations to assign an object to a segment. Random Slicing, even though it requires a persistence store, seems the most balanced and computationally efficient solution.

2.4.2 Data Access Frequency

Frequency of accesses to data items is a key aspect that should be considered by a caching system to determine which elements should be kept in the cache. But at a smaller scope, when looking at each caching server individually, there is a notion of load associated with it. This load can be seen when looking at the amount of requests received by a server and the amount of responses it can produce in a bounded time window.

Typically, each node is subject to different loads according to the amount of data items it holds and the access frequency of each of those data objects. The presence of high access frequency data items can be highly disruptive, causing a bottleneck in the server reception queues, that affects the response times of all requests received by that server [28].

Since it is unavoidable to have some data objects that are accessed with a much higher frequency, it becomes relevant to identify those objects and unsure that globally, all servers share these objects as to enforce a uniform local distribution in terms of the number of request received per time unit. These are multiple ways to identify such "hot"objects both locally (in the context of a server) and globally (across the servers). We now discuss these techniques in some detail.

2.4.2.1 Local Detection

In local detection, decisions are made by each caching server individually, while disregarding the rest of the system. Spore [10], is an example of such system that resorts to one of these mechanisms. In this system, selective replication is used based on local highest access frequency. In order to detect if a server is under heavy load, it marks a threshold in each of the caching servers receiver queues. This threshold represents the maximum admissible latency that the application can accept. Once this threshold is reached in a server, there is a need to relieve the load on that server in a fast and efficient way. To do this, Spore strives to identify and replicate the data item with highest access frequency, as this will share the local in serving that object with another server, promoting load reduction and ensuring the availability of the data item.

2.4.2.2 System Wide Detection

To achieve System Wide Detection of "hot"data objects one usually relies on mechanisms that compute a global view of the access frequencies for all cached data items. Based on the popularity of such data items the system can infer the appropriate measure to be taken as to avoid having a some servers overloaded. To achieve this, there are several approaches such as (1) Inter-node communication or (2) client side sampling.

Inter-node communication requires each server to keep track of each of their data items popularity and exchange their information with others servers. This introduces a large amount of communication to achieve a concrete and up-to-date view of all data items access frequency. Such solution incurs in additional complexity, while generating more traffic for each of the servers in the caching system.

Client side sampling is based on a simple assumption that there are a set of alerts (typically the application server) that distributes requests to servers originated by end users. Assuming that each client observe a uniform sample of request, if a data item has high access frequency it will also have high access frequency in each of the clients (compared with the other data items). Allowing each client to infer locally which are the popular data items based on their local view and without additional communication.

The Zebra algorithm [5] uses a sequence of bloom filters, where each bloom filter represents requests for a given time interval (tick). This sequence has a fixed length where after each tick a new bloom filter is added and the oldest discarded. Content is deemed popular based on the union of the bloom filters. The sliding window of bloom filters allows for data item popularity to increase and decrease over time.

Using a sampling mechanism like Zebra, allows the use of a Two-Tier caching, where the client side relieves the load on the caching servers by caching themselves, the most popular data objects. This overcomes the fact that serving very popular data items degrades the overall response time for several other data items. This is based on the concept of a front-end cache that stores very few elements, which by itself results in faster response times for those data items. As explained in [28] *any typical front-end cache reduces load imbalance*, but might incur in coherency problems that is usually deal by caching objects at this layes for every small time windows.

2.4.2.3 Discussion

Dealing with load imbalance locally typically requires the use of replication. This allows to keep up with the data item access frequency by distributing the load across N servers (where N is the total number of replicas of that object). This by itself, reduces the congestion in a local receiver queue allowing for a more even response time for all data items. Therefore, replication enables the reduction of local access frequency while providing more availability, allowing the system to have more stable response time.

On the other hand, Two-Tier caching mechanisms also provides a good solution, since it releaves the load on the caching servers, while providing faster response times per data items with very high access frequency.

The reader should note that, both mechanism might complement each other where one adjusts local maximum access and the other amortizes the load of global maximum frequency data items

2.4.3 Data Size

Data size imbalance comes as a concern since it conditions several factors, such as (1) data transmission and (2) memory management.

In the context of (1) data transmission, high variances in data item size will cause variability in response times impacting the time it takes to process a request. In this scenarios, there are several approaches that aim at solving it, such as the use of RAID levels that allow parallel access for large data objects, usually used applied in the context of DBMS, and disk arrays, that allow a width range of different configurations to exploit different benefits. Other approaches go through simply splitting data items in several chunks and shard them across multiple servers, this can be seen in Erasure Code [27, 35]. Most of these solutions, besides decreasing data item size variance in each node, also allow higher throughput since they parallelize the fetch of a large data item across multiple small chunks, rebuilding the large data item in the client. On the other hand, the overall response time is conditioned by the slowest server across all servers steering chunks of the data object.

In (2) memory management there are several scenarios where data size can have an impact on the system performance, but we restrict the variants to a caching philosophy where there are no Swap-in/out operations, and ignoring L1, L2, L3 or higher internal cache level, considering only the overall hit/miss ratio over the RAM. Our main focus revolves around four features: (i)Memory Usage, (ii)Memory Fragmentation, (iii)Data Structure Overhead, and (iv)CPU Utilization.

In this context, memory management is resumed to Heap Allocation, and how to manage it. To store a data item a segment of memory blocks is required to accommodate it. To allocate a given amount of blocks, a system call is required, which consists executing kernel operations. This is commonly deemed as an expensive process. The same kind of process is required to free previously allocated memory. These operations in conjunction lead to efficient space memory management since internally, they avoid memory fragmentation. However, this space efficiency process takes a toll on the overall execution time.

Other approaches such as Slab Allocation moves memory management to the application level, where large heap memory segments are allocated and managed according to some application logic, that may result in memory under usage due to memory fragmentation, while benefiting the execution time, by avoiding the allocation and release of memory frequently. In these cases, when stoning data items high variation of sizes, one can incurin tends to increase memory fragmentation.

The analysis carried out in [28] shows that load imbalance increases with the amount of shards in relation to the coefficient of variation of the data item size, while chunking (spliting a data item in several parts) is a very practical and effective solution to reduce load imbalance, that allows to mitigate the effects on data item popularity by making per-chunk decisions.

2.5 Replication

Replication comes as a simple concept, where multiple instances of the same data item coexist in the same system. Typically, this is a strategy used to increase reliability and load balance, since several instances of the same data item allow it to be read simultaneously by several clients while remaining available even if some of the copies disappear. A common approach in replication relies on having the replicas at different storage devices for improved fault tolerance.

