Miguel da Silva de Brito Cordeiro

Degree in Computer Science

Rethinking Distributed Caching Systems Design and Implementation

Dissertation 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

Examination Committee

Chairperson: Hervé Miguel Cordeiro Paulino
Raporteur: João Coelho Garcia
Member: João Carlos Antunes Leitão

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2018

Rethinking Distributed Caching Systems Design and Implementation

Copyright © Miguel da Silva de Brito Cordeiro, Faculty of Sciences and Technology, NOVA University of Lisbon.
The Faculty of Sciences and Technology and the NOVA University of Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Paciência, café e perseverança são imprescindiveis para superar qualquer obstáculo.

Acknowledgements

The realization of this dissertation was only possible thanks to the contribution, in many ways, from various institutions and persons to which I would like to dedicate a few words of gratitude.

My thanks to Professor João Leitão and Professor Doctor Vítor Duarte, my dissertation advisor and co-advisor for all the support and patience they had with me.

To all of my friends and colleagues who have always been there to motivate me, help me and have share with me this entire college experience.

Finally, to my parents, sister, and all my family who, with great care, support, and headaches, didn't measure any efforts for me to reach this step in my life.

This work was partially supported by the European Research Project LightKone under H2020 grant agreement ID 732505, and FCT through Project NG-STORAGE under contract PTDC/CCI-INF/32038/2017. The experiments presented in this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

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 execution environments and workloads, under which such systems typically operate, several problems arise in the form of load imbalance.

This thesis addresses the sources of load imbalance in caching systems, mainly: i) data placement, which relates to distribution of data items across servers and ii) data item access frequency, which describes amount of requests each server has to process, and how each server is able to cope with it. Thus, providing 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. Our solution revolves around increasing parallelism through load segregation, and solutions to overcome the load discrepancies when reaching high saturation scenarios, mostly through access re-arrangement, and internal replication.

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

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, principalmente: i) colocação de dados, que se relaciona com a distribuição dos dados pelos servidores e a ii) frequência de acesso aos dados, que reflete a quantidade de pedidos que cada servidor deve processar e como cada servidor lida com a sua carga. Desta forma, 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. A nossa solução gira em torno de aumento de paralelismo atraves de segregação de carga e em como superar superar as discrepâncias de carga a quando de sistema entra em grande saturação, principalmente atraves de reorganização de acesso e de replicação interna.

Palavras-chave: Sistemas de caching distribuídos, Memcached, Replicação seletiva, Colocação de dados, Alta disponibilidade, Paralelismo, Equilíbrio de carga, Gestão de memória

Contents

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

2 Related Work **5**

2.1 Preliminaries 5
2.2 Cache Overview 6
2.3 RAM Key-Value Stores 7
2.3.1 Single Cache Instance 7
2.3.2 Distributed Caching Systems 9
2.4 Load Imbalance 10
2.4.1 Data Placement 11
2.4.2 Data Access Frequency 15
2.4.3 Data Object Size 17
2.5 Data Replication 18
2.5.1 Data Consistency 19
2.5.2 Replication Protocols 20
2.5.3 Discussion 23
2.6 Cache Interaction Models 23
2.6.1 Cache Coherence 24
2.6.2 Handling Stale Data 25
2.7 Memcached 25
2.7.1 Slab Allocation 26
2.7.2 Eviction Policy 26
2.7.3 Concurrency Control 28
2.8 Memcached Variants 28
2.8.1 R-Memcached 28
2.8.2 Spore 29
2.8.3 CPHash 30
2.8.4 MICA 30
2.8.5 Other Systems and Proposals 32

2.9 Summary 32

3 Conceptual Design **33**

3.1 Cache Systems 33

3.1.1 Cache Server Model 34

3.1.2 Data Item Availability 35

3.2 Partitioned Server 35

3.2.1 In-Memory Management 36

3.2.2 Partition Access 38

3.3 Local Access Frequency Imbalance Management 40

3.3.1 Decision Making 41

3.3.2 Socket Shift 42

3.3.3 Selective Replication 43

3.3.4 Indexing Replicas 45

3.3.5 Summary 46

3.4 Client Architecture 47

3.4.1 Global Access Frequency Imbalance Management 48

3.4.2 Random Slicing Implementation 48

3.5 Summary 50

4 Implementation Details **51**

4.1 Internal Network Model 51

4.1.1 Exclusive Partition Access 53

4.2 Memory Partitioning 54

4.2.1 Slab Memory 54

4.2.2 Item Access 55

4.3 Load Monitor 57

4.3.1 Internal Full Mesh Network 60

4.4 Socket Shift 61

4.5 Selective Internal Replication 64

4.5.1 Burst Control 65

4.5.2 Shrink 66

4.5.3 In Memory Replication Protocol 66

4.6 Summary 72

5 Evaluation **75**

5.1 Experimental Setup 76

5.2 Data Placement 77

5.3 Data Item Popularity 79

5.4 Partition Scheme 80

5.4.1 Skewed Workloads 82

5.5 Open File Descriptors and Polling 85

5.6 Socket Shift 86
5.6.1 Exponential Moving Average (EMA) 87
5.6.2 MakeSpan 88
5.7 Selective Internal Replication 90
5.8 Summary 93

6 Conclusion and Future Work **95**

Bibliography **97**

A Impact of File Descriptors **103**

I EMA CPU load **105**

II System Limits **109**

List of Figures

2.1 In line Random placement strategies 13
2.2 Cache Item Data Structure (CIDS) 26

3.1 Cached pool interaction 34
3.2 Cache server layout 35
3.3 Two-Tier Indexing Schemes 39
3.4 Correlation between sockets and partitions with exclusive access 43
3.5 Correlation between sockets and partitions with exclusive access 46
3.6 Random Slicing Structure 48

4.1 Network model 52
4.2 Accept New Connections 53
4.3 Memcached Memory Structure 54
4.4 Logical view of the socket shift control plane 62
4.5 Socket Shift Frame 63
4.6 Segmented LRU 64
4.7 Get/Set operations 67
4.8 Replication Frame 67
4.9 Chain Loop 68
4.10 Deadlock scenarios at data item level 70

5.1 Section of Rennes network topology 76
5.2 Evaluation of data misplacement on a finite set of servers 78
5.3 Zipfian generator for 2M keys with 20M accesses with varying α 79
5.4 Comparison of Memcached vs its Partitioned version under a uniform data distribution 81
5.5 0.08% maximum variability in data accesses for 4 server threads with 95% read operations 82
5.6 Comparison of Memcached vs its Partitioned version under a Zipfian distribution with $\alpha = 0.99$. 83
5.7 Comparison of Memcached vs its Partitioned version varying Zipfian distribution with 95% read and 5% write operations. 84

5.8 Increase the amount of partitions with 4 server threads and similar increase of file descriptors from Libevent. 85
5.9 Comparison of Memcached vs its Partitioned version with Socket Shift. . . . 86
5.10 30 second trace of to eight partitions with 95% read operations. 88
5.11 Throughput and Load Distribution of the Partitioned version with socket Shift 89
5.12 Base Line comparison with Socket Shift 91
5.13 average SIR with Min-Max markers 92
5.14 SIR comparison based on Throughput, Latency and Miss ratio 92

A.1 Comparison between Memcached and Socket Shift with 8 Sever Threads . . 103

List of Tables

2.1 Comparison between key distribution algorithms 14

5.1 Zipfian generator for 2M keys with 20M accesses with varying α 80

5.2 Relative throughput Gain for N static partitions and 4 server threads 83

5.3 Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations 89

5.4 Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations 93

A.1 Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations 104

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 web evolves, so do its challenges. A massive use of web services and applications lead to an increasing number of client requests in which the user-perceived latency is determinant, for instance, in cases such as Facebook, Youtube, Flickr, Twitter, or Wikipedia. Ensuring low response times requires fast data retrieval from web services. In turn this motivates the need for highly efficient designs of mechanisms that can speed up data access times when compared to simply fetching data from a (potentially slow) database or set of databases.

A study performed by Microsoft and Google [34] demonstrates that response times matter, revealing that server delays have a substantial negative impact on the continuous usage of web applications. In particular, delays under half a second affect business metrics, with a direct translation on generated revenue. Moreover, they have shown 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, in addition to web servers, might also contain application servers. This component 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. This enables client operations to be handled by any 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 third component: storage.

The Storage component might be composed by one or several storage systems, including SQL or NoSQL databases, or any other persistence storage, such as a distributed file system. Typically, a web application server is used to mediate the access to one or multiple storage services when handling different client operations. Upon getting a reply from the storage services, the web application server computes (and composes) an answer for the client and sends it.

When designing a web application, the storage is a crucial component of the application. This design has to be addressed carefully, due to its significant impact on the overall application performance. 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 different types of application data.

In these cases, the response time is dependent on the slowest storage system. Still, there are several factors that weight in the response time of a web application, such as the computational cost of executing a given query, the size of the receiver queues in the database, and the network communication time. In particular, relational databases tend to be 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, which can wreak 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 the reuse the previously computed replies, and hence lower the user-perceived latency of such requests. In general, this entails the use of caching systems.

A caching system is a secondary storage component, typically materialized by an in-memory data store, that provides fast access to data items or previously computed replies, that either exist or depend on data that is (persistently) stored in one or more of the databases used by the application. The web application servers typically exploit this Caching component to avoid duplicate work and speed-up data access times. Caching systems usually follows a principle of non-interference, where they do not interact with other storage systems directly. This design choice provides simplicity to their design and avoid increasing the complexity of the main 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 requires establishing some form of 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), that attempt to ensure that data items that are more frequently accessed remain more time in the cache.

When clients make a request, the web application servers will first try to access the cache, only resorting to the slower data storage component when the data is not currently cached. 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 more often.

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. However, this also brings 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, whose capacity may become exhausted.

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

There are several examples of caching systems in the literature that are deployed and used by many applications. Each of these systems has their own properties, such as: MICA [18] that achieves extremely high throughput; Aerospike [41] that is optimized for flash-memory, unlike most alternatives, allowing data persistence by design; Redis [30] which resorts to dynamic memory allocation, and Memcached [2] that employs static memory allocation. We discuss these different systems in more detail in the next chapter.

Problem Statement

The response time in a web application is a crucial aspect that directly impacts the user experience, as such we take fast response times as primary concern, which can be mitigated through a better cache usage regarding memory management and access distribution.

Most web based environments tend to display a Zipfian distribution as its access pattern, where the increase in accesses is strongly coorelated with data item popularity. This distribution can become highly disruptive, becoming the cause of several bottlenecks.

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 production environments throughout the world.

Approach

Our solution strives to increase the overall throughput by exploiting parallelism while resorting to a memory access partitioning scheme. However, this kind of approach imposes limitations in high saturation scenarios with synchronous clients, where the highest latency experienced by a partition limits the overall response time. To address this, we developed two algorithms that mitigate such limitations with different goals: (1) redistributing the load, and (2) internal replication for a more fine grain load relief.

Document Structure

The remainder of this document is structured as follows:

- Chapter 2 - Introduces the related work, focusing on the standard Memcached, variants, eviction policies, and key sources of load imbalance in caching systems.
- Chapter 3 - Presents the conceptual view of our solution and discusses the desired cache properties to mitigate the sources of load imbalance in caching systems, striving for an overall better performance.
- Chapter 4 - Discusses the relevant technical aspects, algorithms, and protocols applied in our solution.
- Chapter 5 - Provides an experimental evaluation of our solution, following an analysis of access patterns, data placement algorithms, considering the individual components of our solution and their combination.
- Chapter 6 - Concludes the document with final remarks and directions for future work.

Chapter 2

Related Work

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

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, that serves as a departure point for the remainder of the presentation in this chapter.

A Pseudo Random Number Generator (PRNG) is a class of deterministic functions, that take as input a seed and generates sequences of numbers, trying to approximate a true random number generator. Most PRNGs guarantee a certain level of randomness within a sequence of generated numbers. Although with low guarantees regarding the correlation between sequences generated using different seeds, which can lead to more predictable distributions and less fairness [24] (randomness).

A cryptographic hash function takes an arbitrary sized input and generates a fixed size output with the following 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 required number of CPU cycles.

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

Collision Resistance: It should be difficult for two different inputs $m1$ and $m2$ to have $hash(m1) = hash(m2)$, in more detail, knowing $hash(m1)$ and $m1$, it should be hard to find an arbitrary $m2$, such that $hash(m1) = hash(m2)$.

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 persistent (and slower) data store, or they can be the result of performing computations over multiple inputs stored in one or multiple persistent 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 and hard disks. As caches have limited space (RAM is limited), in general, not all data items can be maintained in the cache systems at all times. Hence, when cache space is exhausted, eviction of existing data objects trough an eviction policy, is essential 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 indicator for caching systems is the hit-ratio, where higher values imply better performance, which usually directly translates to lower response times for the application using the cache system.

2.3 RAM Key-Value Stores

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

Proxy caches are mainly accessed by clients and employ a philosophy of caching data closest to the clients. In order to achieve it, a common approach is to rely on an hierarchical caching architecture, where the root cache is the only one that interacts with a persistent storage system [9], and all other cache levels contents are also present in caching layers above it in the cache hierarchy.

Service caches are the caching systems in which the work presented in this thesis focus on. 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 that serve to manage that object within the cache system.

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, 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 user requests, particularly damaging if the system exhibits a high miss ratio, as most interactions with the cache system will simply be delaying the answers for client 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 key challenge in this aspect resides on the automatic selection of which data items to evict. For that, there are several algorithms [46] that can be used, and have demonstrated good performance, at least for specific 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 completeness, we now briefly describe some of the more popular eviction policies:

FIFO Evicts the oldest data item residing in the cache system, without considering either recency or frequency. The clear benefit of this policy is that it is simple and can be implemented without tracking accesses to individual data objects.

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 eviction rule to quickly remove cold data items from the cache that become unpopular (i.e, less frequently accessed).

2Q policy [15] 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 support data structure, 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 Proposes an interesting and different approach. 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 the 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 among 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 independent decisions.

2.3.2 Distributed Caching Systems

A distributed cache is typically deployed in a cluster of nodes, offering a logical view of the cache as if it was materialized by a 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 [17]. This allows for load distribution across nodes, where each access targets the node that's responsible for the requested data item. Also, it allows a simple way of scaling-out when in need to accommodate more data items, by simply adding more nodes to the system. This is a great advantage since scaling-up is not cost-effective, where the cost of adding more resources is not linear with the improvements provided by it [24].

When in the presence of a node failure, it does not result in complete outage of the cache, as the system will still continue to operate, taking only a small overhead, as another node will have to take over the responsibilities of the failed node.

In the following, we study the sources of imbalance on the system, that force some nodes to struggle with the load they are subjected to, resulting in an overall 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 suitable solution, since they can easily scale out, allowing them to better adapt to the applications needs [45].

Some core features that are desirable in caching systems are high hit-ratio and fast response times with low variance. Typically, applications are subject to accesses that can be modelled 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 degradation of 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.

This can be described by the popular *Balls into Bins* problem [29], where one considers a finite set of m balls, being thrown into a finite set of n bins, each throw is independent, and if the throws follow an uniform distribution, it implies that the probability of ball falling into a bin is $\frac{1}{n}$. It is well know that the maximum load is approximately $\frac{logn}{loglogn}$, with a high probability.

If two distinct balls fall into the same bin, and we consider it as a collision, then the amount of collisions directly translates to the amount of balls in each bin. This will tend to be similar across all bins. However, relative to our current problem, there are a set of practical restrictions, that cause a deviation from such expected uniform behavior:

- Each bin has a maximum capacity. Once a bin is filled, to accommodate a new ball, it implies that some other ball must be discarded.
- Each ball has two weights: (1) size weight, and (2) popularity Weight. The first weight is fixed per ball, and the second is a dynamic value that changes over time for each different ball. These weights regulate which balls should be discarded, if needed.
- Re-insertions, fall in the same previously used bin, and affect (2) the weight popularity.

A clear understanding on imbalance in caching systems, allows the distinction of three main vectors as a source of load imbalance: (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. We note that in the work presented in this thesis we do not tackle the imbalance due to varied object data sizes. However, for completeness we also discuss it in this chapter.

2.4.1 Data Placement

Achieving uniform distribution in data placement comes as a key aspect in distributed storage systems. In this particular case, distributed caching 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. Requiring more nodes to accommodate the same global amount of data items.

Previous work [14] 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, placement 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 object (or 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 $hash(O)$ $mod(N)$. This 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 an 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, similar to the resize operation in the $HashTable$ example, it is required to perform a remapping of all to keys to servers, which is a very expensive and disruptive operation.

2.4.1.2 Consistent Hashing

This method, described in [17], 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 (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 $[0, U[$, so 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 a server leaving means that 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 technique popularized by *Dynamo* [7, 8], which is the 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 to, or by concatenating the data item identifier with a salt value in the range $[0, K[$.

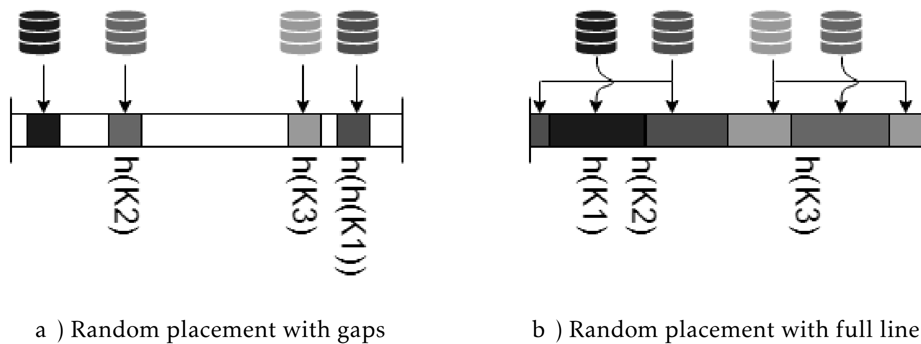
2.4.1.3 Highest Random Weight

This method described in [39] exploits the properties of an 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 [6] , ASURA [14], and SOURA [47]. Although this require a retry mechanism in order to pin point the target segment, where this can be accomplished by iteratively generating random number using a PRNG 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 [24], 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 these 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 [36, 37], to avoid multi-hop routing. Instead, solutions where each node knows the global membership of the system are favored [12].

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, *apriori*, ensures uniform distribution [23]. 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 presented in [24] and [14], we can establish a base comparison between several of these algorithms in terms of fairness, memory usage, and lookup time. In this context, fairness refers to the ability to evenly spread a large set of data items over a finite set of nodes.

The comparison is presented in Table 2.1, and it shows that Consistent Hashing with a high amount of virtual nodes incurs in high space overheads. This is required to maintain the logical ring between all servers. The 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.

Due to this, we will explore this alternative as a way to mitigate object assignment imbalance on distributed caching systems.

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, making the server a bottleneck in the system due to large request reception queues. This affects the response times of all requests received by that server [33].

Since it is unavoidable to have some data objects that are accessed with a much higher frequency, it becomes relevant to identify them, so that they can be flattened. In some solutions, the servers can share objects while striving to achieve a uniform local distribution in terms of the number of request received per time unit. There are multiple ways to identify such *"hot"* data items both locally (in the context of a server) and globally (across all 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 [13], is an example of such a 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, efficient, and non-disruptive way. To do this, Spore strives to identify and replicate the data items with highest access frequency, as this will share the load in serving that object with another server, promoting load reduction and ensuring the availability of the (highly popular) 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 measures to be taken as to avoid having 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 locally gathered 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 observes a uniform sample of requests, 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 [6] 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 time for those data items. As explained in [33] *any typical front-end cache reduces load imbalance*, however it might incur in coherency problems that is usually dealt by caching objects at that layer for every small time windows.

2.4.2.3 Discussion

Dealing with load imbalance 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 lower response times 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 times.

On the other hand, Two-Tier caching mechanisms also provide a good solution, since it directly reduces the load on the caching servers, while providing faster response times for 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. Is the context of this thesis we explore the use of replication, as we are addressing caching solutions that can support any application, and two-tier caching can be damaging for applications that have significant complex processing happening on web application servers.

2.4.3 Data Object Size

Data object 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 scenario, there are several approaches that aim at solving it, such as the use of RAID levels, allowing parallel access for large data objects, usually applied in the context of DBMS, and disk arrays. This allow a wide 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 is akin to the use of Erasure Code [32, 43]. 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 at the client. On the other hand, the overall response time is conditioned by the slowest server across all servers serving chunks of the data object.

In (2) memory management, there are multiple scenarios where data size can have an impact on the system performance. Although, 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 on 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, move memory management to the application level, where large heap memory segments are allocated and managed according to some application logic. It typically benefits the execution time, by avoiding the frequent memory allocation and release. However, in cases of high variance in data item sizes, it may result in memory under usage due to memory fragmentation.

The analysis carried out in [33] shows that load imbalance increases with the amount of shards in relation to the coefficient of variation of the data item size. While chunking (splitting 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.

2.4.3.1 Discussion

We do not analyze any further the data size imbalance, since the Slab Allocation, currently in place in Linux systems, is highly successful. It provides one of the best designs for memory management to avoid memory fragmentation, since all the data items are segregated by size, and the allocations are performed in batch.

2.5 Data 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 the object to be read simultaneously by several clients with lower interference, while remaining available even if some of the copies of the object disappear. A common approach in replication relies on having the replicas at different storage devices or machines for improved fault tolerance (i.e, to avoid correlated failures).

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 times more memory consumption. When thinking about cache systems 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. Therefore, it becomes essential to decide the actual importance of fault-tolerance in a distributed cache.

Also, under the assumption of failures not being frequent and information being written in a persistent manner elsewhere, the overhead to ensure fault-tolerance, given the limited amount of RAM available and the additional required coordination to maintain some degree of consistency among the the different replicas of a data object whose value can be update, might not be the ideal solution for a distributed cache system.

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 overall performance of the system and as a technique to mitigate imbalances due to varied access frequency for different data objects.

There is an extremely large set of replication schemes, and presenting them all goes beyond the goals of this thesis. Instead we define a set of dimensions in which we categorize most existing 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 of generating client responses.

   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, other replicas are only used to read data objects (or fault-recovery).

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

Any replication scheme can be seen as a composition of design choices among these three dimensions which present trade-offs between performance, fault-tolerance guarantees and replication semantic (often refered as data consistency).

2.5.1 Data Consistency

Replication becomes a non-trivial challenge when we allow more than just read interactions with the data. The operations that change data items (commonly refered as write operations), can lead to a divergence in the data kept for the different replicas on different points in time. Such inconsistencies can be exposed to clients which can be unacceptable

for same applications. An usual requirement is to maintain some form of consistency over the replicated data items exposed to clients.

Furthermore, there is a need to specify the restrictions on how data can be perceived by the client. This specification of interactions follows some norms that are called data-centric consistency models [25, 26]. 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 replicas, where any read operation returns the effects of the most recent write over a data item.

Sequential Consistency There is a total order of events across all replicas (both, reads and writes), implying a coherent history of the system that justifies all values exposed to clients.

Causal Consistency Write operations that are potentially causally related must be executed by all replicas in the same order. Concurrent writes might be seen in different order by different replicas (and consequently, clients).