However, in the particular case of distributed caching systems, this approach always translates into an increase in memory overhead, since in most schemes replicating a data item N times requires N time more memory consumption. When thinking about caches in general, there is a common concept that is always present, the fact that the information in the cache is transient, and will only reside in the cache for a given (maximum) amount of time. Therefor, it becomes essential to decide the actual importance of fault-tolerance in a distributed cache.

Furthermore, under the assumption of failures not being frequent and that information is written in a persistent manner elsewhere, the overhead to ensure fault-tolerance, given in memory space and required coordination to maintain some degree of consistency among the replicas, might not be in our best interests.

As seen before, fault-tolerance might not be a feature needed in the cache, so there is no need for replicating all data items. Instead, a more selective replication strategy should be encouraged.

Following the reasoning above, holding more data items in the cache will lead to higher hit-rates, which aligns with the principle of no fault-tolerance in the cache. Nonetheless, the lack of replication will lead to an increase in response times when dealing with high access frequency data items. For this, replication seems to become relevant from the perspective of performance.

There are several replication schemes. We define a set of dimensions in which we categorise the replication mechanisms:

1. First Dimension: What and how the content is replicated.

   Active Replication all operations are executed by all processes.

   Passive Replication operations are executed by a single process, and the outputs are propagated to the other processes as an update.

2. Second Dimension: When does the replication occur.

   Synchronous Replication the replication happens in the critical path to the clients response.

   Asynchronous Replication a local update takes place, and the replication only starts after the reply is sent to the client.

3. Third Dimension: Which processes handle direct client operations.

   Single Master Only a replica process receives and executes operations that modify the state of the data.

   Multi-Master Any replicas can receive and execute any operation.

Any replication scheme can be seen as a composition of several of these dimensions.

2.5.1 Data Consistency

Replication becomes a non-trivial challenge when we allow more than just read interactions with the data. The write operations change the data items, which might lead to a divergence in the data item state, across the multiple replicas of that data item, and inconsistencies can be exposed to clients. An usual requirement is to maintain some consistency over the replicated data items.

Furthermore, there is a need to specify the restrictions on how data cen be percieved by the client. This specification of interactions follows some norms that are called data-centric consistency models [21, 22]. We describe the most relevant, where any relaxation of sequential consistency is typically reference as being a form of weak consistency:

Strict Consistency Assumes updates are propagated instantly to all processes, where any read operation returns the most recent write on a data item.

Sequential Consistency There is a total order of events for all processes (both reads and writes).

Causal Consistency Write operations that are potentially causally related must be seen by all processes in the same order. Concurrent writes might be seen in different order by different processes.

FIFO/PRAM Consistency All processes see all the write operations performed by each process in the same order that the process has executed them.

Cache/Coherent Consistency Similar to FIFO but with a granularity of single data items, where writes to different data item may be observed to occur in different order by different processes.

Eventual Consistency Updates are propagated on background,where there is no total ordering of operations, and conflicts must be dealt by some additional mechanism.

2.5.2 Replication Protocols

Caching systems usually demand a fast response time, which is accomplished in most systems by having a single instance of a data item. However, when the access frequency of requests for a data item exceeds the ability of a server to respond in bounded time, it

requires more instances to distributed the load across servers avoiding clogging a single server with requests. The replication in ideal conditions would reduce the access frequency to a data item by N (number of replicas) times. Still, this does not happen, since having multiple instances requires coordination and appropriate consistency guarantees. So there is a tradeoff between the benefits of replication and the coordination overhead to keep replicas consistent.

In the case of a Caching system, in the ideal scenario, a client only has to contact a single server to perform a read or write operation with minimal execution time. Since write operations require ordering, a simple solution that does not incur in high coordination overheads, forces all write operations to be submitted to the same server, respond to the client, and only start the replication process afterwards. This kind of solution is compliant with a Single-Master Passive Asynchronous replication approach.

Moreover, in this ideal scenario, a client should be able to perform a read operation over any replica, this would allow for a load balancing algorithm to take full advantage over the read requests, but this would relax consistency constraint to a form of weak consistency.

A Caching system needs to maintain some degree of consistency, in which it allows clients to make (read/write) requests to different replicas of the same logical data items and not obtain different results.

Sequential consistency is always a desired feature in most systems, yet protocols to enforce it do not scale well since it requires linearization, which demands a total ordering of operations. This level of consistency can be accomplished either by a high degree of coordination or through a single process deciding the global order. It typically incurs in problems regarding fast data access or lack of proper load balance in the accesses to the several instances of a data item.

In the following we briefly describe some replication protocols.

2.5.2.1 Primary/Backup Protocols

Primary/Backup Protocols [3, 32], are the common class of protocols for achieving passive replication. These assume a primary replica, which coordinates all writes on the data items that it is responsible for, depending on the variant of the protocol, it might allow for reads to be performed at any replica. When the primary fails, the protocol blocks until a new primary is elected.

2.5.2.2 Chain Replication

Chain Replication [26] is a variant of the Primary/Backup Protocol that aims to achieve high throughput and availability while maintaining per object linearizability.

Most replica management protocols in the presence of failures either block or sacrifice consistency to recover from failures. However, this protocol overcomes this problem by

organizing the nodes in the form of a chain where each node of the chain represents a replica. A chain is composed of three different types of replicas:

Head node responsible for handling all write requests;

Middle set of nodes between the head and the tail;

Tail node responsible for handling all read requests;

This scheme requires all write operations to be processed by the head, providing write ordering. Afterwards, write operations are propagated down the chain to the tail (primary), which ensures per object linerizability. However, only after the write propagation reaches and is processed by the tail, can the reply to the client be sent. Therefore, the full length of the chain becomes the critical path to the client's response time when processing a write operation.

Furthermore, the protocol also provides high robustness since it allows tolerance to N-1 concurrent faults, which is the worst scenario where only the tail remains available.

Despite this, even though this solution provides robustness against failures, it disregards load balancing, where the replication is only used to achieve fault-tolerance since the tail must process all requests (reads and writes).

A variant of this protocol, weak-chain replication, relaxes the consistency constraint, allowing read requests to probably read stale data by interacting with any server in the chain. Contrary to the expectations, it might underperform in the presence of more than 15% write operations when compared with the original chain replication protocol.

2.5.2.3 ChainReaction

ChainReaction [2] is a storage system that relies on an adaptation of chain replication, where the core concept remains the same, but relaxes the consistency requirements to provide load balancing.

In this variant, the chain splits into two segments, the first K elements and the remaining elements of the chain. The first K elements follow the philosophy of Chain Replication, which defines the critical path for the client's response time, in which the writes are submitted at the head of the chain, and the K^{th} node establishes a linearization for all operations.

The second part of the chain receives updates through lazy replication (lower priority relative to the other requests).

This mechanism provides load balancing by generating a token which represents the last position of the chain the client contacted while a write operation is in progress. This token restrains the range of nodes the client can interact with in following read operations to achieve causal consistency, ensuring that once a value is observed it is not possible to observe a previously written value. The causal consistency is guaranteed by the downstream propagation of the write operation.

Once the whole chain has completed the write operation, a back propagation of a notification will indicate the end of the write operation, allowing for clients in a subsequent access to discard the token and choose any replica to submit read operations.

2.5.3 Cache Coherence

Looking at a cache system from a more generic perspective, the cache is simply a secondary storage system for transient data that is used to speed up accesses, by avoiding requests to a primary persistent storage system.

Typically, the persistence storage defines and enforces rules over data, such as the ordering of operations, which works well when it is the only storage in the system.

However, considering two independent storage systems that share information over a set of independent intermediaries (application servers in most cases), maintaining the same write ordering becomes non-trivial.

Cases such as two clients making a write operation on the same data item, one after the other, might lead to incoherences, as changes in the primary durable storage do not imply that the secondary storage will apply such changes in the same order. Common causes are the lack of coordination between the intermediaries, latency in message transmission, and queue management.

There are several solutions to address this lack of coherence displayed between primary and secondary storage systems which typically fall in two groups [32]: (1) disallow shared data, or (2) allow shared data. The first class of solutions are based on caching only private data, resulting in an effective solution but with limited performance improvements; The solutions in the second group [9] have a more promising performance improvement but mainly resort to time-based expiration mechanism, explicit invalidation mechanisms, pre-fetching and piggy-back validation. All these solutions either increase the miss rate or generate more load in the database leading to additional delays in accessing data objects.

Another naive solution is to add to the queries a parameter to return the (*currenttime*) and use this information to coordinate the writes in the secondary storage. A time-based solution would only work if there was just a single instance of the database.

A more generic approach, would require databases to expose a sequence number of the transactions with each query reply. The sequence number would define the write ordering for the secondary storage.

Since databases typically do not provide or expose this information, there is a simple conclusion. It does not matter if the consistency inside the cache is strict if it is not coherent with the whole system (in particular the contents of the primary storage system).

2.5.4 Discussion

Chain Replication and ChainReaction, unlike the primary/backup protocol, does not block while performing a write operation, which allows higher throughput.

Analysis of Facebooks [23] deployment shows that cached data items tend to be a monotonically increasing snapshot of the database, where most applications can use a stale values.

Since for caching systems, it is acceptable to hold stale data for a bounded period, further relaxing this constraint to support causal consistency will provide more availability and better load distribution, which can be achieved using a solution inspired in the variant of Chain Replication used by ChainReaction.

Furthermore, in our use case when considering Chain Reaction as a replication protocol, with a K value set to 1, it allows the execution of local write operations, thus only initiating the replication after the response to the client is sent.

Since we are assuming no fault tolerance, if a server fails, all the writes in progress will have to be discarded.

2.6 Memcached

In our work we will use Memcached as a case study. Due to this we now discuss the current design of this system.

Memcached [1] is an open source, high-performance, distributed memory object caching system. The system is general purpose, due to its simplistic data sharding standalone server model. This system is composed by a client-server architecture where the clients store a HashTable with a key-server mappings managed by Consistent Hashing. In each server, only one copy of an entry is kept for the whole system. This solution has some obvious drawbacks (partially discussed before), such as limited Fault-Tolerance, Server Warm Up Phase after recovery from a failure, Hot-Spot due key distribution, and key access patterns cause by load imbalance.

Usually, Memcached can be found in large companies with data intensive workloads such as Facebook, which have read intensive access patterns, where data objects have significant variation in popularity. If not the case, this would pose as an obstacle for caching, since the whole application state would be equally viable for caching. But in fact, there are known patterns, in which data objects access frequency tend to follow a power-law leading to a Zipf Distribution, where few elements are accessed very frequently, a medium number of elements have a medium frequency, and many elements are accessed with a very low frequency. This leads to an important implication, higher frequency occurrences should be privilege in a caching. Also, mostly-read environments are usually the target for caching, allowing elements to stay in the cache with low probability of being written.

Since these features are a baseline on caching principles, intuitively there are some adjacent properties that results in the improvement of the cache hit-ratio:

- holding more popular data items.
- holding data items for a longer period of time.

Holding data for long periods of time will often lead to stale data since, if the cached data is changed in the primary storage component, these changes will not be reflected on the cache immediately. In this situations, there are common solutions like invalidation schemes, but this kind of mechanisms introduce some overhead in the system. More simple solutions are usually employed to avoid the overhead of invalidation, such as using a time-to-live (TTL) associated with each cached object. This limits the amount of time an object can remain in the cache, implicitly defining a bound in the time window where stale data can be served. To do this, each cached data item have a counter associated with it, that hold the creation timestamp based on the local server clock.

A Memcached instance is composed by a Cache Item Data Structure (CIDS)(fig.2.2)[36], which is an aggregation of data structures to manage the data items, containing several components: (1) HashTable, that is used to locate a data item in CIDS , (2) Eviction Mechanism, and (3) Slab Allocator.

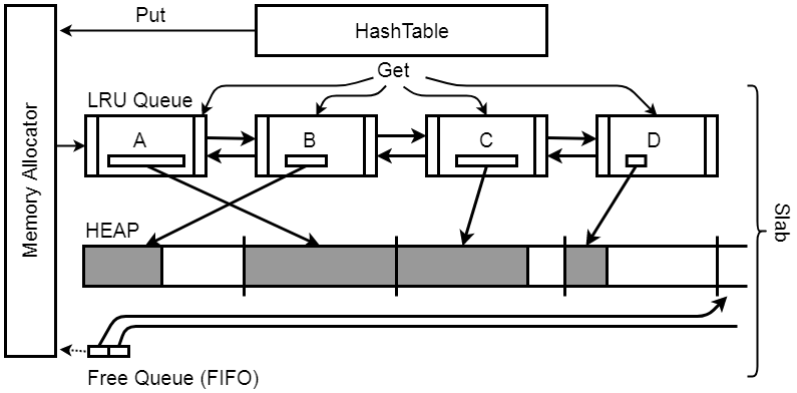


Figure 2.2: Cache Item Data Structure (CIDS)

2.6.1 Slab Allocation

Slab allocation is a mechanism based on a pool of slabs, where each slab is characterized by a continous segment of memory with a defined size and a known amount of memory chunks it can hold. Each chunk inside the same slab has the same size. This kind of memory organization tries to avoid memory fragmentation (trashing) and lowers the overhead in memory allocation since it avoids allocating, initializating, and freeing memory for data items individually, which otherwise would require the use of slow system calls [36].