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

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

Eventual Consistency Updates are propagated on background, where there is no total ordering of operations, and conflicting updates over a given data item must be dealt by some additional mechanism. Clients can observe any value of an object at each read operation despite the order or the client that issued different write operations over that data object.

2.5.2 Replication Protocols

Caching systems usually demand fast response times, 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 a 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 times(where N represents the number of replicas). Still, this does not happen, since having multiple instances requires coordination and enforcing appropriate consistency guarantees. This implies that there is a trade-off between the benefits of replication and the coordination overhead required 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 (in this case eventual consistency).

A Caching system needs to enforce 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 arbitrary 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 slow 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 approaches and discuss their usefulness for improving the design of distributed cache systems.

2.5.2.1 Primary/Backup Protocols

Primary/Backup Protocols [4, 38], 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 on any replica. When the primary fails, the protocol blocks until a new primary is elected among the existing backup replicas.

2.5.2.2 Chain Replication

Chain Replication [31] 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 receiving all write requests from clients;

Middle set of nodes between the head and the tail;

Tail node responsible for handling all read requests, and issue replies back to clients;

This scheme requires all write operations to be processed by the head, defining a total order for all write operations. Afterwards, write operations are propagated down the chain to the tail, 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.

Notice that despite its simple design, the protocol provides high robustness since it allows tolerance up 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 potentially read stale data by interacting with any server in the chain. Contrary to the expectations, it might under-perform in the presence of more than 15% write operations when compared with the original chain replication protocol [31].

2.5.2.3 ChainReaction

ChainReaction [3] is a storage system that relies on a variant of chain replication, where the core concepts remain the same, but relaxing 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 Kth node establishes a linearization for all operations. The second segment of the chain receives updates through lazy replication (lower priority than the other write requests). Clients are allowed to read at any replica albeit, with restrictions as to enforce casual consistency guarantees.

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 Discussion

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

The analysis of Facebook's [27] deployment shows that cached data items tend to be a monotonically increasing snapshots 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 of time. Furthermore, 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 [3].

Furthermore, in our use case when considering ChainReaction 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 assume no fault tolerance, if a server fails, all the writes in progress will have to be discarded. This can be tackled by manipulating the value of k akin to what is proposed in [3].

2.6 Cache Interaction Models

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. The interaction with both storages can be categorized by the read and write operations.

For the read operations there are a few choices: (1) interact only with the primary persistent storage (no cache interaction), or (2) try to fetch data from the secondary storage, and if not present (miss) fetch from the primary storage and insert it into the secondary.

The write operations can in most cache systems be described through its hits and misses, where a hit mean a specific data item already existed in the cache, and the miss its opposite.

In terms of a hits, the two possible scenarios are: (1) Write-Through, where the write operations are applied to both storages, or (2) Write-Back, where the write operations are only applied to the secondary storage, and latter on (mostly when data items are replaced or evicted) to the primary.

The Write-Through approach is several times slower than Write-Back, since it has to wait for the operation to be performed in the slower storage. However, it allows crash faults in the secondary storage without compromising the consistency of the primary.

In terms of misses, the two possible scenarios are: (1) Write allocate, where the write operations are performed in the secondary storage, and (2) No Write Allocate, where the write operations are performed on the primary storage and not to the secondary.

These two follow different perspectives. The Write Allocate tries to exploit temporal locality, since it assumes that there is a high likelihood of read operations afterwards. The No Write Allocate, is the exact opposite, by trying to reduce the secondary storage evictions, by not wasting space for a data that might not be read.

Even thought there are several combinations of behaviours of hit and miss patterns we will be assuming hence forward Write-Through with Write allocate as a common configuration for web environments.

2.6.1 Cache Coherence

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 component in the system.

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

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, (variable) latency in message transmission, and request 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 [38]: (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 [10] have 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 to validate the freshness of cached data leading to additional delays in accessing data objects.

A naive solution would be to add to the queries response from the primary storage a parameter such as the current time, 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, most databases however, do not provide this.

We note that the level of consistency offered by the distributed cache system, even if strong, is not ”relevant” if its contents are not coherent with the whole system (in particular the contents of the primary storage system).

2.6.2 Handling Stale Data

In caching systems there are known access 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 privileged in 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 on the primary storage component, these changes will not be reflected on the cache immediately, potentially making those changes not observable by clients. In these situations, there are common solutions like invalidation schemes, but this type of mechanisms introduces some overhead in the system since it requires more coordination with the persistent storage. 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.

2.7 Memcached

In our work we use Memcached [2] [1] as a case study, which is an open source, high-performance, distributed memory object caching system that can be found in large companies with data intensive workloads such as Facebook, which have read intensive access patterns, where data objects have significant variations in popularity. 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 an HashTable with a key-server mappings managed by Consistent Hashing. Only one copy of an entry is kept for the whole system. This solution has some obvious drawbacks (most of which have been 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.

A Memcached instance is composed by a Cache Item Data Structure (CIDS)(see Figure 2.2)[44], which is an aggregation of data structures to manage data items, containing

[1] implementation wise, we consider version 1.5.10 available in https://github.com/memcached/memcached

several components: (1) HashTable, that is used to locate a data item in CIDS , (2) Eviction Mechanism, and (3) Slab Allocator.

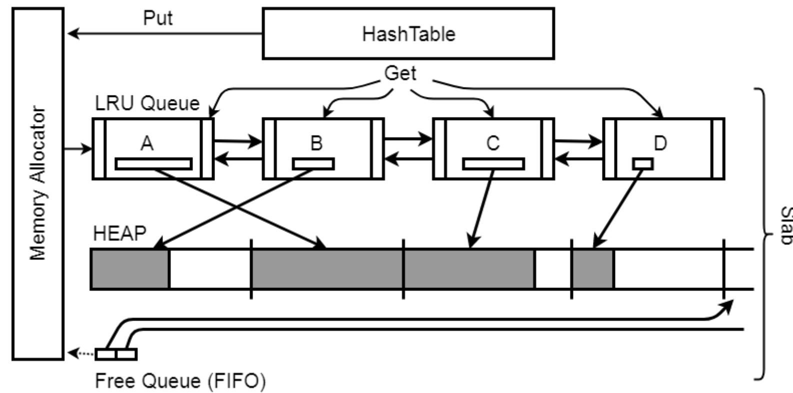


Figure 2.2: Cache Item Data Structure (CIDS)

2.7.1 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 and lowers the overhead in memory allocation since it avoids allocating, initializating, and deallocating memory for data items individually, which otherwise would require the use of slow system calls [44].

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 of Memcached presented in [5] evidences that Memcached can have close to 10% of memory under-usage.

2.7.2 Eviction Policy

In Memcached, 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, Segmented LRU policy based on the 2Q Policy variant, more specifically the OpenBSD variant.

In the former (1) eviction algorithm, each slab contain an LRU built using a doubly linked list, where there were no working threads to maintain it, and items were either moved to the head at each access or eventually discarded when reaching the tail.

This had serious implications, since according to the authors of Memcached, "the Multi-threaded scalability is heavily limited by the LRU locking". Furthermore, it also refered that scaling beyond 8 worker threads in this simple LRU policy was dificult.

Due to the scalability problem, the Segmented LRU algorithm is currently in place. This algorithm consists of a set of queues (sub-LRU's), HOT, WARM, and COLD, where items are ”bumped” between queues asynchronously. By itself these queues do not solve the problem, as such, there are two special threads to manage them:

LRU Maintainer Thread Responsible for managing all the sub-LRU's, peeking at tail items, reclaiming expired items, moving items when necessary, and process asynchronous bumps.

LRU Crawler Thread Examines all sub-LRU's for each Slab class in order to reclaim expired data items. It inserts special crawler items in the queues that traverse from tail to head, using a round robin policy at item level across the HOT, WARM, and COLD queues.

This policy has allowed the system to scale quite well, as pointed out by the authors: "Very read heavy workloads have scaled nearly to 48 cores once optimizations were made.". These improvements have greatly reduced LRU lock contention, since items are never bumped when directly fetched, with no increase in per item meta-data size.

The major difference the latter eviction policy has when compared with 2Q policy, resides in the shadow queue that is used to keep an history of previously cached objects to determine what was previously in the cache (and evicted meanwhile) by only storing the keys. In the variant currently employed by Memcached, there is still a notion of three queues, but now they are named HOT, COLD, and WARM, where all these queues store data items.

The management is done according to the following strategies:

HOT queue ingress point on the segmented LRU, and only a transition state (there are no bumps within this queue) before an item is sent either to WARM queue if it has received at least two hits, or to the COLD queue otherwise.

WARM queue items that received at least two hits when reaching the tail, are bumped to the head of the queue, otherwise they are sent the COLD queue instead.

COLD queue places at the head of the queue data items evicted from HOT and WARM queues. Data items at the tail that received at least two hits will be queued asynchronously in the WARM queue, otherwise, they are 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.

2.7.3 Concurrency Control

Memcached is naturally multi-threaded to process clients requests, where there is a set of available operations which can be split into 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 (i.e., modifying the value of a data item).

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 [44].

The HashTable is managed by the maintenance thread, responsible for performing the HashMap resize operation when the data item count threshold is triggered. It forces all worker threads to pause, until the resize is finished. This threshold verification is a timed event.

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.

2.8 Memcached Variants

2.8.1 R-Memcached

R-Memcached [19], 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, therefore 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 low overhead.

2.8.2 Spore

Spore [13] 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 records 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 these data items, SPORE makes use of a very small cache ($\simeq$ 3% of the size consumed by all 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) [20] 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, that they store locally.

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 for that object. 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 (that is the variant featuring the second mechanism) uses Two-Phase Commit in order to ensure strong consistency.

2.8.3 CPHash

CPHash [21, 22] is based in an isolation conceptual design, in which the main goal is to create a hash table that works well on CPUs with multiple cores. 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 functions. 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, 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.

Each partition has several buckets and chains of data items (similar to an HashTable) to locate a data item, which uses the former (single LRU per SlabClass) eviction policy of Memcached. The partition scheme does not require locking due to the exclusive access policy, and since there are no other threads other than the partition owner meddling in eviction process.

2.8.4 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 [18], 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 CPU 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. It also maps the requests directly to the specific CPU Core at the Network Interface Card (NIC), redirecting the request to its respective receiver buffers without any high level interference.

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, CPU).

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. This 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 a 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).

2.8.5 Other Systems and Proposals

Intel developed a version of Memcached [44] exploiting concurrency control and increased parallelism. The main focus was achieving finer-grain locking for the underlying data structures while applying light weight locks (over critical sections). 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 a (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 [28].

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.

2.9 Summary

In this chapter we discussed the sources of load imbalance in caching systems, how they are detected, existing approaches to deal and minimize them, as well as other system implementations that explore some of those topics.

This analysis allows us to understand the key features surrounding caching systems and how to maximize resource utilization, namely through partitioning, and data distribution.

Considering the previously referred partitioning schemes, their main advantage are the lack of lock contention.

However, indexing partitions seems to pose additional challanges, where some utilize in-server clients for partition indexing, which wastes a large amount of processing power (close to half of the available CPUs), while others use direct NIC Access to process high level packets.

The following chapter presents the conceptual design of the proposed solution, which combines some of the ideas explored in previous work and novel ideas with the goal of leveraging the load among CPU Cores for maximum utility.

Chapter 3

Conceptual Design

This Chapter presents the design of our solution for caching systems that use a memory partitioning scheme with exclusive access. The Chapter is divided in three parts. It begins by providing a description on standard caching systems and discussing their key characteristics. This is followed by the presentation and discussion of the design of our solution of distributed cache systems, which is the main contribution of this thesis. Finally, the Chapter provides the key insights on making clients compatible with our proposed distributed caching system.

3.1 Cache Systems

Caching systems operate as a secondary storage for transient data items, which are maintained persistently in a storage system. In most scenarios, these systems are deployed within private networks. Figure 3.1 depicts one such scenario, which is composed by three main components (highlighted in the figure):

Cache Pool: Contains the set of servers dedicated to caching. These, commonly execute an in-memory key-value store in isolation, where each server has its own limited resources.

Registry: Functions as a middle man between the caching servers, the clients for server discovery and server membership updates, keeping up-to-date information about the servers through an independent coordination system that is fault-tolerance (e.g., Zookeeper [16]).

Client Driver: The component that provides clients with a logical view of the caching servers as a single instance.

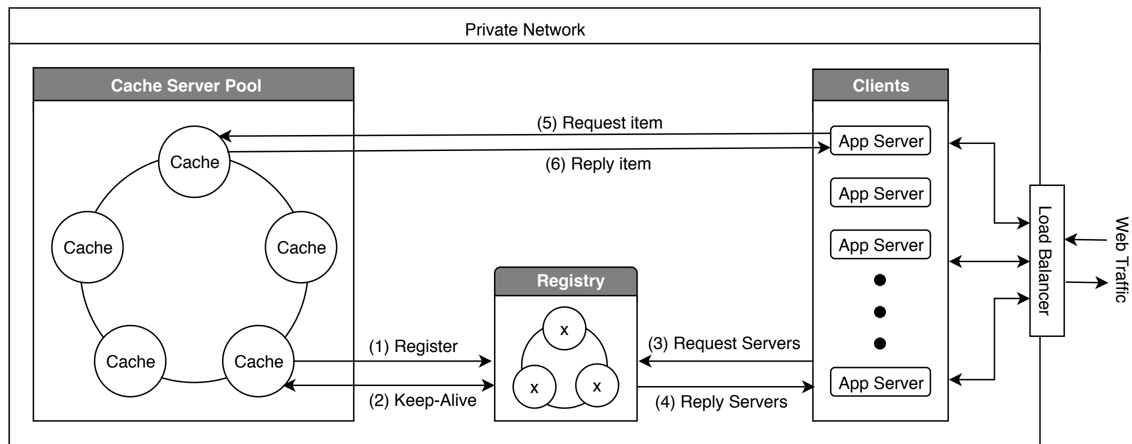


Figure 3.1: Cached pool interaction

These three main components are the foundation to ensure the correct operation of the system. Typically, the system operates following a structured interaction as shown by the numbered arrows in Figure 3.1. A new cache server joins the cache pool, by (1) registering itself in the Registry's membership to be considered active. The new server is required to have a persistent connection (2) to the Registry, making it possible for the Registry to verify the server's alive status. If a server fails, the connection is lost, making the Registry discard that server from the membership.

For application servers (i.e., clients) to become operational, they require information regarding the cache servers, which is acquired from the Registry by fetching the current membership of the cache server pool ($3-4$).

The application servers receive inbound web traffic, and perform some application logic in order to provide a reply to the application clients. The information for the application logic is retrieved by querying the caching layer ($5-6$). This is achieved via the client driver that contacts the server (or set of servers) that contain the requested data.

If the request data is not present in the caching system (miss) the application server will be forced to retrieve the data from a slower persistence storage elsewhere, or in some cases, the cache could do it transparently for the client.

3.1.1 Cache Server Model

Caching servers typically demand reduced latency when fetching data items, while dealing with a high volume of client connections. As such, in most cases, this is accomplished by an event-driven model sustained by a thread pool, as depicted in Figure 3.2.

In this model, incoming requests are queued by arrival order, where each request will trigger an event associated with a function/procedure (callback) that contains the logic for the reply. The event loop manages those event triggers that will be retrieved according to thread availability, where each thread becomes responsible for answering a given request and fetching another as soon as the last is finished.

The amount of threads is limited and typically created in the server bootstrap phase (before the server is considered operational), remaining alive during the full program execution. This set of threads is commonly referred to as a thread pool.

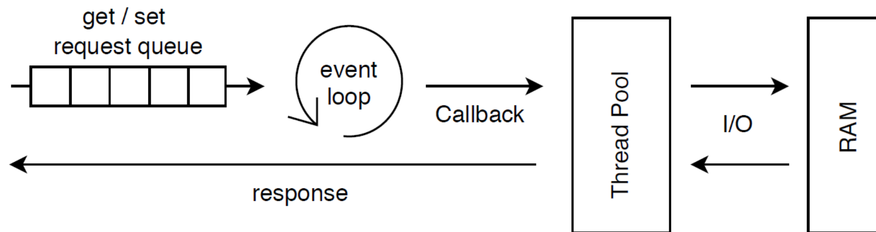


Figure 3.2: Cache server layout

To answer a request, a thread is required to fetch (or write in the case of the set operation) data items stored in RAM. However, the threads are working simultaneously over the same memory space (shared memory access). In this scenario, several working threads can access the same data item, while resorting to mutual exclusion to ensure consistency.

This mutual exclusion at the data item level is commonly achieved through the acquisition of a lock (or set of locks) before accessing a data item, which imposes some overheads and limitations on data item availability. The lock acquisition itself can be considered a fast process, and it does not comprise the majority of the delay when accessing a data item.

This delay is caused by the contention generated around a data item, which is more notorious as its popularity increases, and threads compete among them to hold its lock.

3.1.2 Data Item Availability

Sharing data items among several threads is one of the most popular methods to increase resource utilization.

However, instead of having the threads compete with each other to hold the data item lock, which generates contention, one can assign the subsets of data items to separate memory spaces (partitions) and only grant access to one thread per partition.

This hypothesis tries to exploit increased parallelism, where the lock contention in the best case is completely removed, since each thread is only responsible by a subset of items. Hence, it also responsible by the respective subset of the request that targets only those items, providing additional data locality.

3.2 Partitioned Server

Increasing parallelism through data segregation in the form of partitions, which are owned and only accessed by a given thread, aims at maximizing the overall efficiency

achieved by each individual thread, striving for improved server latency, which inherently improves the performance of the system in its distributed context.

The partitioned server can be achieved considering a single cache server instance, inspired by MICA and CPHash, where the total reserved memory can be split into equal-sized memory segments (partitions).

Each of these partitions is uniquely identified and is only allowed to be read or written in an exclusive mode. Thus making it is impossible for any two worker threads to access a partition at the same time, removing the explicit notion of concurrency when accessing the cache memory, in which is typically located the main source of contention.

Since the threads are now responsible for its own memory segments, which consist on a set of data items with an associated popularity, it becomes advantageous for the threads to be bound to a CPU Core. It avoids unnecessary context switching by avoiding moving the thread between CPU Cores, and achieves improved data locality by avoiding the invalidation of the CPU caches (e.g., L1, L2, and L3).

However, the concept of a thread owning a partition (or set of partitions), implies that each thread has to deal with all the load of accesses over that partition.

There are cases, in which a given partition might hold several high access frequency data items. That might introduce delays when accessing data items in other partitions if both are managed by the same CPU Core, due to large request queues associated with that memory partition.

As such, in our solution the partitions are not bound to any specific thread, as it would provide a sub-optimal solution. Instead, there is flexibility through thread mobility, since the partitions can be redistributed among the worker threads, giving them the opportunity to improve their access time.

3.2.1 In-Memory Management

We have explored our proposal in the context of building a novel partitioned version of Memcached with exclusive memory access while maintaining all its internal management policies as close as possible to the original Memcached.

Exclusive access is achieved by dividing the total available memory among the partitions, where each partition has its own slab memory. As seen in Memcached, slab memory provides a great benefit regarding performance, since it does not require constant reallocation of memory, while maintaining reduced memory fragmentation.

However, in this particular case, exclusive access becomes a loosen term since it only holds from the clients perspective. In practice, this is not entirely true, since internally there are other auxiliary threads that interact with the data items and slabs. Thus, locking is still required, only resulting in very low contention when accessing a data item (and hence, additional parallelism and less CPU time being wasted).

Client accesses to data items in the Slab memory are conducted through an Hashmap, where each partition has its own map, that is intertwined with the respective segmented

LRUs of the partition.

Each partition having its own Slab memory implies that in the server there can be N slab classes of the same size, and inherently N segmented LRU's for each slab class instead of one. This allows the lack of lock contention to become more notorious as concurrency increases, especially in write operations, since LRU and slab locks (coarse) have to be acquired.

Increasing the amount of locks has reduced impact on get operations, since in most cases, there is no need to acquire either the LRU or the slab locks to fetch an item.

The partitioned scheme provides a very low increase in space overhead, since only the amount necessary to maintain the additional control structures is not used to code data objects, the total amount of data items remains (mostly) the same.

Furthermore, we allow two equal data items to reside in the same server. In shared memory this would naturally pose a problem, where typical approaches to deal with it would be to rename them and store two distinct keys with some mapping for a data item. However this would increase the amount of meta-data and only takes into account shared memory systems. Since we are dealing with exclusive memory, there should be no need to rename an item, since data items reside in distinct partitions, and are fetched by different hashmaps. This effectively allows to atribute more CPU time to process request for highly popular data objects

3.2.1.1 Item Lock Acquisition

Data item access is regulated by distinct HashMaps, one per partition, which provides some isolation between memory segments. This allows for new situations to arise, such as, the possibility for two equally named data items to reside in the same server in different partitions. The partition scheme should naturally adapt to this scenario. However, in Memcached locking is still required due to existence of management threads that access all data items.

In this particular case, data item locks are acquired at the scope of the process, through the use of an hash function that takes the data item identifier as its argument, causing two equally identified data items to share the same lock. Keeping the lock acquisition only based on the data item identifier would generate the same contention as if the threads were competing to access the same data item, removing the possible benefits of having two equal data items, only taking its toll.

In shared memory systems the problem of dealing with two equal data items is typically solved by renaming data items and storing two distinct keys with some mapping for a data items, but this increases the amount of meta-data.

Instead, we are dealing with the abstraction of a partition, where there is no need to store any additional information or extra memory required to distinguish the data items, since each partition is uniquely identified.

As such, by prepending the partition identifier to the data item identifier, and taking advantage of the Hash function properties, it allows the same data item to map to different locks depending on the partition it resides. (note that we choose prepend instead of append, since it would cause more entropy in the output of the Hash function due to the avalanche effect.)

3.2.1.2 HashMap Resize

In the original Memcached version, the HashMap is a shared resource, monitored by the main thread in a timed event. Each time the HashMap data item count surpasses a given threshold, it triggers a re-size, that is carried by the Maintenance Thread.

The Maintenance Thread has the sole responsibility of performing re-sizes on the shared Hashmap. Each time it is activated, it requires the system to halt, forcing all worker threads to pause, resulting in server downtime.

Given that we are working with partitions, and each Hashmap is independent, only a thread, or set of threads, should be required to pause according to its respective re-size thresholds at a given moment.

Since the HashMaps are no longer a shared resource, there is no need to keep the Maintenance Thread. The main process can directly inform a thread that a partition it owns has surpassed the size threshold, and that it needs to perform a re-size.

This implies that each thread might pause several times, according to the number of partitions it holds. Also, the threads halt independently, and for small amounts of time, when compared with a re-size of the Hash map with all the items.

The overall time in which each thread is stopped should be slightly higher than in the original version, due to the additional memory allocation for the expansions that have to be done in each Hashmap. This method avoids system downtime, by taking short pauses for each thread according to its necessities (that are not necessarily synchronized).

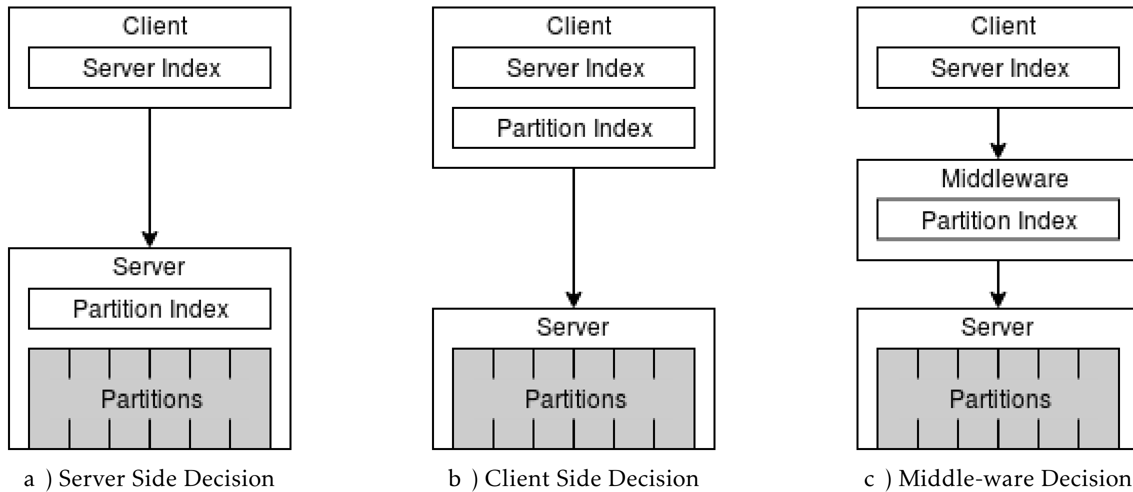
3.2.2 Partition Access

As seen in CPHash, one could have a top layer in the server responsible for directing each request to its respective partition, as depicted in Fig. 3.3 a). However, this would increase the number of threads, context switching and number of messages.

According to the authors of Memcached, the recommended number of server threads should not surpass the number of available cores for maximum usability. Hence, if one adds a set of threads to act as clients, it will decrease the CPU time for each thread and increase context switching, since there are far more threads than CPU Cores. Each request would be required to go through internal client thread for indexing beforehand.

Furthermore, the communication between client and server threads is commonly based on message passing, where each request will demand two event triggers to be performed in different threads, while containing the information of which socket to attend to. This implies that the request has to be processed twice, or a connection item has to be

allocated in the client thread, passed to the server thread for processing, and finally freed before the reply.



a) Server Side Decision b) Client Side Decision c) Middle-ware Decision

Figure 3.3: Two-Tier Indexing Schemes

To avoid all this computation and its downsides which impacts negatively server performance, we expose a socket port for each partition, creating a multi-port server, and push partition access decisions on to the client side, as depicted in Fig. 3.3 b).

This is achieved by performing the computation for partition indexing in the client itself. The client possesses all the means to perform indexing simply by knowing the amount of statically configured partitions in a server. If different servers have a different number of partitions, one could add that information to the registry. From here on, it would only require a step further than the normal indexing scheme already performed by the clients, where it first chooses a server, and the respective target port afterwards.

Furthermore, there is also the option of relying on midleware to perform the second tier of indexation, as depicted in Fig. 3.3 c), which might solve the downsides of the CPHash model benefiting its throughput. However, it would incur in increased latency.

3.2.2.1 Summary

The exclusive memory access policy is represented by a set of sockets that are strongly correlated to their respective partitions. At any given point in time, each worker thread is responsible for a subset of partitions. Each subset only allows the respective worker to access a disjoint set of data items from all the data items present in the server.

Even though each partition has its own memory management, achieved through individual Slab Structures and respective LRUs, the presence of a single HashMap per partition is what provides the necessary abstraction to ensure the goal of restricting the clients access to each partition.

Entering a paradigm of exclusive access relieves the data item lock contention problem, and introduces new bottlenecks, mostly due to the fact that each partition is seen as

a block that has to be managed by some thread.

If the load caused by the access to a partition or set of partitions, is greater than the CPU Core capacity that those partitions can possess, then the access to all those items will become negatively affected.

In particular, the number of operations that can be performed in a single CPU Core is limited and will be the cause of additional server latency, by keeping requests waiting to be processed for a longer amount of time in a given partition or set of partitions.

This latency will be perceived by the client, and will limit the overall throughput when compared with Memcached, since it is able to distribute the load across the threads, and decrease the waste of CPU time when processing client requests.

In our case, the server will become exhausted when the first CPU Core reaches full utilization, at this point the server cannot process more requests per time unit without a penalty on response times.

3.3 Local Access Frequency Imbalance Management

Detecting load imbalances does not necessarily imply that a partition holds the most accessed data items. There are several circumstances where a partition or set of partitions might experience increased latency, mainly due to the load on the CPU Core being employed to execute other unrelated tasks.

A given CPU Core might be exposed to more load than simply the worker thread answering the client's requests, it might also be subject to the Main process, Maintainer Thread, Crawler Thread, or even external tasks that the Operating System (OS) triggers. All the threads that are not the worker threads are considered free threads, which have to be assigned to some CPU Core through the OS scheduler.

To detect cases in which a CPU is overloaded, we monitor each CPU and calculate its occupation as a percentage in a one-second time window. However, this percentage only reflects the occupation over the last second, which is too sensible to load peeks. That might be caused by a (short) burst of requests, or some free threads that are temporarily using a CPU Core.

As such, we amortize the load over a given time period using the Exponential Moving Average (EMA) [40], providing a concept similar to the average where the most recent readings have more weight, giving us some sensibility to the tendencies, and not a flat estimate of the load. Furthermore, to handle the load we should not intervene with any of the free threads, which might rely outside the program scope. As such, our solution to relieve the load requires managing the partitions, and for a more fine-grain solution, the data items.

3.3.1 Decision Making

The first step to make a meaningful decision is to understand when exists a problem which needs handling. For this we define an overload state for the CPU Cores, where some threshold is defined and when surpassed, triggers a mechanism to readjust the load, if possible.

As suggested in Spore, a threshold in the receiver queues that expresses the latency requirements for an application, might guide the design of an adequate solution. However, taking into account that there are two main types of connections: persistent, and non-persistent, marking a threshold in the receiver queues of each socket is not a plausible solution for persistent connections. For example in cases such as TCP connections, the socket that constitutes the ingress point on the system only deals with accepting new connections. Hence, to make a measure of how much load a partition is subject to, would require keeping track of all the child socket queues and establish some cumulative threshold.

As such, we apply a threshold for each amortized CPU Core utilization instead. When the threshold is surpassed (overload state), it is probably due to the high access frequency of data items, which triggers a re-balancing mechanism where the first approach tries to balance the server itself. It does so by redistributing the partitions by the CPU Cores, relieving the bottleneck in the overloaded CPU Core.

In this scenario, it becomes convenient to have several partitions per thread, which allows a more divisible load and better internal rebalance. The load in each partition is described through the use of the Exponential Moving Average (EMA) over the number of accesses to each partition per second.

The current load imposed over a thread can be inferred based on the amortized amount of accesses to each partition and the current distribution of partitions by the threads.

Furthermore, a clear insight on the discrepancy between the number of accesses to the threads can be achieved by computing a standard deviation. If the standard deviation exceeds a specific threshold, it will try to redistribute the partitions by the threads while aiming at achieving a lower standard deviation over the number of accesses managed by each thread.

The best outcome of this rearrangement can be solved through the Minimum Makespan Problem [11, 42], where the algorithm considers a set of tasks with a given duration to be distributed by a set of processing units with minimal variance.

In our case, instead of task with a duration, we consider partitions with an associated load, which are analogous. However the exact minimum is an NP-Hard problem with an exponential time complexity, as demonstrated in [42], which would severely damage the system performance as the number of partitions increases, and since this has to be calculated on a regular bases.

As such we turn to an approximation of the algorithm for load balancing, more particularly, the special case in which there is no partial order between "tasks". This algorithm provides a very good approximation in linear time, as described in Equation 3.1, where $c*$ represents the optimal value (minimum load), while c is the load that resulted from the approximation algorithm, which aims at $\frac{c}{c*}$ staying relatively close to 1.

$$\frac{c}{c*} \leq \frac{4}{3} - \frac{1}{3m} \tag{3.1}$$

The algorithm itself, consists of ordering the partitions in decreasing order of their load, and assigning each partition sequentially to the thread with the lowest cumulative load.

If there is no viable rearrangement of the partitions by the CPU Cores, or the new standard deviation generated based on the approximation algorithm, is still above the threshold, it will resort to replication as a more fine-grained load relief method. It will select a set of data items with the highest access frequency from the partition with the highest amount of accesses of the overloaded CPU Core, which should translate to the most load relief.

3.3.2 Socket Shift

As mentioned before partitions and sockets are strongly correlated, where the socket regulates the access to a given partition. Any partition itself is reachable from any thread since we are dealing with an abstraction over shared memory. So we only need to relocate the ingress point of the access to a partition in order to rearrange the partitions by the CPU Cores, as depicted in Figure 3.4.

For persistent connections one requires to keep track of which connections are associated with each partition. This is required so that if load rises, triggering a rebalancing, we can shift them to another thread maintaining the exclusive access policy. This also requires to stop the target thread until all the relevant connections are moved.

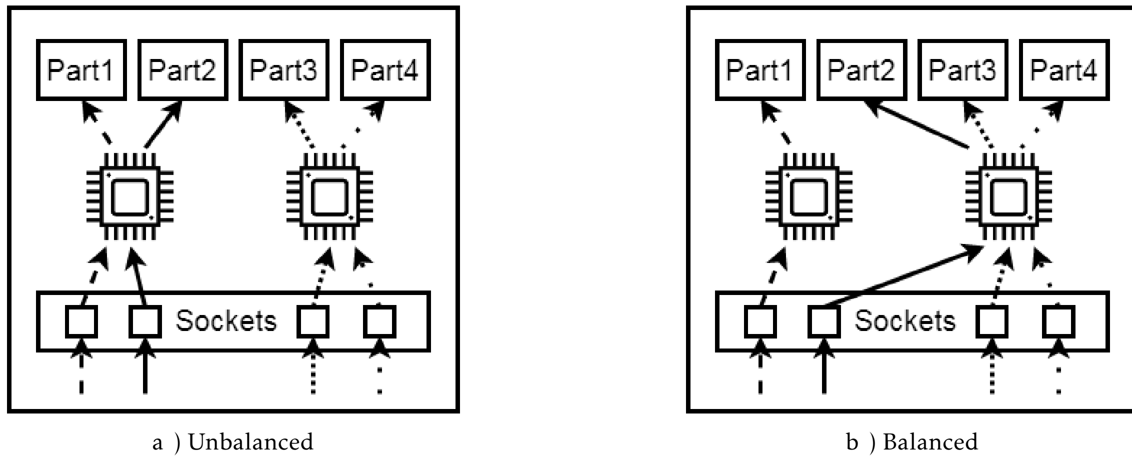
However, this does not work too well for non-persistent connections, since it would require constantly adding and removing connections from a tracker data structure, which can lead to significant bookkeeping overhead. Performing these operations would impact the server latency, as well as an increase in the amount of meta-data to establish the correlation between the all the sockets and the partitions.

To overcome these challenges, we developed the socket shift algorithm that requires very low meta-data and is agnostic to connection persistence.

We do not try to keep track of all the sockets from a top-level view, instead we grant to each socket the ability to verify if they are in the same thread as the socket that originated the connection (master socket). This validation is done before each operation, since the algorithm only moves the master socket to the desired thread.

When a child socket notices a change in ownership, it will move itself to the thread where the master socket currently resides.

The shift is not a forced bulk of connections being moved, instead, each connection moves itself when required to. The result will be similar to the bulk, but will still allow operations to be performed by the worker threads interleaved with the shifts.



a) Unbalanced

b) Balanced

Figure 3.4: Correlation between sockets and partitions with exclusive access

3.3.3 Selective Replication

Selecting popular data items is a non trivial task, since maintaining access frequency information over all data items becomes impractical due to the necessity of high amounts of meta-data. Using sampling, might prove to be a reliable mechanism, since it allows to pinpoint exactly which are the most popular items.

However, sampling also requires some permanent extra memory to keep track of the most requested items, while having to work as front end cache for each partition separately.

This would increase the sever latency, since all request would have to go through the front-end cache first, before accessing a partition. The first cache would need a high performance to avoid increasing the server latency to much.

Thus, we arrive at our solution, a form of replication that tries to exploit the peculiar characteristics of a cache, that typically does not occur in normal storage systems, which takes advantage of the partition scheme, and its internal LRU's.

Knowing the LRU policy present in the cache allows to estimate the location of the most popular data items. However, it does not allow to pinpoint precisely which are the most popular items without incurring in a persistent increase on management metadata. The amount of partitions minimizes the probability gap when selecting the most popular by restricting each decision to small subset of data items.

As such, we cause a controlled burst of replication with fast shrinking, and set an upper bound on the number of data objects that are replicated per burst, and overall. This will result in only the truly popular items maintaining their replicated status.

The burst is controlled by a single background thread that performs the replication by increasing a γ value for each selected item that represents its replication level (replica count), similar to the mechanism used by Spore.

The shrink phase is controlled by a threshold in the LRU, in this particular case, performed either by the Maintainer or the Crawler Thread, when a replicated item drops from the WARM queue.

Common sense would say that replication should be done across servers, but this is typically under the assumption that servers are multi-threaded, working with shared memory access, and the main focus besides availability is mostly fault-tolerance.

These assumptions do not hold in the current scenario, since even though it is a multi-threaded system, each worker thread performs its tasks mostly in isolation, since each partition can only be accessed by a single worker thread at any given time, which is perceived by the clients as a single server itself.

3.3.3.1 External Replication

Replication typically takes consistency over the replicas as a prime concern, since the closer to strong consistency, the higher amount of coordination and latency is required to manage the replication.

In caching systems, the information is transient and needs to have a fast data retrieval time. Hence, a notion of causality should suffice. Taking as an example a variant of ChainReaction, configured with a K value of 1, such replication solution enables write operations to be performed on the "home" node of a data item, and respond to a client before starting the replication, reducing the critical path for the client's response into a single node.

This example would allow us to have writes performed without blocking while propagating write operations. However, there is a lot more to external replication that implies certain overheads, such as a server being required to keep track of all the currently active servers, and either maintain a persistent connection for each server, or open shortly-lived connections according to the replication protocol needs.

Furthermore, it provides the oportunity 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.

This kind of replication entails additional message passing through the network, where one should strive to achieve minimal data movement, on the other hand, our item selection process revolves around bursts with fast shrinking, which requires each server to keep track of who owns a given item, and send notifications for decreasing the replica count for that item.

It also implies that the clients must acquire some form of leases during a write operation over a replicated item, to avoid fetching inconsistent states. This lease can be

discarded after the write is completed.

When considering external replication, sampling is advised, since the items being replicated are the most popular in that partition. Therefore, there is no need for the other servers to give feedback on the item, where the growth and shrinking of each item is regulated based solely on the home server. This would provide minimal data transfer, but incurrs on extra overhead, where the worst is the penalty of persistent loss of a portion of memory for sampling.

3.3.3.2 Internal Replication

This is an unusual type of replication since in most systems, where there is no notion of partitions, performing in-server replication seems counter-intuitive and requires renaming the items and keeping track of the originals.

This is not our case, we can simply distinguish the replicas by the partition they are placed on, and freely access them since in its core we are still dealing with shared-memory.

Implementing any replication protocol in this environment seems trivial, since it should allow a large number of simplifications.

However, the data item locks impose restrictions yet again, since there is the possibility of hash collisions in and between sequences of lock acquisitions, even though the probability of one of this occurrences is extremely slim, it would still result in a deadlock.

Hence we resume the implementation of a replication protocol using message passing through operative system pipes for inter thread communication, while following the lines of the original algorithm, where a thread is only allowed to hold locks from the partitions it owns, or a single lock at any instant.

Since latency is a prime concern, we discarded strong consistency algorithms which are known to be very time consuming, and focused on causal consistency, in particular Chain-Reaction, that allows progress without blocking while performing an update.

The principles of fast growth and shrinking discussed before are still applied, where the overall amount of replicated items that survives the shrink phase will tend to be very low, as well as the number of replicas for each data item.

3.3.4 Indexing Replicas

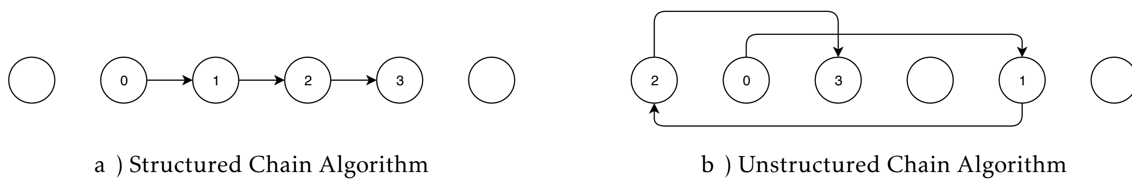
Server driven replication requires each server, in some sense, to also behave as a client, where it has to index the target partition for a given key to be written in a deterministic fashion, where clients will also have to implement this behavior.

Since there is an immutable amount of partitions in a server, it becomes easy to index a partition by using the modulus over a hash function with the number of partitions.

However, since we are applying replication, some changes have to be made in the indexation to pick more than one target partition for the same item.

Employing any structured algorithm to define the chain, such as ordering the ports of the server and applying a shift in the interval $[0,\gamma]$ beginning in the owner port, might not be the best choice.

Structured algorithms will lead to all the replicated items from an overloaded partition to target the same sequence of partitions. This has severe implications, such as probably overloading the next partition, by flushing the same amount of items that are being replicated from a single partition, and in the worst case scenario the targeted partition might reside in the same overloaded CPU, resulting in a total waste of memory with no benefit (the impact would be similar if performing external replication).



a) Structured Chain Algorithm b) Unstructured Chain Algorithm

Figure 3.5: Correlation between sockets and partitions with exclusive access

We use the same concept of Spore, unstructured replication, by performing the modulus with the number of partitions over the hash function of the item key concatenated with a value in the interval $[0,\gamma]$, ensuring that the access to replicated data items can be balanced by varying the chosen value.

This method has a very high probability of scattering replicated items across the partitions. However, this also implies that some items might target partitions that already have those items (including the owner partition) and partitions residing in the same overloaded CPU.

In the case a replicated item targets a partition which already has that item, the λ value is maintained, but the write is discarded, resulting in the same state as if that replica did not exist, only reducing the amount of load distribution for that item.

In the second case where it targets a partition in the overloaded CPU there is not an actual problem, since the other replicated items will relieve the CPU and it will behave normally.

This unstructured replication reaps a crucial benefit, not flushing a large set of items from a single partition, it flushes small sets of items across all partitions, leading to the least increase in the miss ratio, since the items flushed are the lowest access frequency items of the server.

3.3.5 Summary

Our solution performs a memory split, proving exclusive access to each partition through a socket that is owned by a worker thread. Each worker thread is bound to a CPU Core and might have assigned one or more partition, each partition is self-contained having its own LRU's, and a Hashmap to perform item indexation.

Having exclusive access to partitions will eventually cause at least one of the CPU Cores to become overloaded. For such cases, there is a rebalancing mechanism in place with two algorithms: Socket Shift, and Selective Internal Replication. These might become complementary between iterations of overload triggers.

The Socket Shift rearranges partitions across the CPU Cores, while Selective Internal Replication, performs a more fine grain load relief controlled bursts of replication, with a fast shrink pace based on the popularity of data items. The replication scatters the replicated data items across all the partitions.

This method allows a server to keep working at full capacity, for its intended purpose, storing the most popular data items, while providing more availability than Memcached. Considering that instead of keeping all the data items while struggling in overload as Memcached does, it grants more privileges to the popular items through internal replication, while inherently discarding the least popular, achieving the best outcome for the limited space constraint.

3.4 Client Architecture

Conventional indexing schemes typically display the servers in a flat structure, where each target is at the same level as the others. However, due to internal replication, this is not the case there is a need to distinguish between server and ports, forming a two-tier indexing scheme.

The first data placement algorithm should decide which server to place them on, and the second should determine what port to use.

The two algorithms follow different assumptions. The first algorithm has to deal with dynamic memberships, where servers might join or leave. The second algorithm only needs to take into account the number of ports open in the chosen server. The number of ports is a fixed and either all or none fails, where the first case is interpreted as a server failure.

Both algorithms should strive for near-optimal data placement algorithm with low overhead able to scale, since it provides close to the equal probability of a server owning a data item, which reflects the maximum usability of the server pool. We use Random Slicing as our use case to index the servers, since it 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 or Hash function to select a target server.

For performing the port selection we simply mimic the server replication indexation scheme.

3.4.1 Global Access Frequency Imbalance Management

To prevent additional load on caching servers, typically introduced by very high access frequency data items, a Two-Tier caching solution is typically in place. In this scenario, most of the very high access frequency data items will appear to client as high access frequency data items in comparison to all the other data items, as such, storing them in a local client-side cache for the most accessed data items implicitly reduces the accesses to the caching servers. However, this local-cache will demand low TTL to avoid incoherence.

The updates to the local cache should be done asynchronously, since it is irrelevant if some of the updates are delayed or fail, because the only interest resides in the data items that show a very high access frequency compared with others in a local context.

3.4.2 Random Slicing Implementation

The Random Slicing algorithm is a representation of a line segment of the interval [0,1], which in itself is composed by a set of segments, each representing either a portion or the full quota of the assigned space for a server, according to its weight relative to the others.

At top level this line segment is represented by a View which is composed by two main lists: the *Servers* linked list, and the *Strip* which is comprised of buckets forming a doubly linked list.

The Sever list adds new elements to its head, and the removes are performed by IP address match.

The Strip is composed by an ordered set of buckets, each with an interval, where the total sum of all intervals quotas amounts to 1. The first bucket starts at 0, the last ends at 1, and all the intermediate buckets begin where the previous ended.

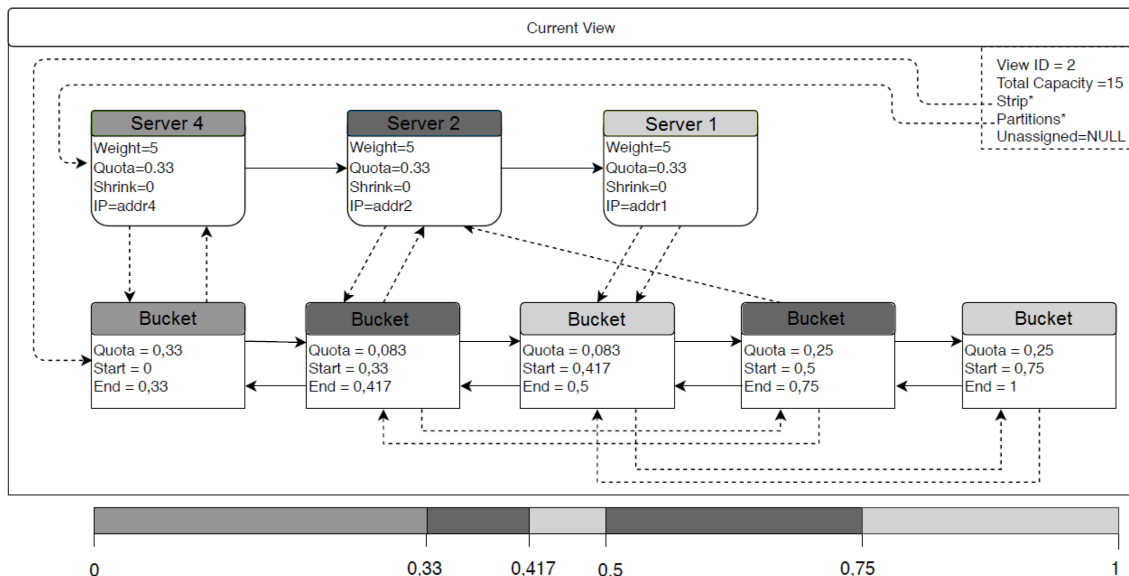


Figure 3.6: Random Slicing Structure

In its initial state, there is only one bucket that condenses the interval of the full strip ([0,1]) as unassigned.