There is a trade-off between fast placement of data items and memory usage, where this solution requires the data items to be placed in the slab that can better accommodate the data item according to the slab chunk size and the data item size. Since objects might be smaller than the chunk size, some space might not be used leading to under-usage of server memory. The chucks of memory in a slab, once allocated will not be freed, there is only a transition between states, where if a chunk is not being used for hosting a data item it is marked as free for allowing subsequent data placements. An in-depth measurement

study on performance in [4] evidences that Memcached can have close to 10% of memory under-usage.

2.6.2 Eviction Policy

In Memcaches, the eviction policy is limited to an individual slab, where each slab is independent from all others. Currently, Memcached allows two options regarding eviction: (1) LRU, and (2) a more recent eviction policy based on the 2Q Policy variant, more specifically the OpenBSD variant. The major difference resides in the 2Q policy shadow queue that is used to keep an history of previously cached objects in order to determine what was previously in the cache (and evicted meanwhile), which only stored keys. In the used variant, there is still a notion of three queues, but now they are named hot, cold, and warm, where all these queues store data items. New data items start in the hot queue.

The management is done according to the following strategies:

HOT queue places at the head of the queue new data items and data items that received a hit in the WARM queue, evicts from the tail directly to the head of the COLD queue

WARM queue places at the head of the queue data items that received a hit in the cold queue, evicts from the tails to the COLD queue

COLD queue places at the head of the queue data items evicted from HOT and WARM queues, data items at the tail will be evicted from the cache.

This eviction policy that follows along the line of 2Q policy, provides a safety for elements residing in the WARM queue against "scanning", where a wide range of data items are accessed only once, flushing out data objects accessed more frequently.

2.6.3 Concurrency Control

Memcached is naturally multi-threaded to process clients requests, where there is a set of available operations which can be split in two categories (1) reads and (2) writes. The write operation condenses all the operations from the API that involve the manipulation of the contents of cached data items.

In a typical execution flow, either of these operations requires a HashTable lookup in order to find the data item, additionally the data item is simply retrieved or updated, resulting in an update in the respective slab eviction mechanism being used. Both these steps require synchronization that is achieved through the use of locks, and as a penalty, the locks for both queues involved in the operation have to be held simultaneously since they are shared data structures with a strong coorrelation [36].

Furthermore, there is also a subset of internal operations such as: the crawler for checking TTL expiration and generation of statistics that also require locking. Some of these

mechanisms resort to coarse grain locking which can impact negatively the performance of Memcached.

2.7 Memcached Variants

2.7.1 R-Memcached

R-Memcached [15], that stands for Replicated Memcached takes the same approach as persistent Key-Value Stores, where each data item is replicated K times, and is assigned (and accessed) using Consistent Hashing. It exploits three replication strategies: (1) Two-Phase Commit, (2) Paxos, and (3) one that provides Weak-Consistency. Each of their replication strategies is used to address different requirements from applications operating on top of R-Memcached.

R-Memcached ensures fault tolerance through the use of replication. All employed replication schemes requires a minimum of three in order to tolerate at least one fault. This hard lower bound on replication induces a high memory consumption by a factor of three over the total data items. However, the replication factor is controlled by a parameterizable value of K, there for one can used additional replicas if required.

This schema provides a solution for access frequency hot spot until some extent delimited by K, where the reads might be load balanced between K servers for all data items. We note however that this solution presents an necessary overhead in memory consumption, by replicating all data objects.

Replicating only highly accessed data items could still benefit load balancing with a lower overhead.

2.7.2 Spore

Spore [10] follows a more reactive design, where it tries to leverage the frequency of data items with the latency requirements of an application. To do this, Spore defines a threshold in the receiving queue for each of the Memcached servers. This threshold is used to trigger (selective) replication for the most frequently accessed data items. It uses Reactive Internal Key Renaming (RIKR), that appends a suffix to the original key (i.e, identifier) of each data item. This suffix is defined as γ and it controls the replica count for each data item, not including the home server (i.e, the server responsible for that data item when the item is not replicated). This process requires the clients to maintain a client side cache that keep the γ values only for the replicated objects. The RIRK is used for defining key server mappings, combined with Consistent Hashing. To do this, Consistent Hashing is applied over the object identifier appended with a value between 0 and γ.

To select which data items should be replicated, when the receiving message queue threshold is reached, Spore captures the popularity of data items residing in that server. To avoid unnecessary processing overhead this is only performed considering the top

elements, since only the most popular must be replicated in order to provide minimal data transfer with significant impact on load distribution. Sampling provides a solution for this problem, but typically incurs in high space overhead, so in order to achieve a space efficient method to discover this data items, SPORE makes use of a very small cache ($\simeq$ 3% size of all that items stored in that node) that stores the access frequency for those items. The frequency of each data item is measured using Exponentially Weighted Moving Average (EWMA) [16] that computes the variance of an access based on the variance observed previously. This provides a way to allow popularity to decay with time, and adapt to changes in the workload.

The replica discovery is made simply by piggy-backing the γ to the client on a subsequent access to that data item during routine operations, where the servers keep track of the γs for each of the replicated data items.

When a client accesses (i.e, reads) an object that is replicated, it is notified (in the answer) of the current replication factor for that object (the γ value). This value is stored by the client for some time, during which it distributes read accesses among all γ servers. After some time, this value is discarded and the client simply reverts to contacting the home server when fetching objects. The writes are always directed to the home server in order to preserve write ordering.

SPORE provides two mechanisms for replication data objects with different caching consistency guarantees. The base mechanism provides weak consistency, and SPORE-SE (the second mechanism) uses Two-Phase Commit in order to ensure strong consistency.

2.7.3 CPHash

CPHash [17, 18] is based in an isolation conceptual design, in which the main goal is to create a hash table that works well on many core CPUs. In order to achieve it, memory is split into several independent parts (partitions). Key placement is performed among the partitions through the use of simple hash function. Each partition has a designated server thread that is responsible for all the interactions with that partition, where each server thread is pined to a core. This approach where each core works in isolation, due to the disposition hardware components in a server, results in higher throughput and a larger number of cache hits in L2 and L3 caches, especially when the HashTable meta-data fits the hardware caches of the CPU.

This approach when applied in a distributed context, becomes a two-tier indexing schemme to reach a Key-Value store partition, where a client uses Consistent Hashing to reach a server, and internally another layer of hashing is employed to locate the partition and the corresponding thread responsible for managing the target data object.

In each partition, there exists several buckets and chains of data items (similar to a HashTable) to locate a data item, a LRU queue is used to support an eviction policy, and memory management is done though allocation and freeing as data items as they enter and leave the cache.