As servers are added and removed, either split or gathers are performed in the buckets, where each bucket references its owner server. Each server only references the first owned bucket, since all his buckets are linked between themselves forming a doubly linked list of owned buckets which is intertwined with the strip.

For the gap collection algorithm, we use CutShift, as recommended by the authors. The remove and add operations if performed in batch reduce the fragmentation, and allow a more efficient gap collection.

3.4.2.1 Lookup Improvements

To perform a lookup in the Strip, a PRNG function is needed to generate a value between [0,1], the same can be accomplished by using the result of a hash function divided by its maximum value while maintaining the same properties.

The strip is iterated until a bucket has the interval that contains the respective value, from which its owner is the target server.

To achieve a time complexity of O(1) instead of O(n) while performing the lookup, an array was added, which indexes constant offsets, forming a Skipped List over the Strip, where each array position reference the bucket that contains its offset. If the number of buckets is N times greater than the array it performs a re-size, followed by a reference update due to the new offsets.

3.4.2.2 View Computation and Coordination

The View computation is a deterministic algorithm, assuming all clients process all the transformations in the same order. Where each client can compute a new server view by performing a transformation on the previous, as such, for a new client to compute the current view, it would have to compute all the previous transformations starting in the initial view.

To avoid this expensive bootstrap process, the current View can be stored in a shared location with its respective sequence number, which allows new clients to fetch the pre-computed an up-to-date view. However, to store the current view, only one client is required to compute it, since all the other clients can fetch the newly computed view, instead of doing it themselves.

To achieve this, a leader election process is required, hence, the use of Zookeeper, which provides the creation of ephemeral nodes, where the node only exists while a connection is active, and also different types of semantics for node creation, where a node can be created only if it does not exist already.

Furthermore, to compute a new view, a client must maintain watch over all the currently registered servers, in order to detect changes in the membership. Zookeeper also provides a functionality for this, where a client is able to watch over a node and receive a

push notification if any changes are performed on the node or any of its children (registered servers).

Nevertheless, if only the client leader is keeping watch over the servers, only he is able to detect the changes. When it loses its connection or fails, a new leader election process will sprout, resulting in a new leader. This new leader will have to scan through the servers in the view and compare with the currently active servers to detect if any change in the membership occurred during the leader election process.

To avoid the scan over the view and all active servers, all or subset of clients can also keep watch over the active servers, and keep track of the few last changes in the membership. Where in the case of the subset, only those can run for leader election.

3.5 Summary

In this chapter we described typical deployment environments for caching systems, core aspects, and designs. This was followed by the structure of our solution, discussing how to perform memory partitioning and how to expose the partition for client accesses.

We analyzed the developed algorithms and methodologies applied to mitigate the challenges that arise from the partition scheme under stress scenarios. In particular, the Socket Shift is a coarse load relief method that redistributes the load among the worker threads, and the Selective Replication is a more fine grain method, which discards data items with low popularity to provide more availability to popular data items.

Finally, we provided an additional module for the client to index the partitions and a low data misplacement algorithm that strives to achieve low implicit miss ratio.

The following chapter presents the core aspects of our prototype in detail, in particular we describe the changes performed over the original Memcached and the aspects to consider when implementing memory partition, Socket Shift, and Selective Replication, which lie at the core of our solution.

Chapter 4

Implementation Details

Here we will discuss, in some detail, some of the relevant aspects of our implementation that strive to achieve an optimized solution, considering space and complexity of operations.

All operations in Memcached can either read or change an item state, as such, we focus on set and get operation, since all the others are fundamentally similar, and can be handled using similar techniques. Most of the presentation in this chapter is performed at a high level, with details being presented when required to fully grasp the reasons behind our implementation decisions.

All the solutions presented strive to use the least amount of resources possible, while reusing the already existing structures when possible, to avoid increasing overheads, and maintain O(1) time complexity in most operations.

4.1 Internal Network Model

Memcached represents all possible connections using an array (conn* conns) with the maximum number of connections delimited by an adjustable threshold. This array stores all the meta-data for each file descriptor, which can be mapped by its integer value to the respective index of the array.

In order to monitor the active connections, it resorts to libevent (event notification library) for managing file descriptors in an event-driven model, it provides a generic and portable interface that automatically leverages on the most perfoment library available on the host, such as select, poll, or epoll.

Libevent executes a callback function when a specific event occurs in a monitored file descriptor. These callback functions are provided for each file descriptor when added to the libevent-base for monitoring. The libevent-base consists of a control loop that iterates

over all triggered events, executing the respective callbacks or that waits until an event is triggered.

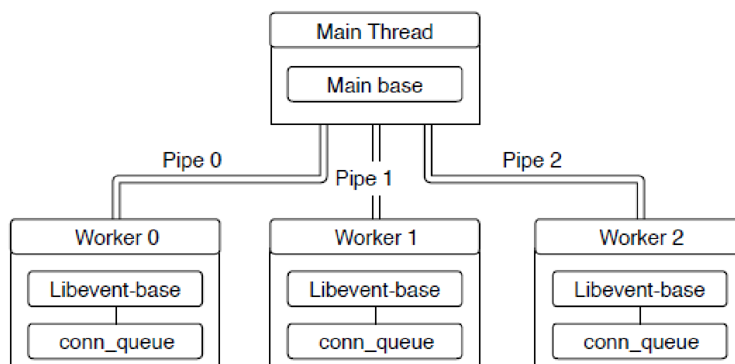


Figure 4.1: Network model

The main process which performs the bootstrap phase in Memcached, launches all the worker threads, one for each CPU core is recommended for maximum performance. The main thread, and all the workers, each have their own libevent-base, henceforward, the libevent-base of the main process will be referred as the main-base. During the bootstrap phase, the main process while creating all the workers, also creates read and write pipes which are identified by a file descriptor, and a connection queue of items (cq_item) for each of the workers, as depicted in Fig.4.1. This pipe and queue structures allow the main process to send content to the workers, by pushing a connection item into the worker queue and triggering an event trough a write operation in its respective pipe, which will signal that a connection item is ready to be poped out of its queue for handling.

The main-base is responsible by only two events: Clock update and Main socket triggers.

The clock update is one of the most important functions which will perform a system call to get the current time and write it to a global variable. This is a periodic event triggered every second, which provides enough accuracy if all the other triggered events on the main-base do not exceed one second to process, otherwise, it will delay the clock. This greatly affects the system performance, where otherwise, the system would require a system call for each time a timestamp is needed. Notice that the clock value is used extensively. As an example, the crawler uses it with a very high frequency while checking the expiration date and the maximum age of the tail items in the internal queues.

The main socket is the ingress point of all new connections in the system, and where the accept system call is performed, hence filling all the meta-data of the respective (spawned) connection. After the accept is performed, the file descriptor is forwarded to a worker thread through a pipe, with a round-robin policy, to be added to the respective libevent-base, and become functional.

The libevent-base in each thread is responsible for monitoring the pipe created by the main process and all the file descriptors of the accepted connections that are assigned to

it by the main thread. Afterwards, any incoming requests from a connection will only be processed by its monitoring thread.

4.1.1 Exclusive Partition Access

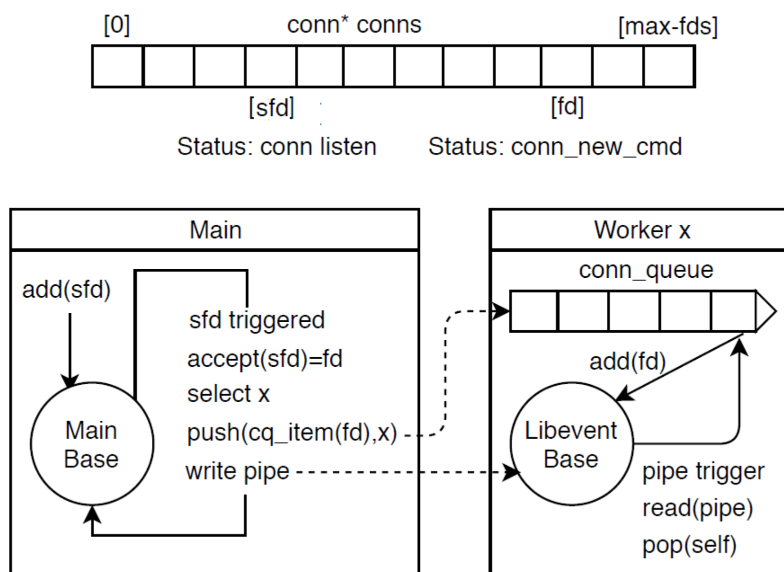


Figure 4.2: Accept New Connections

Achieving exclusive access to a single partition is fairly simple since there is an already existing pipe structure for each worker that is associated with the Main process. Taking advantage of this pipe structure, the Main process in its bootstrap phase creates N sockets (master sockets) with the same configuration as the original Main socket, and distributes them to each worker in a Round-robin fashion.

Each worker thread is bound to a specific core, where its core mask matches the modulus of the worker thread identifier(i.e., thread id) over the total amount of available cores. The worker thread id is a sequential value.

The Main process keeps track of master sockets file descriptors using an array, where each array index position directly maps to the index of the respective partition. This index value is also stored within its respective connection item in the global file descriptor array. The stored partition index value within the connection item is meant to provide the index directly when an event is triggered (it only provides the file descriptor) instead of searching the master socket array for the file descriptor and fetching its index.

Each worker upon receiving a push notification will add the respective master socket to its libevent-base, as illustrated in Fig.4.2. When a new connection is accepted, it is the result of the triggered event on the libevent-base where the master socket resides, hence this new connection will inherit the respective master socket partition index and will be added to the same libevent-base.

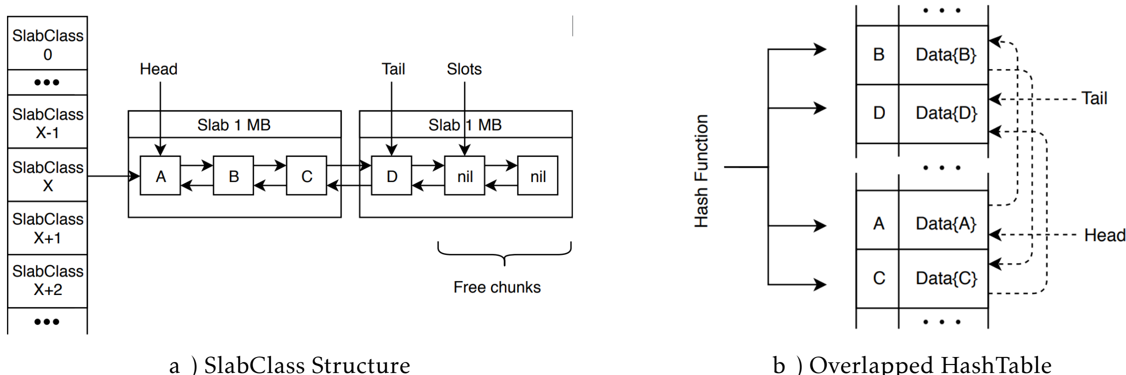
This implies that only the respective thread where the master socket resides is able to access the requests from these connections, that carry request to its local memory partition.

4.2 Memory Partitioning

Our exclusive access policy is driven by the network model, which already provides an abstraction for the memory partitions. As such we identify each partition by the master socket creation order, that maps directly to its index, uniquely identifying each partition and any related component. Since each new connection inherits the associated master socket partition index, they remain within the same access scope.

4.2.1 Slab Memory

The memory management is based on the concept of Slab Classes, an incremental ruler that defines the maximum size of an item it may hold. The memory allocation process revolves around fixed allocation of 1MB memory pages, represented by a Slab, where each Slab is split into several chunks according to the destination Slab Class size. Each Slab Class has its own embedded LRU that dictates which items should be evicted, if page allocation is not possible.



a) SlabClass Structure

b) Overlapped HashTable

Figure 4.3: Memcached Memory Structure

If an allocated Slab is not under any specific Slab Class, it will be placed in Slab Class 0, that represents the Global Free pages, which will replace the next page allocation, and afterwards be split according to the assigned Slab Class.

The Slab allocation creates several equal-sized chunks, even if only one is needed. This avoids the next few allocations for that data item size with minimal memory fragmentation, and also enabling Slab migration between Slab Classes .

This memory displacement scheme is very efficient in terms of memory usage. Still, in isolation, it does not provide a fast lookup mechanism, since it would require traversing

all data items. As such, an HashTable is in place, to provide a fast lookup, over all data items.

Due to this structure, all operations are conditioned by a set of locks: (1) the slabs locks, and (2) the items locks. The Slabs locks are in place over each Slab Class, while items locks are placed on each individual item.

The partitioning scheme is achieved by splitting the overall memory limit over the number of available sockets, where the last partition accumulates both its share, and the remaining memory, if any. Inside each partition, the memory keeps the same original structure, and allocation policies.

This implies that the amount of Slab Classes increases by a factor of N partitions, and so do the Slab locks. This provides a more fine-grain locking policy, which has a great impact over the write operations since when performed, it requires a Slab lock to be acquired to bump an item to the head of the queue.

Furthermore, intuition says that isolated memory segments that are exclusively accessed by a single thread, should not require locking. Still, this is not the case due to the interference of the LRU Threads, that are in place due to scalability issues according to the authors of Memcached.

4.2.1.1 LRU Threads

The LRU Maintainer and LRU Crawler Threads are the main mechanisms that allow Memcached to scale in terms of worker threads, hence, we maintain their internal logic, only multiplexing it over all the partitions, instead of having two threads dedicated for each partition. This required some tweaks on coefficients that parameterize the behaviours of these threads, variables, and minor changes to the asynchronous bumping queues.

The LRU Maintainer Thread is required to perform more peeks to the tails of LRUs, and deal with the asynchronous bumps of all the partitions, which requires it to maintain track of the partition for which each bumped item belongs to, by keeping the them in separate queues.

The LRU Crawler Thread has to create more special crawler items to insert in the LRU's, since there are more LRU's to deal with, but since the amount of items in the server is still the same, the total amount of iterations over the items, and time spent per operation remains the same.

4.2.2 Item Access

The direct mapping from socket to partition provides a foundation to perform a memory split. However, it does not control item reachability, since assigning a socket to a specific thread does not imply exclusive access to a partition. This is achieved by restricting the socket accesses through a single Hash map per partition, instead of a shared hash map for all the data items.

Even though the hashmaps restrict the memory access from the client side, the items are not entirely isolated, and still suffer interference from the LRU threads. Thus, item locking is still required. The locking is external to the hashmap, and is performed before each item access through the use of a hash function based on the data item key.

This locking method works on a global scope, which allows interferences between partitions in the case duplicate items in different partitions. To limit the locking scope we could append the partition index to the key, to be processed by the Hash function.

The append operation would have to be performed for every request. Hence, its cumulative effect would probably impact the server latency.

As such, we do not perform an append, but instead slightly change the hash function to take in another parameter, the partition index (salt), and process it after the key, as if it were appended, maintaining its original desired properties (e.g., avalanche effect). This allows the same key to acquire different locks according to the partition where it resides.

These changes were applied to the murmur3 (default) and Jenkins hashing algorithms present in Memcached. We present, as an example, in Listing. 4.1 the same kind of modification to djb2, which is a simpler and more compact algorithm (This algorithm should not be used since it generates insufficient entropy).

Listing 4.1: djb2 variant

```
1  unsigned long hash(unsigned char *str, unsigned char *salt)
2     {
3         unsigned long hash = 5381;
4         int c;
5
6         /** same effect as prepending the salt value **/
7         while (c = *salt++)
8            hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
9
10        while (c = *str++)
11           hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
12
13        return hash;
14    }
```

4.2.2.1 HashTable Re-size

The HashTable only retains its high lookup performance until an item count threshold is reached. Afterwards, the amount of collisions per bucket increases, and a lot more time is needed to traverse the linked list and find a given item. As such, a resize is in place. During the resize process, no change or lookup should be performed, since it would risk creating a race condition.

In the original Memcached version, to avoid the race conditions, a full stop is issued from the main thread through the underlying pipe structure, and all worker threads have to pause. Meanwhile, the main thread also activates the Maintenance Thread to perform

the HashTable resize, which upon completion, replaces the current HashTable, and allows the worker threads to resume their normal execution.

In our case, since each HashTable is limited to a partition, and each partition is only managed by a single thread at any given time, this problem does not occur. Also, it removed the need for the Maintenance Thread, where instead of the main signalling it to perform the resize, it signals the owner thread of a given partition, if the need for this operation arises.

If a partition has been shifted before the signal has been processed, the mechanism naturally adapts, since the thread will discard the request for a resize for a partition it does not hold, and it will be issued again correctly in the next timed event.

4.3 Load Monitor

This is an entirely new main sub-routine that occurs in a one second interval. This sub-routine should take less than a second to compute, to avoid delaying the global clock update routine, and is responsible for the detection of CPU Core saturation. Once found, it will try to redistribute the current load in an iterative fashion, first by attempting to redistribute the partitions by the worker threads, and if unable to do so, it will resort to selective replication as a mean to distribute some partition load among the others.

Fetching the load of each CPU Core (described by the Equation 4.1, and in greater detail for UNIX systems in Annex I) only provides a load estimate over the last second, which is too sensible to load peeks, as such we use the Exponential Moving Average (EMA) over a number of readings.

$$CPU_load = \frac{\Delta user + \Delta nice + \Delta system}{\Delta user + \Delta nice + \Delta system + \Delta idle} \tag{4.1}$$

The EMA is based on a set of readings in a time frame (window), which defines its multiplier, as seen in equation 4.2. The multiplier gives a higher weight (significance) to the last readings, relative to the previous cumulative readings of the window.

$$Multiplier = (2/(window - 1)) \tag{4.2}$$

Typically the initialization of the EMA requires a Simple Moving Average to be computed over an initial window, where the weight is the same for all the readings. However, we do not apply this method, we assume a value of 0 for the initial input, which mimics the initial state of a stale server. This change only impacts the first few readings, where the result of the EMA will not reflect the actual system load.

Not using the initial SMA avoids the memory allocation, and storage of all the readings for each initial window. Hence, the new EMA can simply be computed by using the previous EMA value, as described in Equation 4.3.

$$EMA = (Value - EMA_{previous}) * Multiplier + EMA_{previous} \tag{4.3}$$

The EMA is also applied over the number of operations performed in each partition. The internal statistics maintained internally, provides a foundation to retrieve this value, since it keeps track of the amount of operations performed, by operation type. As such we perform an aggregations over these values.

The Monitor sub-routine, described in Listing 4.2 is responsible for updating the EMA value of each CPU and partitions. If any of the CPU Cores has reached a given threshold, and is experiencing higher load than the others, it will try to provide the relief method that best fits the system needs, and update the time penalties. Since each relief method after triggered requires a time penalty superior to the EMA window for the system to readjust, and avoid a rebalance trigger to be performed needlessly.

Listing 4.2: Monitor Main Trigger

```
1  Global State:
2    EMA_cpu // array for storing the current EMA over the load of each cpu.
3    last_rebal_wait // value of the last wait time
4    rebalance_wait // current value of the time penalty
5    shift_threshold // load which is worth to try to optimize
6    rep_threshold // reaching high load levels that might require a fine-grain method.
7
8  Local Initial State:
9    try_socket_shitf ← false
10   try_replication ← false
11
12 Upon time trigger do:
13   /** Check for load symptoms **/
14   foreach cpu in cpus{
15     if(EMA_cpu[cpu]>shift_threshold && previous_shift_finished()){
16       try_socket_shitf ← true;
17     }
18     if(EMA_cpu[cpu]>rep_threshold && has_available_space()){
19       try_replication ← true;
20     }
21   }
22
23   /** try to perform rebalance and adjust waiting time penalty **/
24   if(rebalance_wait= 0){
25     done_rebal ← process_rebalance(try_socket_shitf, try_replication);
26       if(done_rebal){
27           rebal_last_wait ← window+1;
28       }else{
29           if(rebal_last_wait<60)
30             rebal_last_wait ← rebal_last_wait*1.5;
31       }
32       rebalance_wait ← rebal_last_wait;
33   }
34
35   /** Update wait time penalty **/
36   if(rebalance_wait >0){
```

```
37    rebalance_wait--;
38  }
```

When dealing with the CPU load, the amount of accesses of a partition might not translate the actual load it is subject to, since the amount of operations performed is restricted by the amount of operations the CPU is able to handle, not accounting for the increasing amount of incoming requests that are waiting in the receiver queues, which will have to be dealt with by the next CPU Core.

Due to this phenomena, what we achieve is an extrapolation of the load, based on the relative amount of requests processed between partitions owned by a CPU. This defines a lower bound on the real load a partition is experiencing.

The currently emplayed Decision Model 4.3, only contemplates two relief methods, the (1) Socket Shift, and the (2) Internal Replication. The Socket Shift, will move partitions between threads striving to achieve a good distribution of load over all the CPU cores. This is a coarse method for load balancing, that has very few downsides, and can be improved by increasing the amount of partitions. On the other side, Internal Replication, will provide a more fine-grain relief method (at data-item level), but at the expense of additional memory usage.

When there is a trigger for rebalance, the first step is to have a clear picture of how far the load is from a uniform distribution. This is achieved by the current standard deviation of all accesses to the threads.

Afterwards, if the socket shift option is valid, it will run a simulation (e.g., Makespan algorithm) of the partition relocation. If the relocation achieves a better standard deviation than the previous, the Socket Shift algorithm will execute the change reballancing.

Independently, if a Socket Shift has happened, or if there is no viable rearrangement of partitions. If the Replication option is valid, it will take the achieved minimum standard deviation, and test if the load is within the desired levels of uniformity (regulated by a slack parameter), if not a Selective Internal Replication process will start, striving to achieve it.