2.7.4 MICA

Most caching systems resort to locking mechanisms to speed up data accesses by leveraging concurrency. This usually leads to heavyweight locks for concurrency control, and when possible fine-grain locks (stripped locks). However, even with fine-grain locking, the synchronization overhead is noticeable, affecting negatively the overall performance of a caching system.

MICA [14], was designed to overcome this performance bottleneck by removing the need of locking while also minimizing space consumption (that is implicit in concurrency control). It partitions the main memory per core (similar to CPHash), where each core becomes the owner of a partition and has a designated receiver buffer for access requests exclusively for the data items in that partition.

The general architecture of MICA is composed of three components:

Memory Partitions The main memory is striped in several partitions corresponding to the number of cores.

Parallel Data Access Each core becomes the owner of a partition with exclusive write access for data items in that partition.

Request Direction It relies in data affinity which distributes data items across partitions, and consequently cores.

MICA comes in the form of two variants that have different approaches on how to deal with concurrency: (1) MICA EREW, and (2) MICA CREW.

MICA EREW provides a non concurrent environment by avoiding internal data sharing, which implies that a set of data items is only accessible though a single core for read and write operations. This eliminates the overheads associated with concurrency control but leads to a potential bottleneck in data access, due to imbalance of the load across partitions (i.e, cores).

MICA CREW provides a concurrent environment where the execution of write is exclusive to the owner core, but allowing the reads to occur from any of the cores. Write-write conflicts are solved by having the owner core define the order of every write operation, leaving only the read-write operations as a potential source of conflict. This is addressed through versioning.

MICA CREW materializes itself as a solution for the bottleneck introduced in EREW, by relaxing its read constraints through the use of versioning. This mechanism introduces its toll on performance due to the use of retry mechanisms. This retry mechanism is based on a 32-bit version number that controls the interactions with a data item based on its state. These scheme can be explained by the behavior of the read and write accesses:

Read Access before and after reading a data item value, this version number must be read. If the version number read is odd or if different between the two read accesses, the operation must abort and retry.

Write Access before changing a data item value, the respective version number is incremented, signaling that a write is in progress (putting it as odd), the write is performed, and the version number is incremented again (becoming even again), which signals that the write has terminated.

Statistical analysis over MICA, allows for e better understanding of the applications of these two variants. CREW compared with EREW offers a small increase in throughput, when mostly read accesses are performed (95% read accesses). When dealing with a lower number of read accesses (50% read accesses), EREW outperforms CREW. This performance decay in CREW is a result of the retry mechanism associated with the versioning technique, while in EREW the read accesses show better performance due to the lack of contention and context switching implicit in the exclusive access architecture.

These solutions do not implement any kind of mechanisms to deal with real world deployments, since intrinsically there is no eviction policy mechanism that is adequate to deal with data item access frequency. This happens because memory management is based on single continuous memory chunk per partition, that works in a FIFO order only to avoid memory fragmentation. The most probable result for this memory management scheme is a significant drop in the hit ratio in some workloads (for instance, in the presence of scan operations).

2.7.5 Others Systems and Proposals

Intel developed a version of Memcached [36] exploiting concurrency control and increased parallelism. The main focus was achieving finer-grain locking for the underlying data structures while applying light weight locks (critical section). This is motivated by the fact that locks in the standard Memcached are in some cases coarse grain, which leads to high contention.

The locking mechanisms force a context switch for lock acquisition/release, this context switch can be classified in two distinct categories: (1) in-process thread switch, and an (2) kernel process switch. The first mechanism is much more efficient than the second, since, in the presence of standard mutex, it requires each use to switch the execution mode to kernel mode, which implies a high overhead in context switching. Using critical sections avoids the kernel interaction since it can only be used by the threads of a single process, not allowing it to be shared across processes [24].

Currently Memcached uses POSIX thread mutex. This is an implementation of the mutex that does not requires switching to kernel mode unless in the presence of high contention. This implies that Memcached is sensitive (performance-wise) to high contention.

Chapter 3

Proposed Solution and Work Plan

In this chapter we present the architecture for our proposed solution to improve distributed caching systems architectures. We further discuss our work plan and some metrics to be taken into consideration when evaluating our proposed solution.

3.1 Proposed Solution

Our proposed solution attempts to mitigate the sources of load imbalance in distributed caching systems, thus providing a more efficient scheme while trying to attain improved server response times within the latency requirement for an application.

For this, we plan to apply a scheme to identify high access frequency data items in the system and amortize its accesses. This can be accomplished by a Two-Tier cache, which avoids accesses to the caching servers through a local client side cache that stores the most accessed data items.

Also, we will employ local detection to try to avoid an increase in the response time of a server. If an anomaly is detected, the load can be rebalanced within the servers, or if the server is already overloaded, we plan on exploiting selective replication, similar to the solution presented in Spore, with a γ value, that represents the number of replicas for a given (popular) data object.

To achieve this, internally, in each server, the memory will be partitioned. These partitions are bound to a socket and assigned to a CPU Core, which has control over several partitions, each with exclusive read and write accesses, so there is no explicit concurrency in accessing cache memory.

3.1.1 Assumptions

There are two assumptions made with respect to deployment of the caching systems in order to improve its performance:

Load Balancer The ingress of all end-client traffic towards the application servers has been balanced by a Load Balancer.

One Hop Distance In the caching system all clients know all the caching servers.

3.1.2 Data Placement

We strive for a near optimal data placement algorithm with low overhead able to scale. Random Slicing fits this requirement, in which it has less the 1 % maximum variability from the normal distribution (for comparison, Consistent Hashing has a maximum variability of nearly 20%), requiring the use a PRNG to select a target server.

Near-optimal data placement provides close to the equal probability of a server owning a data item, which reflects the maximum usability of the server pool.

In the presence of K replicas, it requires an iterative process over the PRNG to find K different segments, there is no maximum bound on the number of iterations required to do this, yet practical results [20] show the amount of iteration is usually low.

Furthermore, instead of the conventional indexing scheme where the target is a server, here we target a composition of the server and local port, in which a server is represented by a set of socket addresses.

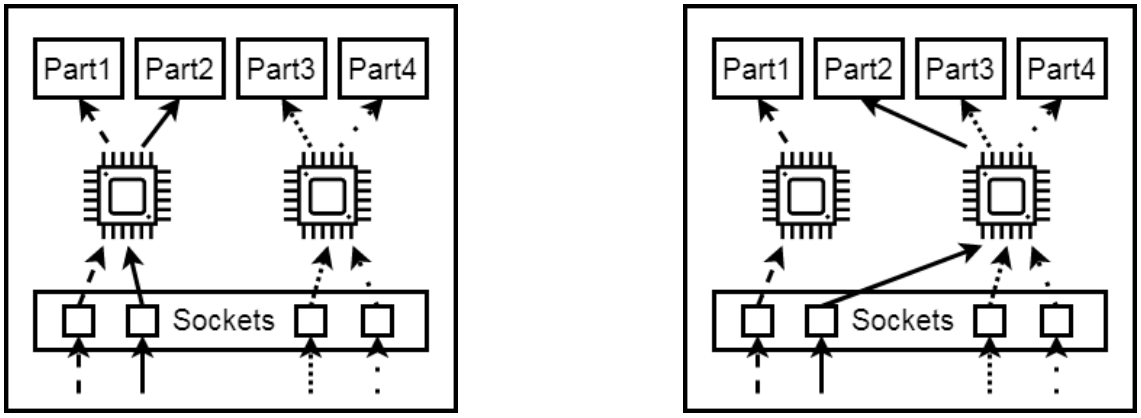
3.1.3 Server Architecture

Focusing on a single instance, inspired by MICA and CPHash, we split the memory into equal-sized partitions so that we can mitigate the local load imbalance through access frequency and data size. Each of these partitions can only be read or written in exclusive mode by a single CPU Core, this removes all the contention and locking overhead associated with concurrency. We do not bind a partition to a CPU Core since it would provide a sub-optimal solution, where a partition might hold some high access frequency data items and introduce delays when accessing other data items in the same partition (or other partitions managed by that CPU Core).

To minimize this problem, we split the memory by the number of sockets made available and bind it to them, which allows for a finer grain control while managing the accesses to the partitions. Each Core becomes responsible for a set of partitions and handles all the interactions with them.

3.1.4 Local Access Frequency Imbalance Management

As seen in Spore, a threshold that expresses the latency requirements for an application might guide the design of an adequate solution, where each of the receiver queues associated with each socket has a threshold mark. When the threshold is surpassed (Swamping state), it is probably due to the high access frequency of data items, which triggers a rebalancing mechanism, where the first approach tries to balance the node itself, by redistributing the sockets by the Cores, to relieve the bottleneck in the swamped queue (attributing additional CPU time for handling that queue).



a) Unbalanced b) Balanced

Figure 3.1: Correlation between sockets and partitions with exclusive access

If there is no viable rearrangement of sockets by cores, it will resort to replication, where it will select the data item with highest access frequency, which should translate to the most load relief (by a factor of N relative to the access frequency of that data item) with minimal data transfer overhead.

3.1.5 Replication

We will resort to a variant of ChainReaction, which has K value of 1, allowing write operations to be performed on the "home"node of a data item, and respond to a client before starting the replication, resuming the critical path for the client's response into a single node.

This solution does not introduce fault tolerance, as when a server fails, all the data items currently being written have to be discarded in all the replicas.

However, it provides the possibility to load balance accesses across replicas for very popular objects, where each client must keep a token of ongoing write operations in replicated data items. This solution naturally adapts to the existence of a single instance of a data item.

3.1.6 Global Access Frequency Imbalance Management

To prevent additional on caching servers, typically introduced by very high access frequency data items, a Two-Tier caching solution is required. In this scenario, most of the very high access frequency data items will appear to each of the clients as high access frequency data items in comparison to all the other data objects.

The Two-Tier caching solution, calls for a detection mechanism in order to identify the most accessed data items per client, and store them in a local cache. This cache will demand low TTL to avoid incoherence.

The updates to the local cache should not wait for a response since it is irrelevant if some of the updates fail since we are only interested in the data items that show a high access frequency compared with others in a local context.

3.1.7 In-Memory Management

Slab memory provides a great benefit regarding performance, since it does not require constant allocation and freeing of memory, yet this comes at the cost of memory underusage. To avoid that, since we are managing RAM and the memory is split into several partitions, a solution for a better memory usage which takes the building blocks of slab memory might become viable. Taking the concept of slab, which reflects a large segment of allocated memory with a defined size that is split into chunks of equal size, and mapping it to a partition, re-introduces the problem of different data item sizes, usually solved by the use of slabs. We intend to experiment a solution that splits all data items into equal-sized chunks, allowing for lower memory waste.

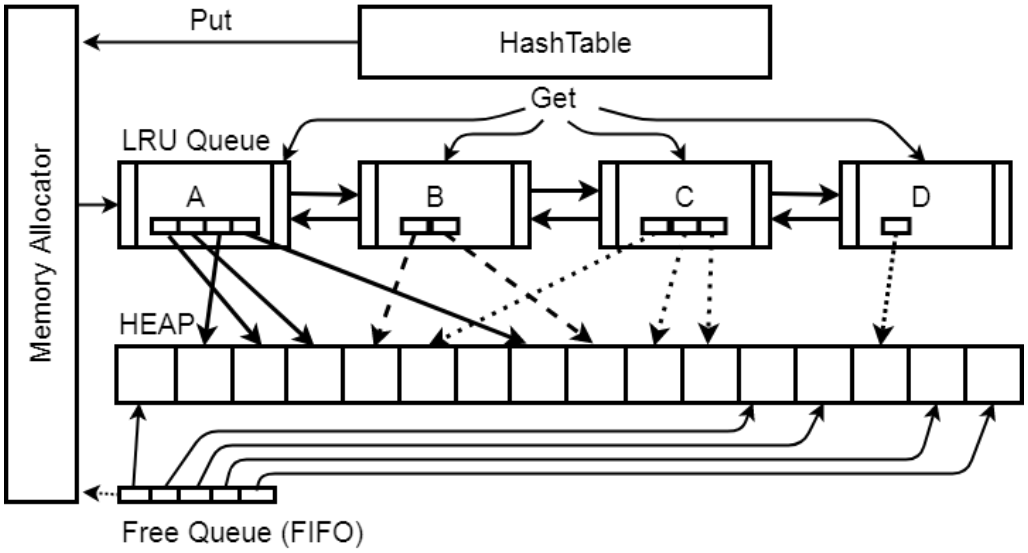


Figure 3.2: CIDS with data chunking

This approach takes the free list as a FIFO queue of chunks, each time a data item is placed in the cache; it will fetch the number of chunks needed to accommodate that

data item, these chunks might be scattered across the heap, not necessarily contiguous segments which is compliant with the definition of RAM. Once an object is removed from the cache, it will simply reference those chunks as free, placing them at the end of the FIFO queue.

These are data partitioning scheme is just a step further of the already existing scheme for slab memory management from Memcached, where instead of fetching a chunk with the size necessary to accommodate the data item, now we fetch a set of chunks, using the same underlying data structures, with more fragmentation.

3.1.8 Membership Management

As previously discussed, fault-tolerance does not come as a requirement for caching, so upon failures, we simply take the loss. The recovery and redistribution of the lost items due to the failure will be dealt with by future client requests.