Listing 4.3: Decision Model

```
1   Global State:
2     EMA_ops // array of structs for storing the current EMA value of each cpu.
3     curr_dist // array that stores to the sum of the EMA_ops of the partitions owned
4     by a thread.
5
6   struct EMA_ops:
7     value // current EMA value of a given partition
8     curr // current owner of the partition
9     new // new owner of the partition
10    socket // socket id value
11
12  boolean process_rebalance(try_socket_shitf, try_replication){
13    done_rebal ← false
```

```
14    curr_dist ← compute_dist(on_curr) // computed based on EMA_ops.curr
15    mean ← get_mean(curr_dist,n_threads)
16    curr_stDev ← stDev(curr_dist,n_threads)
17    new_stDev ← MAX_FLT
18
19    if(try_socket_shitf){
20      makespan() //changes the value of EMA_ops.new
21      new_dist ← compute_dist(on_new) // computed based on EMA_ops.new
22      new_stDev ← stDev(new_dist,n_threads)
23
24      if(new_stDev<curr_stDev){
25        foreach p in EMA_ops{
26          if(p.new != p.curr){
27            Call socket_shift(p.curr,p.new,p.socket, is_master)
28            p.curr ← p.new
29          }
30        }
31      done_rebal ← true;
32      }
33    }
34
35    if(try_replication){
36      if(mean*load_tolerance < min(curr_stDev,new_stDev)){
37        partition ← get_max_partition_in_max_thread()
38        Call Replication(partition)
39        done_rebal ← true
40      }
41    }
42  return done_rebal
43  }
```

The Selective Internal Replication is performed on the highest accessed partition in the CPU with the highest saturation to deal with the excess load. This form of replication does not occur indefinitely, since in the worst case of mass evictions would sprout, and as such there are restrictions to limit how much a saturated server can adapt.

4.3.1 Internal Full Mesh Network

Both relief methods require inter-thread communication, since each worker thread should be responsible for its own resources.

A simple and effective solution would be to use the currently in place client interfaces for each partition, where the messages sent would only go through the loopback interface, yet we would have to open a set of connections to those sockets.

These connection could be persistent or short-lived, in the case of short lived would require a connection establishment for a set of messages to be transmitted, followed by closing the connection.

In the case of persistent connections, we can employ two distinct strategies to have a set of sockets perform write operations shared among the whole system, while being protected by a lock, to ensure mutual exclusion; or each thread can have its own set of sockets to perform the writes.

Either-way, in any of these cases we are increasing further the amount of open file descriptors, in the best case, linearly. This has a negative impact in the system performance.

Or solution does not open new connections, it simply re-uses the current in place operative system pipes structure, where each thread has its own pipe to write, and read from. The same considerations as above are present in this scenario, where the pipes can be shared and accessed from any thread to perform a write.

However, pipes have a peculiar feature, according to POSIX.1, the write of less than $PIPE_BUF$ bytes must be atomic, where POSIX.1 requires $PIPE_BUF$ to be at least 512 bytes. As such, we can avoid locks by performing write operations with lengths below the $PIPE_BUF$'s lower bound.

The read from pipe operation performed from each worker thread are batched, where a read will try to fetch a large buffer from the pipe, instead of fetching a single $Frame$ of data. This avoids very large pipe buffers, that can be occur due to replication bursts behaviour.

As a final remark, the internal full mesh network was already in place from the very beginning, the only additional consideration is how to perform the writes to preserve atomicity.

4.4 Socket Shift

The Socket Shift algorithm strives for a low increase in meta-data while making a smooth connection transfer, without large amounts of connections being transfered all at once, and forcing threads to stop while performing several adds and removes of the respective libevent bases.

As such, the shift only occurs directly over a master socket, all its children will act accordingly upon answering client requests. All the interactions are coordinated through the control plane that is composed of four components:

(1) The master sockets indexes, matching the partition index.

(2) An array of Master sockets that stores each file descriptor.

(3) CPU masks that match the respective thread ids.

(4) The partition array that stores the current owner CPU mask/thread id.

The abstraction of the control plane that is provided, diverges from the actual implementation for simplicity, but the memory accesses in this process has O(1) time complexity with reduced meta-data, where the master socket index is actually stored inside

the connection items on all sockets, to avoid array iterations. The only components that match the implementation are (3-4) in Figure 4.4 that provides the ownership storage with direct access.

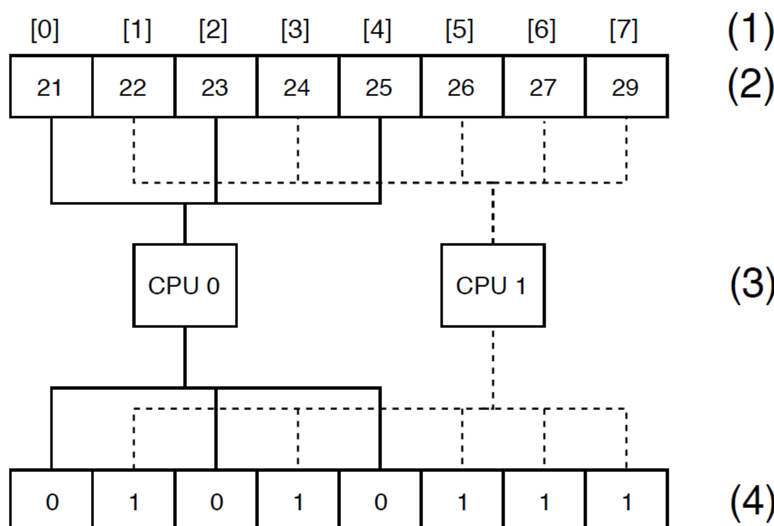


Figure 4.4: Logical view of the socket shift control plane

The socket shift process uses the underlying network model to smoothly transfer the partitions across threads, and is composed by two phases: (1) master socket migration, and (2) self-migration.

The first phase, is coordinated by the main process, which indicates which socket is to be moved from the owner CPU to the targeted CPU, since each CPU can only host one thread, and each thread is bound to a CPU, it implies a direct correlation between them, where referring either produces the same outcome.

The information for this transition is passed down from the main to the thread that currently owns the partition, and afterwards to the target thread. This is achieved through message passing acording to the listing 4.4 using the *Frame* structure represented in Figure 4.5.

The actual transition accurs when the source thread is signaled to remove the file descriptor from its libevent-base, and concede the ownership to the target thread by placing the target thread id in the respective partition index on the partition array.

Afterwards, when the source thread has forwarded the frame to the target thread, it will acquire the socket, by adding it to its libevent-base, and resuming normal execution.

The second phase, starts in the source thread context, before each client request is processed, by performing a lookup on the partition array, based on the inherited partition index.

If this value does not match the current thread id, the thread will remove the file descriptor from its own libevent base, create a *Frame* with the target thread as its rightful owner, and send it to the target thread, with a flag as *pending*.

The target will behave similarly to the first case, only with a slight change, it will fist add the file descriptor to its libevent-base, and only afterwards fulfill the pending request.

Answering the pending request must be the last task, due to *closeconnection* requests, where we can't add the file descriptor of a connection that no longer exists to the libevent-base.

1 Byte	2 Bytes	2 Bytes	4 Bytes
Flag	Source CPU	Target CPU	File Descriptor

Figure 4.5: Socket Shift Frame

Furthermore, the means for answering a client request are stored in the global connections array, that can be accessed by using the file descriptor as the index from anywhere in the system.

Listing 4.4: Socket Shift algorithm

```
1  //frame : message that is passed down along the threads
2
3  on monitor:
4          frame.flag ← shift_socket
5      frame.source ← swamped CPU core id
6      frame.fd ← master socket file descriptor with lowest accesses in
7      frame.target ← CPU core id with lowest load
8      source.pipe.write(frame)
9
10 on shift socket trigger:
11     frame ← self.pipe.read()
12     self.event_base.remove(frame.fd)
13     partitions[conn[frame.fd].index] ← frame.target
14     frame.flag ← acquire_socket
15     frame.target.pipe.write(frame,none)
16
17 on process request:
18     target ← m_partitions[conn[fd].index]
19     if(self.thread_id != target){
20             frame.flag ← acquire_socket
21         frame.fd ← fd
22         frame.target ← target
23         self.event_base.remove(fd)
24         target.pipe.write(frame,pending)
25     }
26
27 on acquire socket trigger:
28     self.event_base.add(frame.fd)
29     if(pending)
30         process_request(frame.fd)
31
```

```
32 Note: The triggers have a prior validation that the received frame is either in the right
33 thread, if not the message will be redirected to the correct thread.
```

It is crucial that the initial validation of the second phase (self-migration) is extremely fast and simple, since it is performed in every request, and any unnecessary overhead will have a cumulative negative impact on performance.

4.5 Selective Internal Replication

The Selective Internal Replication (SIR) is an overload relief mechanism that attempts to flatten skewed partition accesses, and is regulated by configurable soft and hard limits and burst sizes. Unlike most forms of replication, it is an iterative algorithm for in server replication, that is not applied to all data items, it does not have a fixed replica count, and is performed in bursts.

It strives to take advantage of the already in place eviction policy in the caching systems. In this case, the segmented LRU algorithm, designed to be resistant to scanning workloads, that provides a mechanism to protect the most popular items by default, which is accomplished by the three queues and the interactions between them.

The HOT queue is the ingress point on the segmented LRU, and only a transition state (there are no bumps within this queue) before an item is sent either to the COLD or WARM queues. So the HOT queue only contains the most recent items, which by itself only allows to distinguish if items are popular during the short amount of time they have resided in the cache. The COLD queue is the only egress point of the segmented LRU,

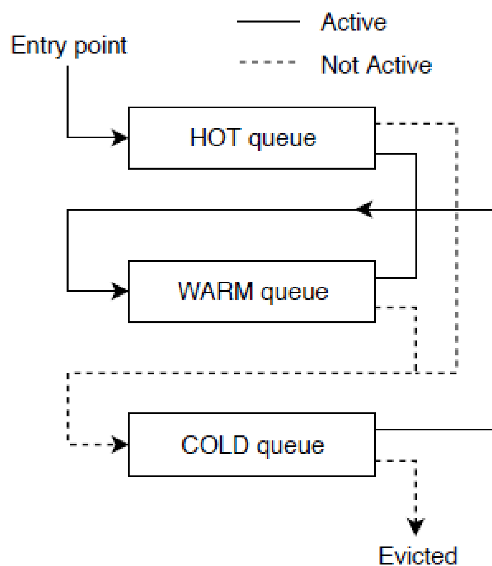


Figure 4.6: Segmented LRU

providing priority for eviction among the lowest accessed items, which does not provide any useful information on the most popular items.

The WARM queue keeps the popular elements, since if items are deemed relevant either in the HOT or COLD queues, they will be placed in the WARM queue, which maintains the most popular items towards the head of queue.

These three queue greatly vary in size, where the HOT and WARM queues are limited primarily by a percentage of memory used, typically the the HOT queue is small, and the WARM slightly larger, while the COLD queue takes the most space, and has no restrictions.

Since probabilistically the most popular items in each Slab Class reside in the head of the WARM queues, we make sudden busts of replication by selecting the first K data items on the heads of the WARM queues of the partition with the highest amount of accesses in the CPU subject to the highest load.

The target partition is chosen at data item level based on a scatter function, in this case, the murmur3 variant taking γ as the salt value (see section 4.2.2), and computing the modulus over the total number of partitions.

Since the burst selection is based on the WARM queues of a given partition, then having a higher number of partitions will provide a more refined replication process, which would increase popular data item availability.

However, not restricting the replication, in cases where the most partitions are saturated, could cause a cascade effect, where the replicated data items from one partition would trigger other partitions to perform replication, and so on, resulting in a uncontrolled flush of data items.

4.5.1 Burst Control

Understanding the impact of the replicas on the system, and in each partition, without a sizable increase in meta-data is not a trivial accomplishment. As such, we do not try to keep track of every replicated data item, instead we impose, and keep track of three configurable limits on replication:

Hard limit percentage of the space used in replication, relative to the total amount of reserved memory.

Soft limit percentage of the WARM queues used in the selection on each saturation trigger.

γ **limit** restricts the amount of replicas of a single data item.

When a saturation threshold is triggered on a partition, its Slab Classes are traversed, from the smallest to the largest, applying the Soft limit. Since there is no correlation between data item size and popularity, as such, allowing the smaller items to take priority in replication provides a higher probability of reducing the saturation, with the least memory waste.

Regulating the Soft limit, will allow a more aggressive or more passive burst. Lower values will tend to achieve higher γ values, allowing truly popular data items to become more replicated, while a higher value provide a faster partition relief with a higher number of data item flushes.

However, when close to reaching the the Hard limit, the fairness between Slab Classes inside a partition is compromised, such as, in cases where the amount of space available is not enough to traverse all Slab Classes. For this particular scenario, we keep track of the last Slab Class subjected to replication on each partition, so that the next iteration can start from where the last has left. Allowing the tracker to be reset between iterations would result in starvation of the larger data items for a chance to be replicated.

4.5.2 Shrink

The shrinking of the γ is not a forced procedure, it is inferred based on the the normal behavior of accesses to replicated data items. If a data item is not able to maintain its current status (remain in the WARM queue) after a burst, then it is not an adequate fit for their current γ value. As such we apply two restrictions:

(1) Reduction If a replicated data item is intended to drop to the COLD queue, a request for γ reduction is issued to the data item home partition.

(2) Bumping If a replicated data item has a γ value greater than 0, instead of dropping from the WARM queue, it will be bumped to the head.

The fist restriction, provides the fast shrinking based on the premise that most items residing in the WARM queue should not be able to handle its popularity being cut by $1/\gamma$, while maintaining their current status. The request to decrease the γ value are issued with an absolute value to avoid cases where several partitions simultaneously issue the same request and remove several replicas at once.

The second, is a safety restriction that avoids data items from being discarded from a partition while being replicated. Otherwise, the replicated data items might drop from its original WARM queue due to its popularity being cut by a factor of γ, putting it as a candidate for eviction. If it is evicted, the other replicas will still be active and cause an undefined behavior, as well as increase the miss-ratio until the item is discarded from all partitions.

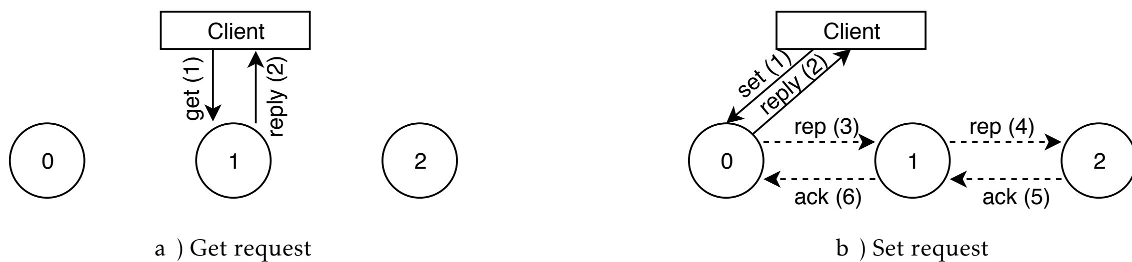
4.5.3 In Memory Replication Protocol

For the replication protocol, we selected *ChainReaction*, which provides causal consistency, since it allows write pipelines, and provides a fast response time, while only being required to interact with a single partition to fetch a data item. This is achieved in write (set) operation by defining our critical path for an update as 1, the home partition of the

data item, and the replicas update as a background propagation, represented in Figure 4.7 -b).

Due to the dynamic replication environment, the replication process is coordinated by the current γ value of each data item, that represents the tail of the chain. The head of the chain has is delimited by the value 0, and the middle by the values between the head and the tail.

To support the operation of the algorithm, each data item stores its current γ (not their relative position in the chain), an update value, and a sequence number, besides the original information. Each of these new fields has a size of 2 bytes.



a) Get request   b) Set request

Figure 4.7: Get/Set operations

When a write is executed, an update chain begins with a sequence number associated with it, the sequence number is an incremental value, that is only altered by the head of the chain, the thread responsible for the data item home partition. As a replication frame, depicted in figure 4.8, is being passed down the chain, at each stop it increments a local update variable on the data items which resumes how many updates are in progress. On the way back, in the *ack* back propagation, this value is decreased, as reference in Listings 4.5 and 4.6.

The read operations are only required to contact a single replica from the replica set as depicted in Figure 4.7 a). This is achieved by selecting a value q between 0 and γ when indexing a data item, which targets one of the partitions that holds a replica.

In the reply its piggy-backed the current γ value that is stored in the data item, and its respective local update value. If the update value is greater than 0, it means that there are updates in progress, which implies that to maintain causal consistency in subsequent accesses, only replicas between 0 and q can be accessed, until the local update value in the next requests is brought back to 0.

1 Byte	2 Bytes	2 Bytes	2 Bytes	2 Bytes	2 Bytes	2 Bytes	2 Bytes	4 Bytes	N < 256 Bytes
Flag	Length	Source Partition	Target Partition	Sequence Number	Hop	Lambda	Old Lambda	Key Size	Key

Figure 4.8: Replication Frame

Contrary to the expectations, the replication frame does not contain the body of the data item, only the means to retrieve it. It is designed this way, to maintain the atomicity

property in the pipe, and to avoid very large *PIPE_BUF*'s. As such, when an update to a replica is performed, the body of the data item is fetched from the previous hop of the chain.

The sequence numbers are in place for two reasons: (1) out of order frame, and (2) chain loops. In the first case, even though the pipe ensures a First-In-First-out (*FIFO*) semantic, due to the socket shift algorithm, where a partition can change its owner thread, it is possible to have out of order frames, since the pipe belongs to threads. This is explicit in the the algorithm as a "redirect for the rightful owner"in Listings 4.5 and 4.6.

The second case, occurs due to the unstructured replication pattern caused by the scatter function, which allows loops to occur in the chain. In this cases, if a partition is visited more than once, it will not perform any operation, only increment the update value of the data item.

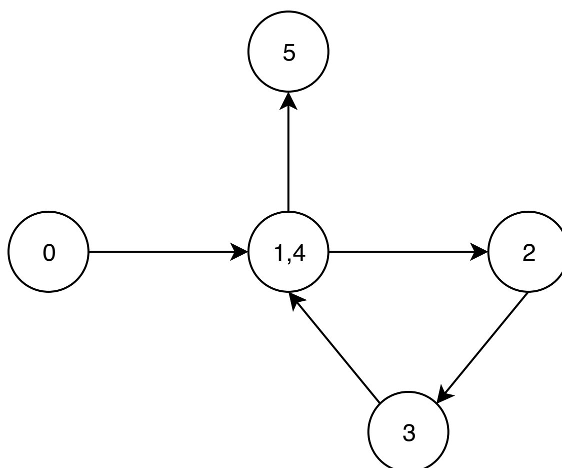


Figure 4.9: Chain Loop

The increase in the update value in the event of a loop is necessary, since the acknowledgements, that decrease the update value, do not have any perception over the sequence number. Hence, they are not able to distinguish between a loop a and a linear chain.

The old lambda value in the frame is in place to allow the frame propagation to reach the tail of the chain. This corner case is also due to the chain loops, where the γ value stored in the data items during the shrink process will not reflect the tail of the chain, since its value as been updated before, as depicted in Figure 4.9.

If a shrink occurs in a scenario without the old_γ value, the hop 1 would update its γ value to 4, and and when the propagation reached the hop 4 it would assume it as being the tail, beginning the ack back propagation, leaving us with an eternal data item (hop 5), that would cause successive shrinks while flooding the pipe of the data item home, since it would be starved from accesses, and constantly trying to drop from its WARM queue.

Listing 4.5: Propagation of replica update

```
1 Global state:
2    used_rep_memory // array with that keeps track of how much bytes each partition
```

```
3                          has in replicated data items
4      m_partitions // array that stores the thread that is managing each partition
5
6   process_replica(frame){
7      owner ← get_partition_owner(frame.target)
8      if (owner != self){
9         owner.pipe.write(frame) //forward the frame to the right owner
10     }
11
12     do_update ← check_sequence_number(frame.target,...)
13     update_value ← get_update_value(frame.target,...)
14
15        if(do_update){
16           if(frame.hop<= frame.lambda){
17              //performs the copy from the data items of the previous hop
18              size ← process_copy_and_update_replica(frame.source,frame.target,...);
19
20              if(size>0){
21                 used_rep_memory[target] += size;
22              else { // enforce removed or expired
23                 used_rep_memory[target] -= expire_data_item(frame.target,...);
24              }
25           }else{ // tail of the previous chain
26              used_rep_memory[target] -= expire_data_item(frame.target,...);
27           }
28        }
29
30        if(frame.hop<frame.lambda && status >= 0){
31              increase_update_value(frame.target); // increment the update variable
32        }
33
34        if(frame.hop<max(frame.lambda,frame.old_lambda)){
35              next_partition ← get_partition(frame.key,frame.hop +1);
36              next_thread ← m_partitions[next_partition];
37              // changes the source, target, and hop fields respectively
38              if(do_update){
39                 frame ← change_fields(frame.target,next_partition, frame.hop+1)
40              }else{
41                 frame ← change_fields(frame.source,next_partition, frame.hop+1)
42              }
43              next_thread.pipe.write(frame)
44        }
45
46        if(hop= lambda){
47              frame ← set_flag_to_ack(frame)
48              previous_partition ← get_partition(frame.key,frame.hop -1);
49              previous_owner ← m_partitions[previous_partition];
50              // changes the source, target, and hop fields respectively
51              frame ← change_fields(frame.target,previous_partition, frame.hop-1)
52              previous_thread.pipe.write(frame)
```

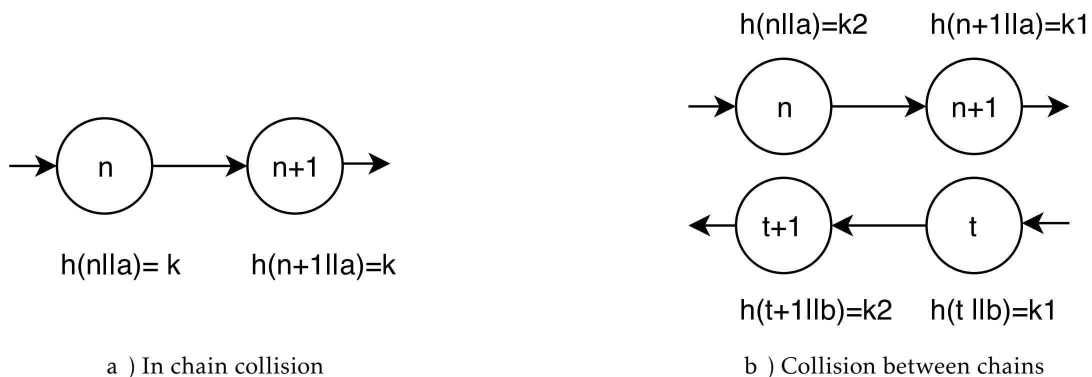
```
53        }
54  }
```

Listing 4.6: Acknowledgement from replication Chain

```
1  Global state:
2     m_partitions // array that stores the thread that is managing each partition
3
4  ack_replica(frame){
5     owner ← get_partition_owner(frame.target)
6     if (owner != self){
7        owner.pipe.write(frame) //forward the frame to the right owner
8     }
9
10    decrease_rep_update( key, nkey,target) // increment the update variable
11
12    if(frame.hop>0){
13       previous_partition ← get_partition(frame.key,frame.hop -1);
14       previous_owner ← m_partitions[previous_partition];
15       // changes the source, target, and hop fields respectively
16       frame ← change_fields(frame.target,previous_partition, frame.hop-1)
17       previous_thread.pipe.write(frame)
18    }
19 }
```

4.5.3.1 Deadlock Avoidance

Due to the locking philosophy applied in Memcached, it becomes easy to enter a deadlock state when dealing with more than a single data item. For instance, each data item lock is computed trough an hash function, which has an implicit collision probability higher than 0. Hence, there are at least two obvious scenarios where it can enter a deadlock state depicted in Figure 4.10.



a) In chain collision   b) Collision between chains

Figure 4.10: Deadlock scenarios at data item level

In Figure 4.10a) deadlock occurs when two data items share the same lock in a chain at 1 hop distance of each other. In Figure 4.10b) it occurs when any two distinct chains acquire an inverse sequence of data item hashes for locking.