However, we strive to minimize this loss through selective replication. We are able to keep the high access frequency data items over which there was no write operation being executed at the time of the failure.

When a server joins the pool of servers, there is a need for several keys to be re-mapped from existing servers to the new server. This can be achieved through a delayed entry, where a threshold will mark the number of data items the server should hold to avoid increasing the miss-rate. The threshold can be easily computed as a fraction of the amount of data items it should hold assuming a close to uniform distribution with some small deviation.

This delayed entry should be coordinated by the clients as requests arrive, to avoid key recomputation in the servers, that would introduce more latency in the response and increase the size of the receiver queues.

If a new server is added, it should result in the new view of the membership. The client during the warm-up phase of a server will need to hold the two membership views simultaneously, the previous membership view to obtain a response, and the new membership view to notify the server to do forward propagation of the data item to the new server.

In this scenario, the request sent by the client has two main sections, the actual request for the server that holds the item, and the notification for forward propagation, that the server will interpret as an asynchronous push of the data item for the server in the warm-up phase without any retry mechanisms.

When the server in the warm-up phase reaches its threshold, it will notify all clients, so that they can discard the previous view, resuming normal behaviour. Since the clients maintain the same γ values of the previous configuration, this will result in a natural eviction process of all the data items the new server now holds in the servers that were previously responsible for them.

3.2 Evaluation

We will perform incremental and isolated evaluations of all our solutions exploiting the vectors of imbalance, to define what gain and overheads are attained in each iteration to understand the tradeoffs of all the components better and how do they scale and adapt to different workloads. The workloads will vary between 95%, 70%, and 50% read operations, and all measurements will be compared against standard Memcached and with a static and dynamic environment, where item popularity changes over time.

Benchmarks will evaluate (1)Hit-Ratio as a measurement of the percentage of request that was able to fetch a data item from the cache and (2) Miss-Ratio as a complementary metric. Another key aspect to be considered during evaluation is the latency perceived by clients. Furthermore, each of the elements tested will consider some additional measurements:

- Data Placement: Change the data placement algorithm from Standard Memcached.

  Variability: measurement of the percentage of misplacement from the uniform distribution.

- Partition scheme: Change Memcached from shared memory to exclusive access with local detection. The evaluation will be based on an incremental combination of these mechanism:

  1. Static amount of partitions a core owns.
  2. Load balancing the partitions per core.
  3. Introduce replication when unable to maintain the latency requirements.

  Assuming these metrics:

  Amount of Bursts: Number of times the threshold in a queue was surpassed.

  Mean Time To Rebalance: Amount of time until the size of the queue becomes lower than the threshold.

  Amount of Socket Shifts: Number of times the sockets (and associated partitions) had to redistributed among CPU Cores.

  Amount of Replicas: Amount of data items replicated.

  Additional Memory Usage: Amount of additional memory consumption due to replication.

  Maximum Queue Size: Maximum number of packets in a queue.

  Throughput: Amount of responses the a server is able to provide per second.

- In-Memory Management: Change the CIDS from Standard Memcached

  Memory Usage: Amount of wasted allocated memory.

Throughput: Amount of responses the a server is able to provide per second.

- Two-Tier Cache: add a local cache for extremely popular data items in the client of Memcached.

  Queue Length Average length of the cache server queues.

An initial approach to conduct evaluation is to leverage the Yahoo Cloud Service Benchmark (YCSB) to exercise a system using controlled artificial workloads.

3.3 Work Plan

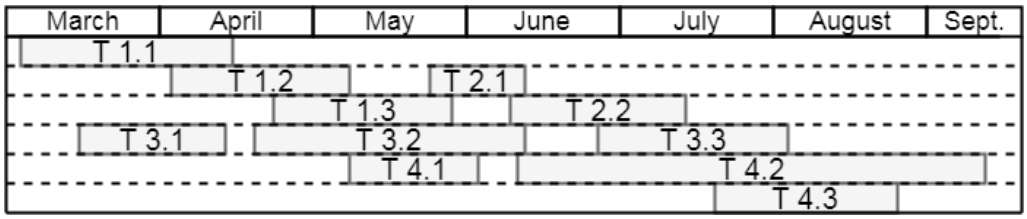


Figure 3.3: Work Plan

Task 1: Design and development, composed by the following subtasks: (1.1) Memory management; (1.2) Partition allocation; and (1.3) Replication.

Task 2: Optimizations, composed by the following subtasks: (2.1) Code review; and (2.2) Remaining bottlenecks.

Task 3: Experimental evaluation, composed by the following subtasks: (3.1) Setup baseline system (Memcached, MICA); (3.2) First phase of evaluation (before optimizations); and (3.3) Final experimental evaluation.

Task 4: Writing, composed by the following subtasks: (4.1) Paper for Portuguese Conference; (4.2) Thesis Writing; and (4.3) Paper for International Conference.

Figure 3.3 presents a schematic representation of the work plan. We plan on writing two papers reporting the results achieved in the thesis (one for the national conference and another international venue).

While the plan presented here is ambitious, we will tackle its execution in phases, which will allow to have valid results even if one task is delayed.

Bibliography

[1] B. Aker. *Memcached - a Distributed Memory Object Caching System*. 2015. URL: http://memcached.org/ (visited on 01/31/2018).

[2] S. Almeida, J. a. Leitão, and L. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 85–98. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465361. URL: http://doi.acm.org/10.1145/2465351.2465361.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. "Distributed Systems (2Nd Ed.)" In: ed. by S. Mullender. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. Chap. The Primary-backup Approach, pp. 199–216. ISBN: 0-201-62427-3. URL: http://dl.acm.org/citation.cfm?id=302430.302438.

[4] W. Cao, S. Sahin, L. Liu, and X. Bao. "Evaluation and analysis of in-memory key-value systems." In: *Proceedings - 2016 IEEE International Congress on Big Data, BigData Congress 2016*. 2016, pp. 26–33. ISBN: 9781509026227. DOI: 10.1109/BigDataCongress.2016.13.

[5] A. Chawla, B. Reed, K. Juhnke, and G. Syed. "Semantics of caching with spoca: a stateless, proportional, optimally-consistent addressing algorithm." In: *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association. 2011, pp. 33–33.

[6] 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: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[7] 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: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[8] "Design Considerations for Distributed Caching on the Internet." In: *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. ICDCS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 273–. URL: http://dl.acm.org/citation.cfm?id=876891.880589.

[9] A. Dingle and T. Pártl. "Web cache coherence." In: *Computer Networks and ISDN Systems* 28.7 (1996). Proceedings of the Fifth International World Wide Web Conference 6-10 May 1996, pp. 907 –920. ISSN: 0169-7552. DOI: https://doi.org/10.1016/0169-7552(96)00020-7. URL: http://www.sciencedirect.com/science/article/pii/0169755296000207.