Furthermore, the data item updates in Memcached do not reuse the same data item, since its size (assigned SlabClass) and header might change over time. Every change in the data item, implies a new data item allocation, and swapping with the previous. Due to this peculiarity, the update is a special case of the creation.

When performing replication, to avoid deadlock states we never hold two data item locks at the same time, or more than one lock between partitions. As such, to cope with this policy all operations (creation, update, and deletion) revolve around the core copy function in Listing 4.7, which performs the memory copy of the data item residing in the previous hop partition to a temporary buffer, and afterwards to the current partition.

The copy operation only fails if the data item has been removed or evicted. If the data item in the previous hop of the chain no longer exists, it must have been removed or evicted, as such we enforce it by expiring our own data item, and forwarding the frame, which will cause a cascade effect, deleting the entire chain.

The algorithm itself first acquires the data item lock from the partition of the previous hop and makes a local copy the header information and stores the body in a separate buffer in the thread context, releasing the lock afterwards.

Now we acquire the lock for the data item in the current partition, and with the header information that was retrieved it is possible to allocate the new data item. After the allocation is successful, we copy the data from the buffer to the newly allocated data item.

Listing 4.7: copy operation

```
1   int rep_copy(char* key, int nkey, short source, short target,
2                               item ** it_recv, uint32_t hv, uint32_t hv2){
3       //hv - data item hash from source partition
4       //hv2 - data item hash target source partition
5       item * it_new ← NULL;
6       char temp [1MB]; // represents the thread buffer
7       item_lock(hv);
8       item *it ← assoc_find(key, nkey, hv,source);
9       if(it= NULL){
10          item_unlock(hv);
11          return -1;
12      }
13      int source_flag ← 0;
14      if (it->nsuffix !← 0)
15          source_flag ← *ITEM_suffix(it);
16      rel_time_t source_exp ← it->exptime;
17      int source_nbytes ← it->nbytes;
18      short lambda ← it->lambda;
19      memcpy(temp, ITEM_data(it), it->nbytes);
20      item_unlock(hv);
```

```
21
22          item_lock(hv2);
23          it_new ← item_alloc(key, nkey,source_flag, source_exp, source_nbytes,target);
24          it_new->it_flags |= ITEM_ACTIVE;
25          memcpy(ITEM_data(it_new), temp, source_nbytes);
26          it_new->lambda ← lambda;
27          item_lock(hv2);
28
29          *it_recv ← it_new;
30      return source_nbytes;
31  }
```

At the end of the copy operation, the data item is not linked yet (not visible) to either the HashMap or the LRU. Afterwards it will be, and when that happens, the data item will enter the *HOT* queue as new data item, where if there was a previous it is removed. Due to this, its flag is preemptively set to *ACTIVE*, this ensures that when a replicated data item drops from the *HOT* queue, it will flow into the *WARM* queue, and not the *COLD* queue.

At the same time, before removing the previous data item, we first copy its update value and the sequence number to the newly created data item.

4.6 Summary

In this chapter we described the implementation details of our solution which is composed by four main components: (1) memory partition, (2) load monitor, (3) Socket Shift, and (4) Selective Internal Replication.

The memory partitioning is used to index each partition with O(1) time complexity and low space overhead.

The load monitor is responsible for providing an heuristic on when to perform a load relief method and which method to apply. The load monitor is supported by a pipe structure that resembles a mesh network. This is achieved by leveraging the already existing pipe structure in Memcached to provide inter thread communication that supports our algorithms.

The Socket Shift is initiated on the load monitor sub-routine which triggers a self migration process, where the existing connections only change ownership by its own initiative when being used. Furthermore, this algorithm is agnostic to the connection type.

The Selective Replication process is strongly dependent on the internal eviction policy to perform the replication, since it is a probabilistic model based on where the popular data items may reside. The amount of relief and how fast it can be provided can be regulated by the burst of data items, although higher bursts will result in higher miss ratio. The miss ratio is decreased through the use of scatter function to spread the replicated data items over the partitions.

The following chapter presents the experimental assessment over the components described in this chapter that composes our solution. In our evaluation we perform various experiments that showcase the benefits of employing our solution against the original Memcached system.

Chapter 5

Evaluation

This chapter presents a performance evaluation of our proposed modifications to Memcached to take advantage of increased parallelism, and the mechanisms to deal with its consequences, namely Socket Shift and Selective Internal Replication. Since the original eviction policies are used in our proposal, the measurements discussed focus mostly on throughput (operations per second), latency, and miss-ratio. The latency is correlated with the throughput, where an increase in throughput typically arises from a decrease in latency. However, the latency does not represent the end client, since the miss rate does not imply the interaction with a slower primary storage.

The evaluation is divided in five distinct sections:

Data Placement: provides an insight on the effects of deviation from the normal distribution for data distribution on caching systems.

Data Item Popularity specifies how the skew in item popularity is generated.

Partition Scheme: focus on measuring how the system deals with the partitioned scheme under uniform, and skewed workloads.

File Descriptors and Polling: demonstrates measurements of the impact of increasing the total number of open file descriptors to be monitored.

Socket Shift: presents experiments measuring the effects of load segmentation, and rearrangement.

Selective Internal Replication: presents the experimental assessment of the benefits that derive from using selective internal replication in high saturation scenarios.

5.1 Experimental Setup

The experimental setup for the experiments reported in this chapter were performed at *paravance* and *parasilo* clusters in the *Rennes* site from *grid*5000[1], where the *paravance* cluster is composed of 72 nodes split in two sections and the *parasilo* cluster of 28 nodes in a single section, as illustrated in the network topology in Figure 5.1. All the nodes in both cluster have the same specifications: 2 CPUs Intel Xeon E5-2630 v3, 8 cores/CPU, 20M Cache, 2.40/3.20 GHz, 128GB RAM, 5x558GB HDD, 186GB SSD, 10Gbps ethernet.

All disk I/O is very slow since it is performed over an NFS that resides in the *parasilo* network segment. The NFS can be easily seen through *netstat* which contains $parasilo-srv-3.renn:nfs$ as a Foreign Address in the Active Internet connections.

We try to run our experiments within a single section of the paravance cluster to minimize network interference. However, booking the servers while striving for isolation is not easy, and its hard to schedule the clusters for extended periods of time. As such times we also perform tests on *parasilo* cluster, which is highly susceptible to interference, as the network switch to which it is connected also serves the traffic from the *paravance* cluster.

All experiments are repeated at least from 3 to 5 times (when ran more than 5 times, the data with higher offsets is discarded), and to minimize the offsets in each experiment before each run of our version, the original Memcached was run in the same conditions.

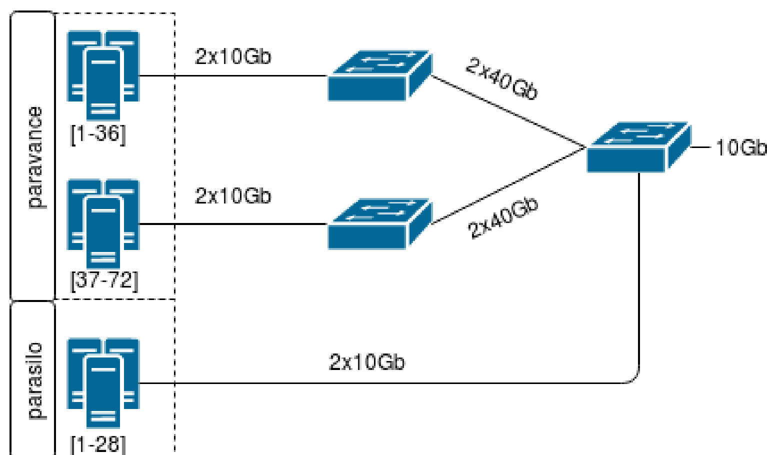


Figure 5.1: Section of Rennes network topology

Most of the performed experiments served to stress the system, using 17 nodes from a single section, where one node was used to setup the experiment, one as the target server executing the original Memcached or our version, and the remaining 15 are used to generate clients operations.

The clients are synchronous and based on the standard version of $Yahoo!CloudSystem Benchmark(YCSB)$, where the clients start executing incrementally every three minutes launching N client threads, where N is a parameter of the experiment. In the experiments

[1]https://www.grid5000.fr/w/Grid5000:Home

regarding replication, the clients were slightly modified to cope with the new server protocol and its interaction.

The default Zipfian generator present in YCSB (ScrambledZipfianGenerator) provides a Zipfian distribution on all client threads of a single process. However, when executing in several separate machines it causes a close to Uniform Distribution on the targeted server. This is due to each clients node having different subsets of *"hot"* data items, and operating over the same full data set. As such we redefined the Zipfian generator to have a deterministic selection, so that the *hot* data items are the same in all clients.

All client operations are performed over a set of 2 million data items with size of 1Kb (10 fields of 100 bytes), and a data item key configured to have 100 bytes. The Memcached stores these data items within the Slabsclass 13, the smallest size where they fit, with a chunk size of 1480 bytes.

During our experiments the Operating System version used in grid5000 was by default a pre-configured image of Debian 9-Stretch x86_64. However, half way through the experiments, from the beginning of the Selective Replication experiments in Section 5.7 and forward, the default version changed to Debian-10-Stretch x86_64, which caused a memory leak in the Libevent library that we were unable to isolate. The version of Libevent 2.1.8 and 2.1.10 triggered warnings when compilling and even though the version 2.1.11 (recently released) does not trigger warnings, it also displays the same memory leak problem.

As such, the latter experiments were carried with a different (from the previous) Debian-9-Stretch x86_64 image in deploy mode (required for none-default environments), but we were not able to fully replicate our initial environment for the experiments presented in the first half of this chapter. This became visible since we carried some of the initial tests where the overall performance was lower than expected, but with a similar pattern in both our solution and Memcached alike.

5.2 Data Placement

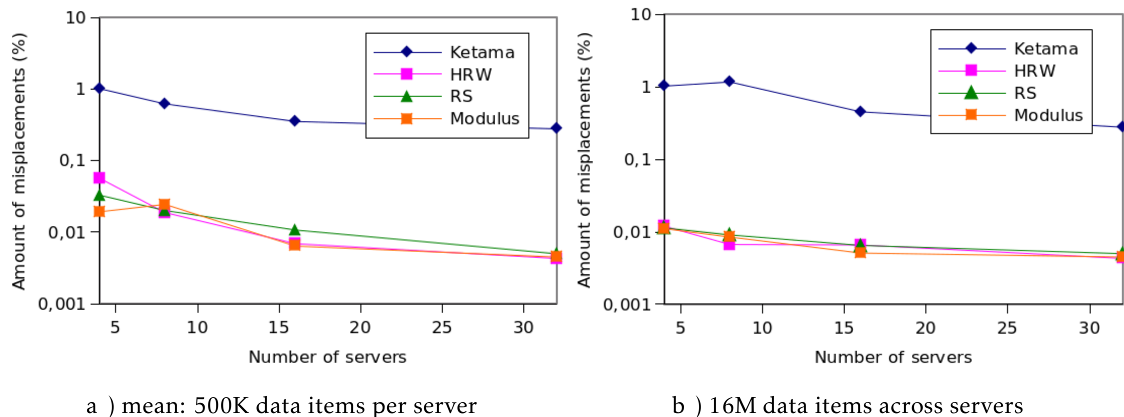
Our experimental evaluation relies on a environment based on the standard balls into bins model where the bins have a limit to the amount of balls they can hold. This cap is the same for all bins since we are assuming an homogeneous systems.

The best outcome for this problem would be a perfect data placement by splitting the set of balls over the set of bins, where all bins have the same number of balls. In the case where we set the bin cap at the mean distribution of all balls, any deviation from the perfect data placement implies that some balls will have to be discarded, and on the other hand, some of the bins will not be filled.

Assuming the balls that have some popularity associated and a bin chooses to discard balls based their popularity, a measurement on the impact of the misplacements can be made. However, this impact will be greatly influenced by the skew in item popularity. This example exposes the assumptions on the amount of misplacement's of a data

placement algorithms being strongly correlated with the miss-ratio of caching system, as refered in section 2.4.1.

We provide a comparison between the Modulus applied to a static number of servers, and fast lookup algorithms for data placement for dynamic environments, which are mentioned and discussed bellow. In the first experiment depicted in Figure 5.2-a) we increase the amount of data items proportionately to the amount of servers, as such we distribute 500k data items per server, and measure data misplacement based on its standard deviation. We note that, the experiments with the same amount of servers use the same data item keys that were generated before hand with a java uuid generator. The second experiment, reported in Figure5.2-b), also measures data misplacement over a fixed set of data items, only varying the amount of servers, in this case we use 16 Million data items.



a) mean: 500K data items per server

b) 16M data items across servers

Figure 5.2: Evaluation of data misplacement on a finite set of servers

The Highest Random Weight Hashing (HRW) is only presented for completeness, since its high computational requirements do not allow it to be applied to large environments, even though it accomplishes very low data misplacement.

For the representation of the Consistent Hashing we use the Ketama variant that is commonly applied to Memcached clients, where each server commonly has between 100 to 200 virtual buckets (values generated based on the hash of the server string), in this case the amount was set to 150 for all servers, with a range of 2^{32} values, commonly referred as continuum.

As a representation of inline data placement, we use our own implementation of Random Slicing, that is classified as a full line algorithm.

As expected, the Random Slicing algorithm is able to keep up with Highest Random Weight Hashing and the Modulus. On the other hand, the Ketama hashing performs several times more data misplacements than the second best in any test. This discrepancy between Ketama and the remaining algorithms can be explained by the difference in the sum of the virtual bucket intervals of each server, since itself resorts to these buckets to

normalize their cumulative intervals. Increasing the number of virtual buckets, or adding several more servers, should steadily improve the algorithm data misplacement.

Typically cache system pools do not tend to be composed by a very large set of servers. Hence, the data placement algorithm selection is an important decision that is able to minimize the impact on miss-ratio of data distribution in a dynamic environment. In our case, it becomes even more relevant, since we are performing a two tier indexation.

It is relevant that both layers of indexation use different algorithms, or at least different hash functions, to avoid bias in data placement, which could lead to worse results for data misplacement, and inherently in miss-ratio.

5.3 Data Item Popularity

Data item popularity is typically defined by a Zipfian distribution that is regulated by a α value that regulates the discrepancy of accesses relative to a normal distribution, applied to a finite set of data items. As the α value increases, so does the access frequency, but over a smaller sub-set of data items, as depicted in Figure5.3.

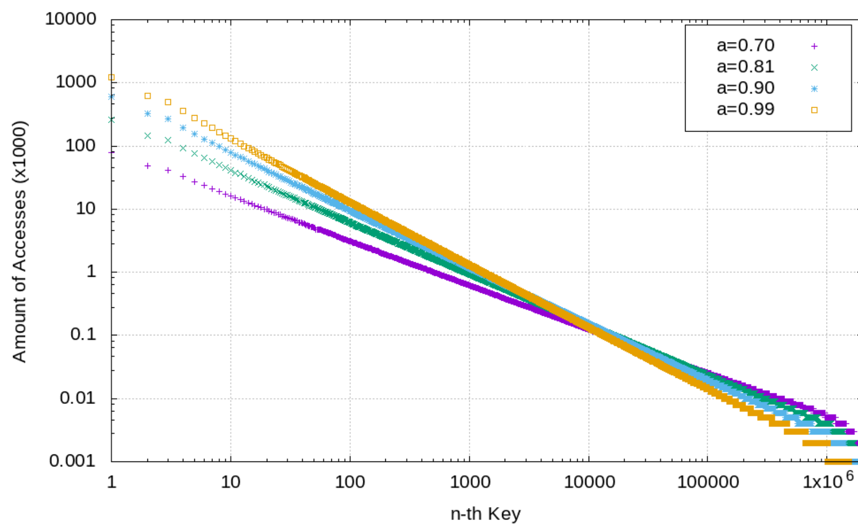


Figure 5.3: Zipfian generator for 2M keys with 20M accesses with varying α

The α values utilized in our experiments $(0.7, 0.81, 0.9, 0.99)$ are representative of typical web scenarios according to [35] which states: *"Most of the reported estimates of α lie in the range between 0.6 and 1.0. Where this study demonstrates that the parameter α is ranged between 0.75 and 0.85 for the Web servers, 0.64 and 0.83 for the Web proxies."*. We also take into consideration α values of 1 and 0.81 according to [1] based on traffic analysis, where the former value is extracted from America Online(AOL) Internet Service Provider(ISP), and the latter an Autonomous System (AS).

The plot in Figure 5.3 is an experiment ran in isolation with a single YCSB client. It performs 20 millions operations over a set of 2 million data items utilizing the Zipfian generator while varying the α value.

Table 5.1: Zipfian generator for 2M keys with 20M accesses with varying α

Zipfian Constant	1st Key*	1st Key (%) *	Unused keys **	Unused keys (%)**
$\alpha = 0.70$	78k	0.39%	26937	1.35%
$\alpha = 0.81$	256k	1.28%	86487	4.32%
$\alpha = 0.90$	602k	3.01%	215460	10.78%
$\alpha = 0.99$	1235k	6.18%	470751	23.54%

* Amount of accesses to the most popular key.
** Amount of keys that were not accessed.

Table 5.1 provides complementary information over Figure 5.3, which shows the effects of increasing the α value of the Zipfian distribution. This causes a large incidence of accesses on the most popular keys, where the highest α value points 6.18% of the 20 million accesses to a single key. On the other hand, this access discrepancy also implies a larger amount of keys have very few accesses or are not being accessed at all.

Henceforward, an α value will be used to specify the skew in data item access frequency. If not mentioned, a value of 0.99 is employed, which is also the default value emplayed by YCSB.

5.4 Partition Scheme

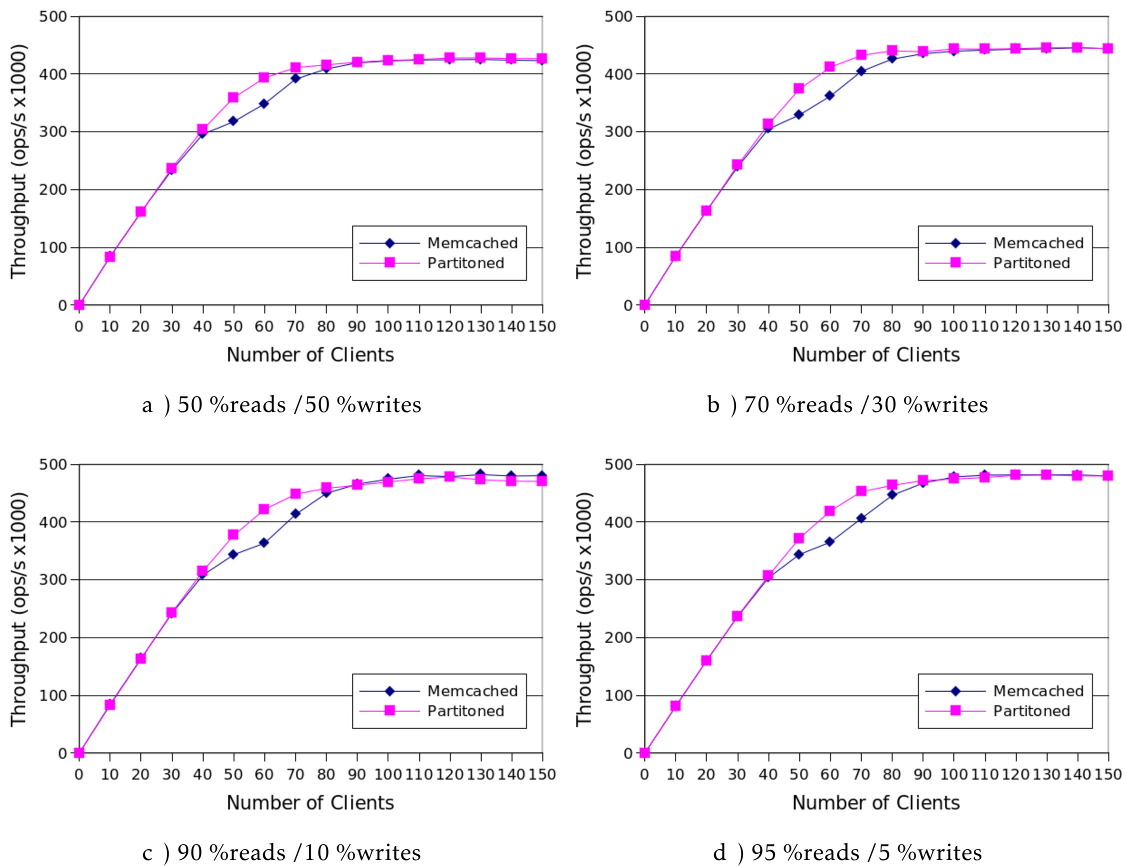
The first component of our solution performs a memory division to provide the concept of memory partitions through a socket interface, where each partition can only be accessed by a single thread that is bound to a single CPU Core. The sockets exposed to the clients create an abstraction for the server, as if it were composed by a set of servers. As such, each partition will be required to deal with a sub set of the data items that are assigned to the server.

The introduced overheads to achieve the partitioning scheme should be negligible: (1) change to the hash function for data item lock acquisition, where each request has to compute more 4 bytes than the original, and (2) indexing the respective partition which is an O(1) lookup.

These experiments use a uniform distribution on the accesses to the data items, as it normalizes the accesses that each CPU Core is subject to, while striving to achieve the maximum usability of the CPU Cores.

Figure 5.4 provides a comparison between the original Memcached and the Partitioned version with 4 worker threads, and different percentages of read and write operations.

In all sub-figures it is clear that the Memcached does not have a linear increase in throughput as the load increases. However, due to the parallelism present in our solution, we achieve a close to linear increase, where we stabilize at the same throughput as the Memcached, when the CPUs enter a very high saturation state, depicted in figure 5.5.



a) 50 %reads /50 %writes

b) 70 %reads /30 %writes

c) 90 %reads /10 %writes

d) 95 %reads /5 %writes

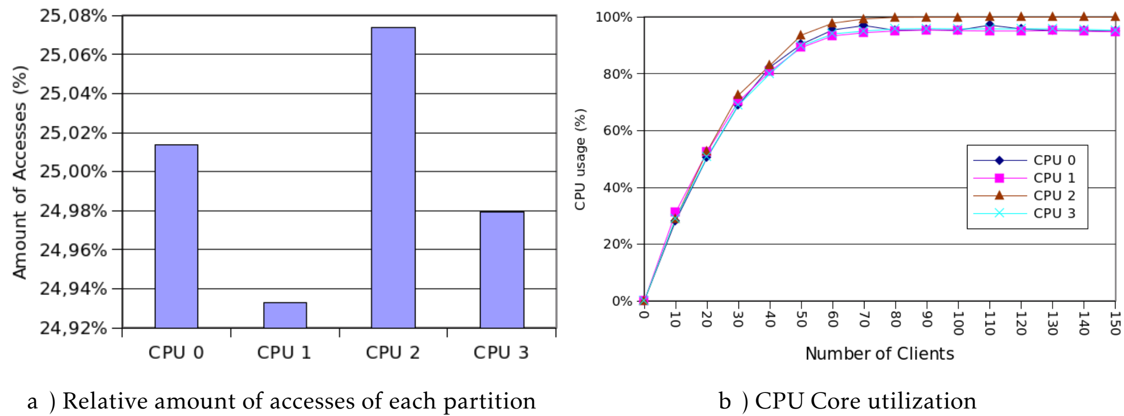
Figure 5.4: Comparison of Memcached vs its Partitioned version under a uniform data distribution

It is also clear that as the percentage of read operations increases, so does the maximum throughput, since the write operations are slightly more computationally demanding, also in all these experiments, our solution seems to reach the saturation point is around 70 *to* 80 active clients using a single server.

Even though the normal distribution used in these experiments does not provide high lock contention scenarios, it shows that our solution can at least match the original under high saturation scenarios when the load among the CPU Cores is properly distributed.

As the amount of client threads increases, so does the amount of lock contention in Memcached, which seem to be the main cause of the abnormal behaviour depicted in all plots (from 40 *to* 90 clients). In our case we achieve higher throughput, and do not incur in this behaviour since we reap the benefits of the increased parallelism, while accessing data items, trading concurrency for request sequencing dealt by socket polling.

This is only maintained until the first CPU Core reaches a saturation state (around 70 to 80 client threads). Since we are dealing with synchronous clients, the maximum that can be achieved is limited by the first CPU Core that becomes exhausted, where afterwards, all the CPUs will tend to have more stable load.



a) Relative amount of accesses of each partition

b) CPU Core utilization

Figure 5.5: 0.08% maximum variability in data accesses for 4 server threads with 95% read operations

At this point, the improvements granted by the increased parallelism will be relinquished, due to a bottleneck in the reception queue that deals with inbound traffic. Since the CPU is not able to handle the current load, it will introduce delays in all accesses to the partitions residing in that CPU Core. At this point, Memcached takes advantage of concurrency to provide load balancing, allowing it to reach the same saturation state later on (around 90 active clients), and across all CPUs.

Furthermore, one of the reasons that limits the increase in throughput is the CPU Core utilization. Even though we strive to normalize the amount of accesses to CPU Cores through memory partitions, they do not translate to a constant amount of load, since some CPU Core that executes a worker thread might incur in additional overhead that the remaining CPU Cores might not experience.

This phenomena is visible in Figure 5.5 that depicts the CPU usage and load distribution in the experiment depicted on the Figure 5.4 d), where even though the accesses to the partitions are uniformly distributed with a very low discrepancy (maximum variability of $0,08\%$), CPU Core 2 reaches 100% utilization while the others have not (see Figure 5.5 b)).

5.4.1 Skewed Workloads

The uniform distribution is not the typical access distribution for caching system in a real world context. As such we use skewed workloads, in this particular case the Zipfian distribution, to simulate a realistic execution environment.

In these experiments we focus on read mostly workloads using 95% read operations, since it appears to be the most common scenario. We only vary the α value to understand how the system behaves when focusing the accesses in increasingly smaller subsets of data items.

The first two plots in Figure 5.6 report the measured throughput increase compared

Table 5.2: Relative throughput Gain for N static partitions and 4 server threads

Zipfian Constant	Throughput (Ops/s) *	Stdev (Ops/s) **	Stdev (%)**
uniformdist.	480k	384	0.08
$\alpha = 0.70$	499k	675	0.14
$\alpha = 0.81$	507k	1858	0.37
$\alpha = 0.90$	485k	3489	0.72
$\alpha = 0.99$	481k	7963	1.65

* Average highest values, after saturation is reached .
** Standard Deviation of the load distribution between the 4 server threads.

with the original Memcached, even under high saturation scenarios. This is different than when we employ a uniform distribution, where both solutions were colinear under high saturation, as raising the α value increases the access to smaller subsets of data item, resulting in an increase of hit ratio in the L1, L2 and L3 Caches allowing faster data retrieval.

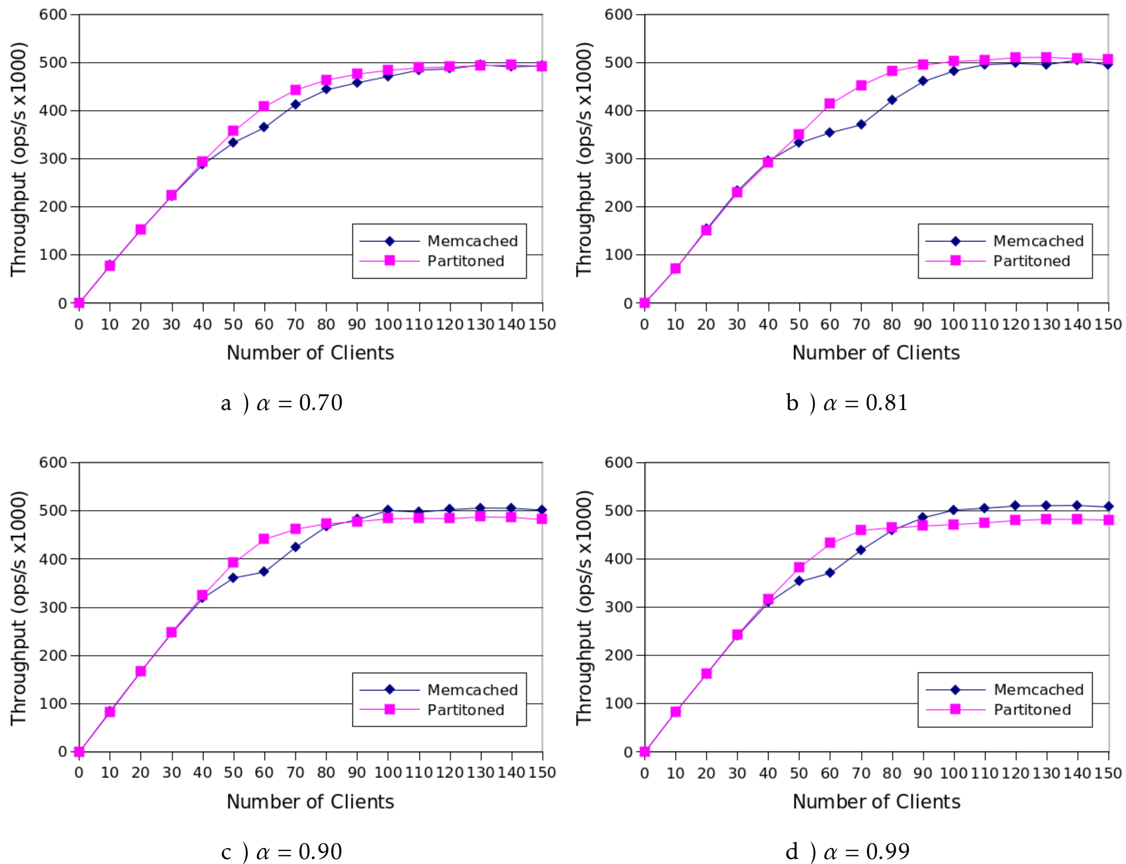


Figure 5.6: Comparison of Memcached vs its Partitioned version under a Zipfian distribution with $\alpha = 0.99$.

The last two plots in Figure 5.6 report the throughput, with similar results from the previous scenario. However, when reaching high saturation, the system can not keep up

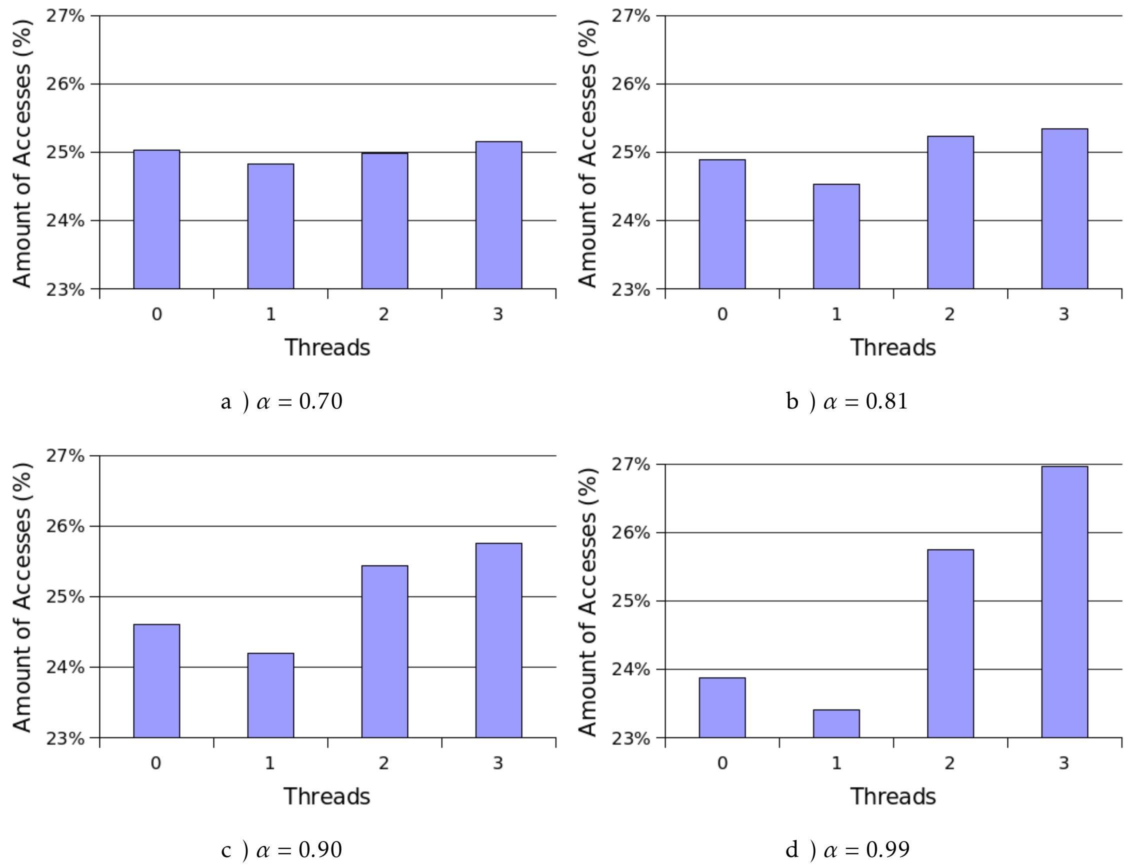


Figure 5.7: Comparison of Memcached vs its Partitioned version varying Zipfian distribution with 95% read and 5% write operations.

with the unbalance caused by the skew in accesses that each partition is subject to. This causes one of the CPUs to reach its maximum saturation sooner, imposing a limit on all other CPUs. Hence, establishing a cap on the maximum throughput that can be achieved.

Since we are applying skew in access frequencies over the data items, it impacts each partition in accordance, where higher α values cause greater load discrepancies. This implicitly affects each CPU Core since its load can be extrapolated based on the sum of all the partitions it owns, as shown in Figure 5.7

This phenomena is also clearly depicted in Table 5.2, where we can observe that as the α value increases, so does the unbalance between partitions, and implicitly the load that each CPU Core is subject to, represented by the standard deviation. Until reaching $\alpha = 0.81$ the system performs extremely well, since its gain from the CPU cache hits out weight the load discrepancy. However, afterwards, the exact opposite occurs and the system performance starts to decline after the first CPU reaches a saturation point. In our last case ($\alpha = 0.99$), it seems that the maximum throughput achieved is similar to the uniform distribution.

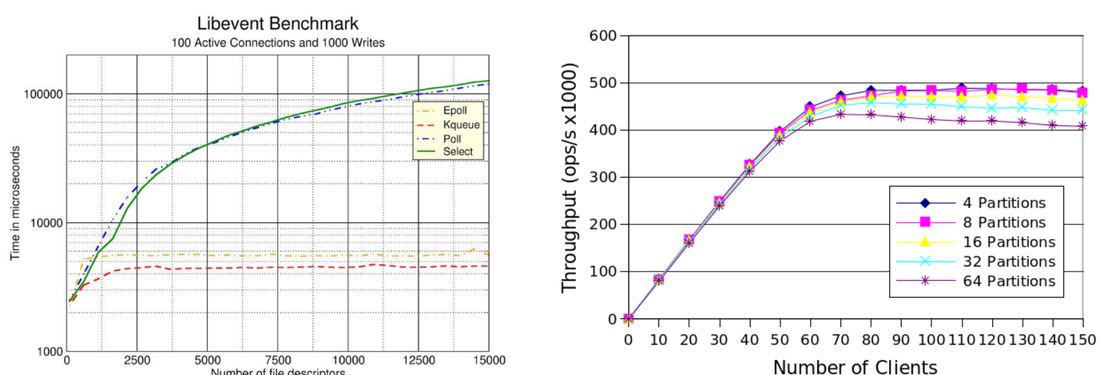
5.5 Open File Descriptors and Polling

In this section we evaluate the impact on increasing the amount of open file descriptors and polling them for data, where we double the amount of partitions in each experiment.

In the end, a total of $15(servers) * 10(clients) * 64(sockets) = 9600$ file descriptors are open in our server ruining with 4 threads.

Each thread should have to deal with the respective file descriptors associated of its assigned partitions (reaching a total of 2400 file descriptors per thread), with an increasing amount of active connections (reaching 150) actively submitting requests.

To change the maximum number of open file descriptors, we modified the configuration file "/etc/security/limits.conf"presented in greater detail on Annex II.



a) Libevent benchmark on increasing the amount of file descriptors.

b) Zipfian distribution $\alpha = 0.99$ and 95% reads.

Figure 5.8: Increase the amount of partitions with 4 server threads and similar increase of file descriptors from Libevent.

The data item distribution per thread does not change as we increase the amount of partitions, since we assign the partitions sequentially, where each thread keeps the partitions that represent multiples of its identifier.

Libevent manages the file descriptors using the most suited library present in the underlying system (select, poll, epoll or kpoll). In our case, epoll is used, since we are using a linux distribution as our experimental environment, which provides a much more efficient file descriptor management than standard select.

Event though it is well documented that epoll can maintain a close to constant time on polling events, it is only true to large amount of file descriptors.

Thus, we seem to be caught in the beginning of the plot in Figure 5.8 a)[2], where the latency increases exponentially (linearly in a logarithmic scale) before stabilizing.

This impact is easily seen in our experiments reported on Figure 5.8 b) when reaching core saturation, as the number of partitions increases. It also suggests that above two

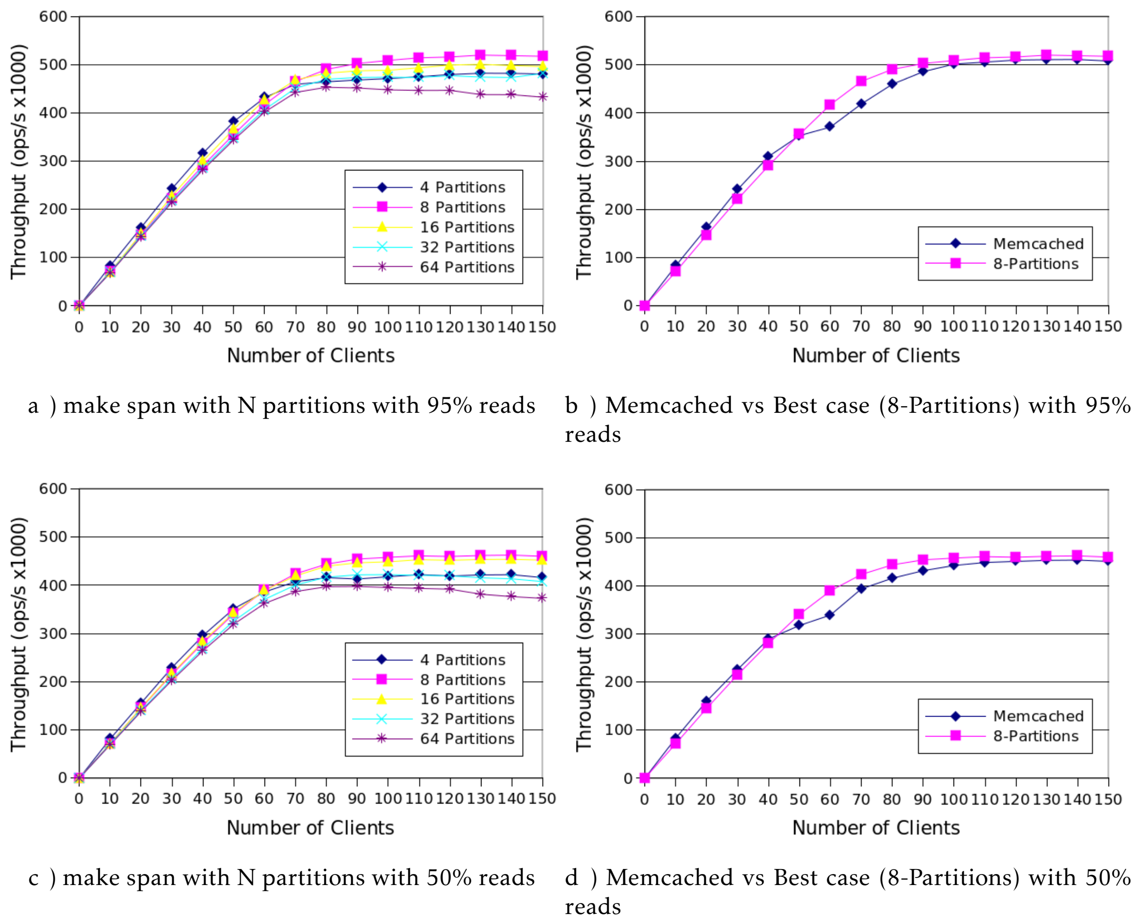
[2]The Figure was taken from http://libevent.org/

partitions per thread the performance will start to degrade just from managing the file descriptors with more than 150 active connections.

Our experiments are focused on four threads due to the bottleneck caused by the amount of open file descriptors when increasing the amount of threads. For more detail the reader can see Appendix A.

5.6 Socket Shift

The Socket Shift is expected to delay core saturation by rearranging the partitions by the available CPU Cores, this relief method only works in the presence of available computational power in the remaining CPUs.



a) make span with N partitions with 95% reads

b) Memcached vs Best case (8-Partitions) with 95% reads

c) make span with N partitions with 50% reads

d) Memcached vs Best case (8-Partitions) with 50% reads

Figure 5.9: Comparison of Memcached vs its Partitioned version with Socket Shift.

In these experiments we enable only the socket shift algorithm to redistribute the partitions by the four worker threads.

However, having only four partitions would not change the current outcome since it does not allow any kind of rebalancing. As such, we increase the amount of partition, allowing a better load division, which strives to overlap or surpass the original Memcached

as the number of partitions increases.

Increasing the amount of partitions, besides allowing a better load splitting, also decreases the amount of data misplacements when indexing, as seen previously in Figure 5.2 from Section 5.2. However, it is followed by the down side of an increasing amount of open file descriptor that have to be managed by each worker thread, as seen in the previous section.

Figures 5.9 and 5.11 present an analysis over the partitioned version of the Memcached with the socket shift algorithm in place, where the condition to trigger a socket shift is a lower standard deviation after performing a simulation of the data placement with the make span algorithm. After the socket shift is triggered there is always a grace period of the same time of the sample window duration.

The four partition scenario provides a comparison on the overhead caused by the socket shift algorithm, where the socket transition between threads disrupts the partition availability reducing the throughput at each step.

Additionally, as the amount of partitions increases, so does the overall amount of file descriptors open by the Memcached process, taking its toll on the performance, since a larger set of file decriptors have to be migrated. This effect can be observed in Figure 5.9 c).

5.6.1 Exponential Moving Average (EMA)

Through careful observation of several samples of the internal state of our solution we were able to infer that only taking the absolute value of the standard deviation under consideration for the socket shift trigger is dangerous, since even though the clients produce a Zipfian distribution in bounded time intervals, at specific points in time it tends to have a high variance.

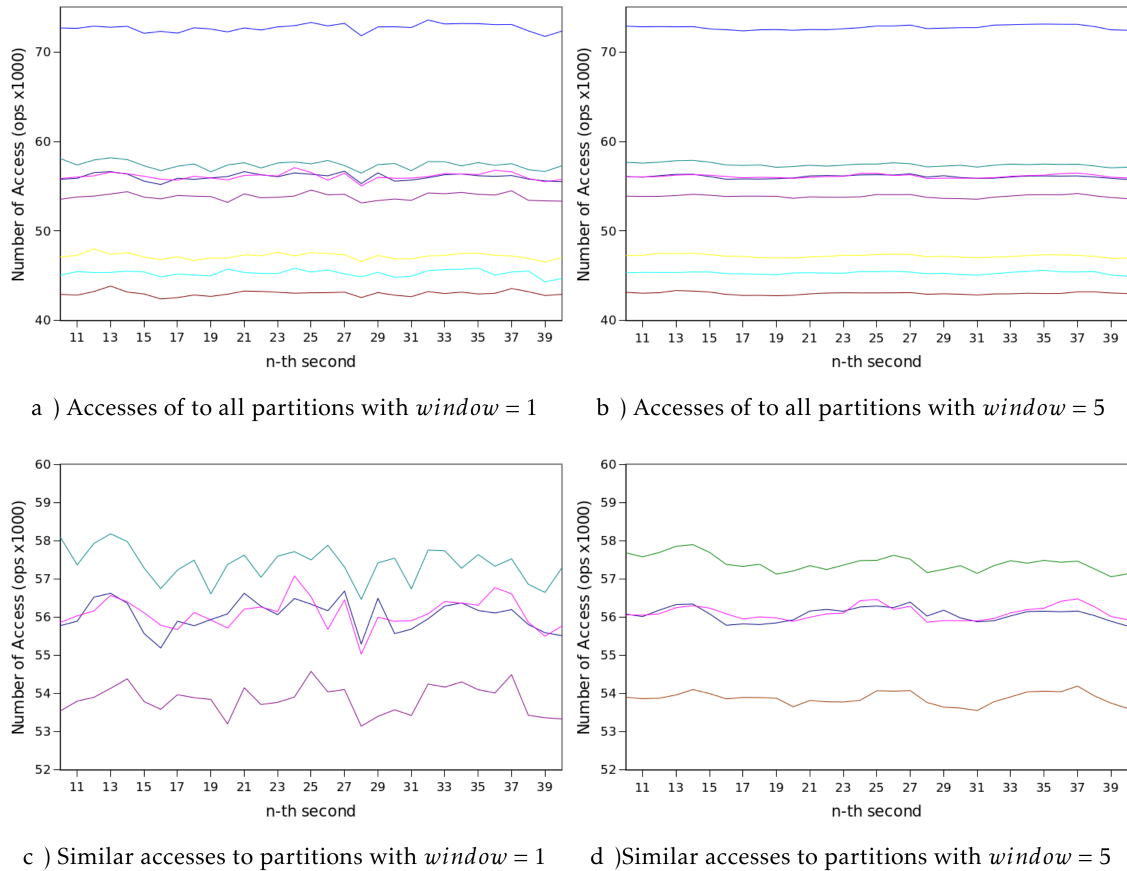
This behaviour was already expected, thus we use the Exponential Moving Average (EMA) to reduce these spikes, as demonstrated in Fig.5.10, where we plot the accesses count of a server with eight partitions with its current load (window = 1 second) and with EMA applied (window = 5 seconds).