[10] Y.-J. Hong and M. Thottethodi. "Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier." In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 13:1–13:17. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2525970. URL: http://doi.acm.org/10.1145/2523616.2525970.

[11] K. Ishikawa. "ASURA: Scalable and Uniform Data Distribution Algorithm for Storage Clusters." In: *CoRR* abs/1309.7720 (2013). arXiv: 1309.7720. URL: http://arxiv.org/abs/1309.7720.

[12] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450. ISBN: 1-55860-153-8. URL: http://dl.acm.org/citation.cfm?id=645920.672996.

[13] 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: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[14] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. "MICA: A Holistic Approach to Fast In-memory Key-value Storage." In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 429–444. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616488.

[15] Y. Lu, H. Sun, X. Wang, and X. Liu. "R-Memcached: A Consistent Cache Replication Scheme with Memcached." In: *Proceedings of the Posters & Demos Session*. Middleware Posters and Demos '14. Bordeaux, France: ACM, 2014, pp. 29–30. ISBN: 978-1-4503-3220-0. DOI: 10.1145/2678508.2678523. URL: http://doi.acm.org/10.1145/2678508.2678523.

[16] J. M. Lucas and M. S. Saccucci. "Exponentially weighted moving average control schemes: Properties and enhancements." In: *Technometrics* 32.1 (1990), pp. 1–12. ISSN: 15372723. DOI: 10.1080/00401706.1990.10484583. URL: http://www.tandfonline.com/doi/abs/10.1080/00401706.1990.10484583.

[17] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. "CPHASH: A Cache-partitioned Hash Table." In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 319–320. ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145874. URL: http://doi.acm.org/10.1145/2145816.2145874.

[18] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. "CPHASH: A Cache-partitioned Hash Table." In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 319–320. ISSN: 0362-1340. DOI: 10.1145/2370036.2145874. URL: http://doi.acm.org/10.1145/2370036.2145874.

[19] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes. "Reliable and Randomized Data Distribution Strategies for Large Scale Storage Systems." In: *Proceedings of the 2011 18th International Conference on High Performance Computing*. HIPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10. ISBN: 978-1-4577-1951-6. DOI: 10.1109/HiPC.2011.6152745. URL: http://dx.doi.org/10.1109/HiPC.2011.6152745.

[20] A. Miranda, S. Effert, Y. Kang, E. L. Miller, I. Popov, A. Brinkmann, T. Friedetzky, and T. Cortes. "Random Slicing: Efficient and Scalable Data Placement for Large-Scale Storage Systems." In: *Trans. Storage* 10.3 (Aug. 2014), 9:1–9:35. ISSN: 1553-3077. DOI: 10.1145/2632230. URL: http://doi.acm.org/10.1145/2632230.

[21] L. P. Miret. "Consistency models in modern distributed systems. An approach to Eventual Consistency." Doctoral dissertation. Technical University of Valencia, 2014. URL: https://riunet.upv.es/bitstream/handle/10251/54786/TFMLeticiaPascual.pdf?sequence=1.

[22] D. Mosberger. "Memory Consistency Models." In: *SIGOPS Oper. Syst. Rev.* 27.1 (Jan. 1993), pp. 18–26. ISSN: 0163-5980. DOI: 10.1145/160551.160553. URL: http://doi.acm.org/10.1145/160551.160553.

[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: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 385–398. URL: http://dl.acm.org/citation.cfm?id=2482626.2482663.

[24] J. Preshing. *Locks Aren't Slow; Lock Contention Is*. 2011. URL: http://preshing.com/20111118/locks-arent-slow-lock-contention-is/ (visited on 01/31/2018).

[25] *Redis as an LRU cache [1]*. URL: https://redis.io/topics/lru-cache (visited on 02/06/2018).

[26] R. van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 7–7. URL: http://dl.acm.org/citation.cfm?id=1251254.1251261.

[27] L. Rizzo. "Effective Erasure Codes for Reliable Computer Communication Protocols." In: *SIGCOMM Comput. Commun. Rev.* 27.2 (Apr. 1997), pp. 24–36. ISSN: 0146-4833. DOI: 10.1145/263876.263881. URL: http://doi.acm.org/10.1145/263876.263881.

[28] L. Saino, I. Psaras, and G. Pavlou. "Understanding sharded caching systems." In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524442.

[29] E. Schurman and J Brutlag. *Performance related changes and their user impact*. velocity web performance and operations conference, 2009. 2009.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: http://doi.acm.org/10.1145/964723.383071.

[32] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[33] D. G. Thaler and C. V. Ravishankar. "Using Name-based Mappings to Increase Hit Rates." In: *IEEE/ACM Trans. Netw.* 6.1 (Feb. 1998), pp. 1–14. ISSN: 1063-6692. DOI: 10.1109/90.663936. URL: http://dx.doi.org/10.1109/90.663936.

[34] *Using Key-Value Store as a Cache*. URL: https://www.aerospike.com/using-key-value-store-as-a-cache/ (visited on 02/06/2018).

[35] H. Weatherspoon and J. Kubiatowicz. "Erasure Coding Vs. Replication: A Quantitative Comparison." In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 328–338. ISBN: 3-540-44179-4. URL: http://dl.acm.org/citation.cfm?id=646334.687814.

[36] A. Wiggins and J. Langston. "Enhancing the scalability of memcached." In: *Intel document, unpublished* (2012). URL: https://software.intel.com/sites/default/files/m/0/b/6/1/d/45675-memcached{_}05172012.pdf.

[37] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. "Characterizing Facebook's Memcached Workload." In: *IEEE Internet Computing* 18.2 (2014), pp. 41–49. ISSN: 1089-7801. DOI: 10.1109/MIC.2013.80. URL: doi.ieeecomputersociety.org/10.1109/MIC.2013.80.

[38] N. Zaidenberg, L. Gavish, and Y. Meir. "New Caching Algorithms Performance Evaluation." In: *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. Spects '15. Chicago, Illinois: Society for Computer Simulation International, 2015, pp. 1–7. ISBN: 978-1-5108-1060-0. URL: http://dl.acm.org/citation.cfm?id=2874988.2875009.

[39] J. Zhou, W. Xie, J. Noble, K. Echo, and Y. Chen. "SUORA: A Scalable and Uniform Data Distribution Algorithm for Heterogeneous Storage Systems." In: *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. Vol. 00. 2016, pp. 1–10. DOI: 10.1109/NAS.2016.7549423. URL: doi.ieeecomputersociety.org/10.1109/NAS.2016.7549423.