However, it seems that our window for the sample is not large enough to eliminate the momentarily spikes that occur with a reasonable frequency. Thus, a larger window should be used to mitigate the sensibility of the EMA.

Scenarios where at least two partitions have a similar access count can be disruptive, as represented in Figures 5.10 c) and d) (notice the two overlapping partitions). It can greatly increase the probability of a rebalance trigger if the standard deviation is reduced in the make-span simulation.

In these cases, it will cause these partitions to switch owners or in the worst case, a cascade effect that also affects the partitions with lower accesses.

The ideal scenario in our experiments would be for the socked shift operation to be performed at most around 8 to 9 times, due to the entry of new clients in the systems



a) Accesses of to all partitions with $window = 1$ b) Accesses of to all partitions with $window = 5$

c) Similar accesses to partitions with $window = 1$ d)Similar accesses to partitions with $window = 5$

Figure 5.10: 30 second trace of to eight partitions with 95% read operations.

which can cause discrepancies in the access count of each partition. However, in our experiments, the amount of socket shift varies, mostly from 28 to 172. Also, there are still some outliers that have only performed 3 or 4 operations during the whole run, and some have achieved close to 360 operations.

With an environment as unstable as ours, increasing the window indefinitely should remove the load spikes. However this is not a viable solution to reduce the rebalance triggers on its own. Additionaly, increasing the window and establishing a lower-bound on the expected gain from performing the socket shift seems to constitute a more suitable solution.

5.6.2 MakeSpan

The Makespan minimization algorithm is the core element in the partition rearrangement, from which our solution derives, where instead of taking a set of task times, we extrapolate the amount of accesses to a partition in a bounded time frame, which are then distributed by a set of workers, in our case the threads.

Increasing the amount of partitions will also increase the load fragmentation, resulting on a better load distribution, since fundamentally it is a combinatorial optimization

Table 5.3: Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations

Partitions	Partitioned*	Socket Shift *	Rel. Growth (%)	Stdev (Ops/s)**	Stdev (%)**
8 Partitions	481k	517k	7.48%	6294	1.22%
16 Partitions	470k	498k	5.96%	5124	1.03%
32 Partitions	442k	474k	7.24%	3386	0.71%
64 Partitions	413k	441k	6.78%	1088	0.26%

* Average values, after saturation is reached .
** Standard Deviation of the load distribution between the 4 server threads.

problem.

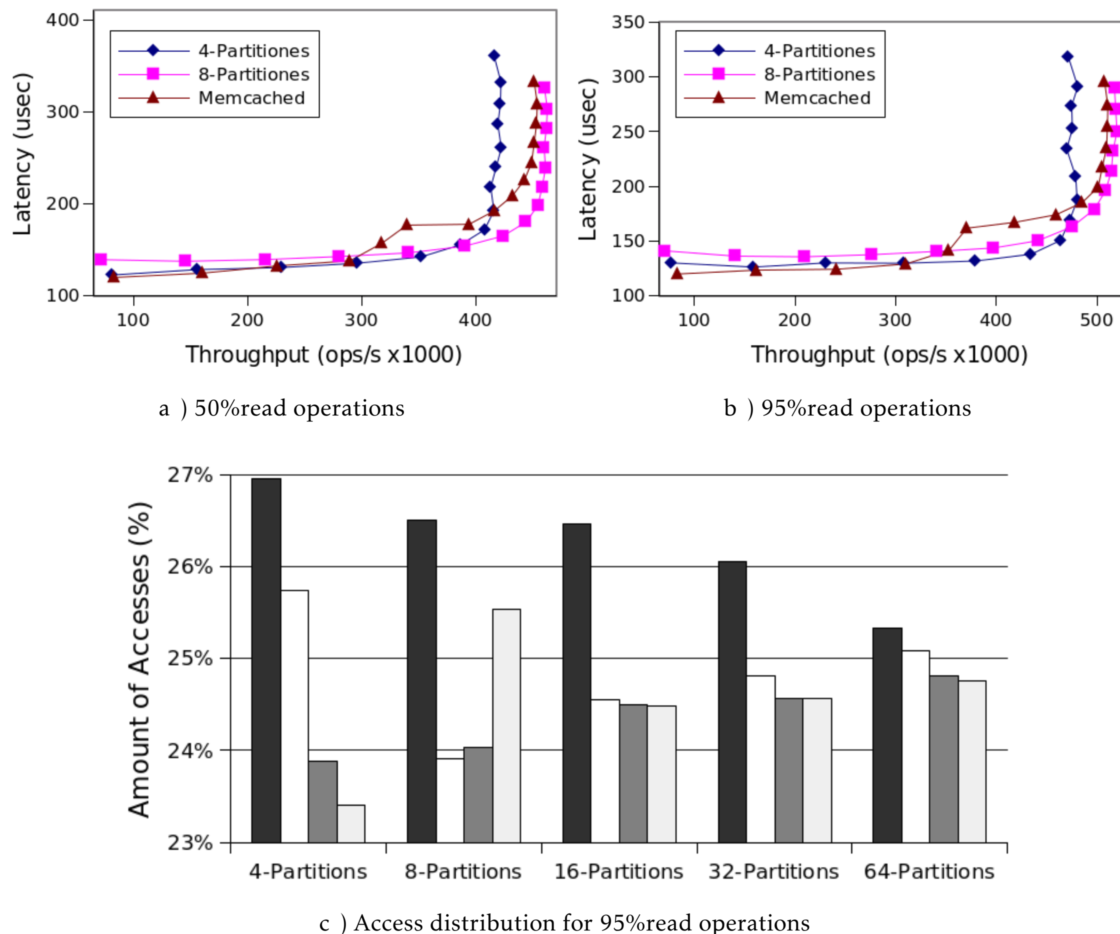


Figure 5.11: Throughput and Load Distribution of the Partitioned version with socket Shift

This phenomena can be easily seen in Figure 5.11 c), which depicts the average load distribution of all threads after saturation is reached, and where we can see that the load discrepancies reduce as the amount of partitions increases.

In Table 5.3, the same is also showed through the standard deviation and the percentage over the socket shift algorithm. However, even though we are reducing the load discrepancy between threads, it seems that the relative growth to the partitioned version does not increase beyond 7,48% in throughput. The four partition scenario is not reported in Table 5.3 since it did not exhibit any performance improvements.

The gains achieved in load balancing are not truly visible, since the amount of file descriptor that have to be moved at each socket shift operation also largely increases as the amount of partitions increases, resulting in higher thread down time.

Figures 5.11 a) and b) show a comparison based on throughput latency of Memcached, eight partition (best-scenario), and the four partition. It provides a reference of the weight of performing several socket shift operations. In the case of the four partition version this operation only occurs at most once, since the load is not divisible. However, we can catch a glimpse of its effect in Figure 5.11 b), because the first measurement in the four partitions scenario denotes higher latency then the following measurements in the experiment.

Furthermore, the four partition scenario is used as a base comparison, and it shows that even though the make span algorithm is being computed regularly (in the main thread) it does not impact the overall system performance, since the results are similar to all the previous experiments, and almost collinear with Memcached in the plots fist segments.

5.7 Selective Internal Replication

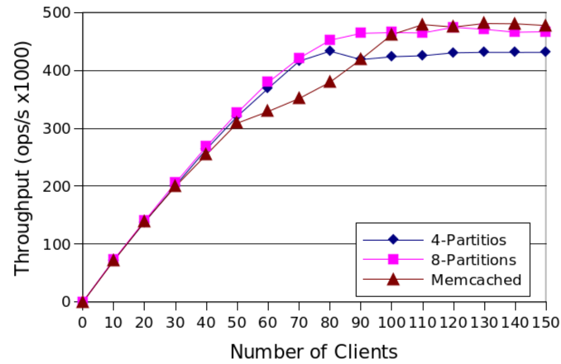
The Selective Internal Replication (SIR) aims to surpass the limits imposed by the partitioned scheme by dividing the load of very popular data item among multiple worker threads, promoting higher item availability for popular data items while discarding the least popular, unlike the Socket Shift that tries to simply redistribute the load.

We remind the reader, that for these set of experiments the execution environment changed.

Before we experimentally evaluate this algorithm, we conducted a simple validation in the new deployment environment with the standard Memcached, the four partition version, and eight partition version with socket shift algorithm to establish a base line for comparison, depicted in Figure 5.12.

To have a clear understanding on how SIR behaves over time as the amount of clients increases in our experiments, we also reduced to the total amount of available space to 1000MB so that we can have a miss ratio with higher significance, allowing a better grasp on the impact of the eviction process.

Furthermore, changing the Soft limit allows a more aggressive or more passive replication burst, where lower values will allow truly popular data items to become more replicated, while a higher value provides a faster partition relief with a higher number of data item flushes. For these experiments the soft limit was to 5% of the *warm* queue,



a) Access distribution in 8 Partitions

Figure 5.12: Base Line comparison with Socket Shift

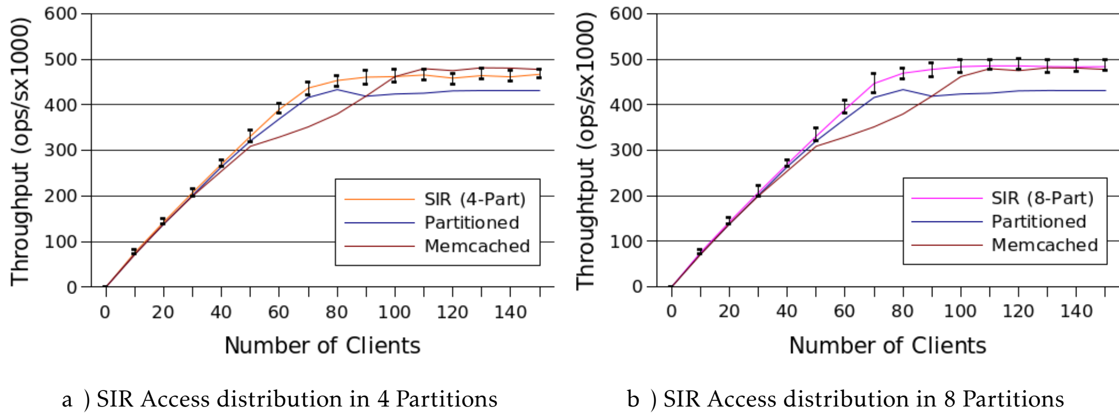
the hard limit to 5% of the overall memory, the γ value as the same as the amount of partitions, and a grace period of 10 seconds.

It should also be taken into account that very high bursts of replication over potentially several data items can happen and the pipe buffer might not be able to hold all of those data objects when they are being replicated, leading some threads to block in writes on these pipes. This happens since bursts are based on a fixed percentage over the *warm* queues and each propagation should require a maximum of 300 bytes, based on the header and maximum key size. The default pipe buffer size is 819200 bytes, which allows at least around 2700 buffered frames. However, this value can be changed, for more details see Annex II.

In the beginning of these experiments there is a *warmup* phase, which consists of two clients performing separate roles: (1) data insertion, and (2) maintaining the data distribution. The first performs 2 million data item insertions in a slow pace, we set it with 20 client threads, when finished it kills the other client and starts the normal procedure. The second client is a normal client, but has a static amount of threads configured, in our case we used 40 threads. The *warmup* phase ends with almost 9% avg miss rate, which quickly decreases in all experiments (see Figure 5.14b)).

In Figure 5.13 we present the results of the SIR with four and eight partitions respectively. The solution with four partition shows improvements relative to the its static counterpart. However, it is not enough to reach the Memcached performance. The eight partition scenario, provides the insight that increasing the number of partitions seems to achieve better results, where some of the experiments have achieved a clearly higher throughput than Memcached, even though the average still remains almost collinear.

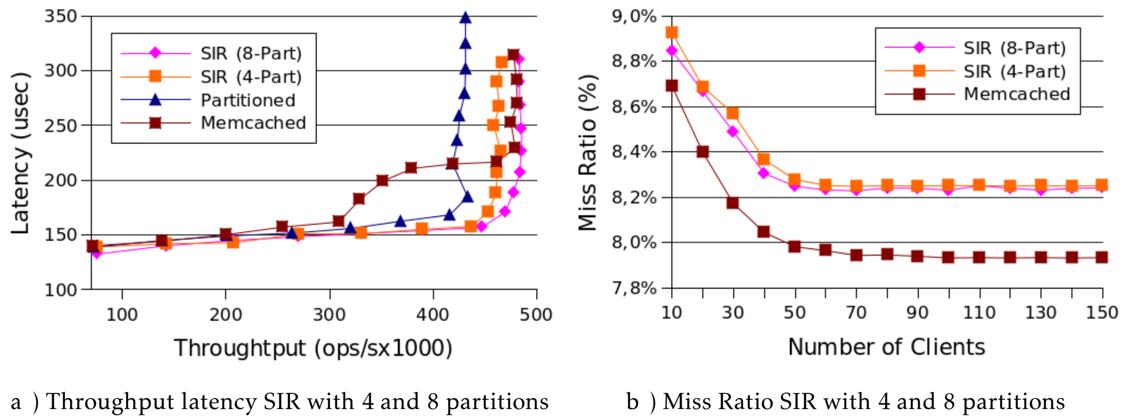
Since the Selective Replication is a probabilistic model it was expected that the values should eventually converge to some clear throughput, but the discrepancy between min and max (denoted by the vertical bar) is not enough to determine the convergence value. This may be due to the frequent shrink and growth process that prevents Selective Replication from stabilizing.



a) SIR Access distribution in 4 Partitions

b) SIR Access distribution in 8 Partitions

Figure 5.13: average SIR with Min-Max markers



a) Throughput latency SIR with 4 and 8 partitions

b) Miss Ratio SIR with 4 and 8 partitions

Figure 5.14: SIR comparison based on Throughput, Latency and Miss ratio

The miss ratio loss in this processed when compared with Memcached is around 0,27% which seems to be a good trade off, considering the gain in throughput from the four static partitions. It is not obvious that there is a discrepancy in miss ratio from the four to the eight partition scenario, since the initial samples are still affected by the *warmup* that can cause abnormalities, and since after stabilizing, they seem to overlap. The cache size was set at 1000MB to only be able to hold close to 35% of the 2M data items. However, experiments with even smaller caches may provide more evidence to infer the miss ratio behaviour in those scenarios.

The results reported in Table 5.4 shows that almost all data is retrieved successfully by the clients, where each client keeps track of the amount of γ values it holds, and the number of distinct γs. However, the number of chain overlaps is provided by the server, through the sequence number verification where the frame is forwarded without performing an update.

By increasing the amount of partitions, besides allowing longer chains, implicitly provides a better load distribution, since the average throughput has greatly increased,

Table 5.4: Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations

Partitions	TP*	Max TP*	Nr Chains*	Nr Max Chains**	Chain Overlaps (%)**
Memcached	475k	481k	0	0	0%
4-Partitions	428k	433k	0	0	0%
SIR (4 Part.)	461k	467k	24,2k	3891	38.71%
SIR (8 Part.)	483k	498k	21,4k	72	31.09%

TP is an abreviation of throughput
* Average values, after saturation is reached.
** Average values, of the amount of chains with the maximum size.

having a maximum achieved throughput of nearly 500K operations per second. The reduction on the total amount of chains can be caused by longer chains, but also due to lower burst sizes (nearly half) when compared to four partitions, which avoids the creation of a very large amount of short temporary chains.

The amount of chain overlaps is much higher than expected, which is limited by the scatter function in place. For the scatter function we used a variant of crc32 hash algorithm, which seemed a good fit in the early stages of this thesis, which was validated over a small set of keys. However, it seems that the cascade effect caused by the $salt$, was not enough to minimize the impact of the reduction of an hash from a large integer to a small interval.

5.8 Summary

In this chapter we presented and discussed our experimental evaluation validating the proposed solution, analyzing each component separately, from the client to the server internals.

Our first experiment confirmed that the Random Slicing is a suitable algorithm for data placement with dynamic memberships, since it achieves low data misplacement, which reduces the implicit miss ratio. The Modules and the Highest Random Weight used in our experiment showed competitive results.

In all experiments, the increase in parallelism from the partition scheme is clearly noticeable through the close to linear increase in throughput.

The method we choose to expose the partitions to the clients proved to be a bottleneck on the performance severely crippling the overall performance gain.

The Socket Shift demonstrated promising increases in performance by fulfilling its role to redistribute the load across workers.

The Selective Internal Replication also provided promising increases in performance, since by slightly increasing the miss ratio we were able to achieve much higher throughput than the partitioned version.

Both these algorithms tend to improve as the amount of partitions increases. However, due to the bottleneck in managing the file descriptors we were only able to validate its tendency.

In the next chapter we conclude the document with final remarks and points directions for future work.

Chapter 6

Conclusion and Future Work

Conclusion

Providing Exclusive data access over a partitioned system has clear gains in the amount of operations (independently from the amount of read an write operations) that can be performed before reaching saturation providing a close to linear throughput increase, since it benefits from very low lock contention.

However, by itself, this scheme is not able to cope with saturation scenarios, where it start to under performed, when compared with a concurrent solution that inherently allows load balancing.

As such, our contribution resides in the overload compensation mechanisms for caching systems with reduced meta-data applied to a stand alone server model, namely Socket Shift as a coarse method for load redistribution without memory waste, and Internal Selective Replication (SIR) based on controlled burst/shrink replication with a scatter function.

These mechanisms rely on higher number of partitions for better results, where more partitions provides a better load division.

The socket shift mechanism, even tough it redistributes the load segments though the available CPU Cores, is not able to achieve high performance on its own with a large number of partitions, mostly due to the overhead generated by the large amount of open file descriptors, and also its effects on the amount of down time each thread experiences during the migration.

The Selective Internal Replication is performed based on load spike discrepancies, where more partitions result in a more fine grain burst replication per iteration. However, it also suffers from overhead due to the large amount of open file descriptors.

Our experimented results, have shown an increase in throughput even under heavy

high load scenarios, allowing each server to scale independently, which greatly impacts when considering traditional web app deployments with clusters of many caching severs.

Future Work

The solution could be further improved by removing most locks. The existence of the segmented LRU eviction policy causes a great interference with the partition isolation, forcing the slabs and data items to be protected by locks.

The eviction policy choice, although outside the scope of this thesis, plays a crucial role in the amount of improvements that can be achieved.

Memcached is able to scale due to two threads that manage the LRU policy that have two main responsibilities: (1) avoid bumping item at each data items access, by performing item bumping within, and between queues asynchronously; and (2) reclaim space by discarding expired data item.

The designing of a LRU queue able to achieve the same functionality, without the need for the auxiliary threads, while keeping the overhead per request very low, would remove the need for locking, except when gathering statistics over the system information, since it an aggregation of the local values is still required.

Furthermore, there is also an hypothesis for external load relief, based on external replication, which could be designed to cooperate with the already existing SIR that could be seen as a local detection mechanism besides it intended purpose, since data items with high popularity will have a high γ value.

Finally, the results point out that a large amount of clients directly connected to the cache degrades its performance since it does not allow a large amount of partitions to exist. As such, mechanisms to perform unsharding of connections, for instance a middleware, could be an interesting future direction to explore.

Bibliography

[1] L. A. Adamic and B. A. Huberman. "Zipf's law and the Internet." In: *Glottometrics* 3.1 (2002), pp. 143–150.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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: *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.

[8] 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.

[9] "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.

[10] 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.

[11] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies." In: *SIAM JOURNAL ON APPLIED MATHEMATICS* 17.2 (1969), pp. 416–429.

[12] A. Gupta, B. Liskov, and R. Rodrigues. "One Hop Lookups for Peer-to-peer Overlays." In: *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*. HOTOS'03. Lihue, Hawaii: USENIX Association, 2003, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1251054.1251056.

[13] 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.

[14] 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.

[15] 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.

[16] F. Junqueira and B. Reed. *ZooKeeper: distributed process coordination*. "O'Reilly Media, Inc.", 2013.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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).

[29] M. Raab and A. Steger. ""Balls into Bins" — A Simple and Tight Analysis." In: *Randomization and Approximation Techniques in Computer Science*. Ed. by M. Luby, J. D. P. Rolim, and M. Serna. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 159–170. ISBN: 978-3-540-49543-7.

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

[31] 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.

[32] 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.

[33] 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.

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

[35] L. Shi, Z. Gu, L. Wei, and Y. Shi. "An applicative study of Zipf's law on web cache." In: *International Journal of Information Technology* 12.4 (2006), pp. 49–58.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] W. Ugaz, I. Sánchez, and A. M. Alonso. "Adaptive EWMA control charts with time-varying smoothing parameter." In: *The International Journal of Advanced Manufacturing Technology* 93.9 (2017), pp. 3847–3858. ISSN: 1433-3015. DOI: 10.1007/s00170-017-0792-1. URL: https://doi.org/10.1007/s00170-017-0792-1.

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

[42] V. V. Vazirani. *Approximation Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 79–83. ISBN: 978-3-642-08469-0.

[43] 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.

[44] 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.

[45] 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.

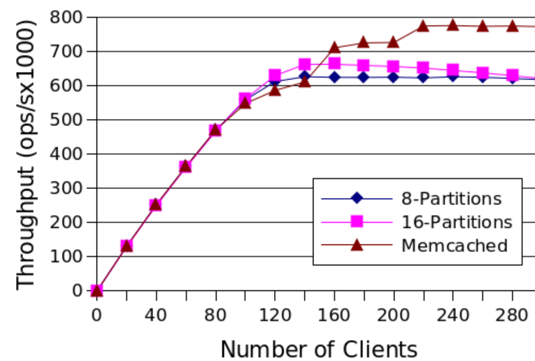
[46] N. Zaidenberg, L. Gavish, and Y. Meir. "New Caching Algorithms Performance Evaluation." In: *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. Spects '15. Chicago, Illinois: Society for Computer Simulation International, 2015, pp. 1–7.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

[47] 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.

Appendix

Impact of File Descriptors

In Figure A.1 we can see the same experiments performed in Section 5.6 but with eight server threads. In this figure we can see a similar pattern as previously seen in Memcached, where the increase in throughput forms a ladder with increased clients. The 8 partition scenario hits a cap due to high access discrepancies as would be expected. However, the sixteen partition scenario displays an odd behaviour, where even though it increased in throughput due to the reduction imbalance, instead of reaching a cap in throughput and stabilizing, it actually decreases after reaching its peak as the amount of open connections increases. The values can be assessed in Table A.1.



a) Memcached vs 8 partitions vs 16 partitions

Figure A.1: Comparison between Memcached and Socket Shift with 8 Sever Threads

Table A.1: Socket Shift relative throughput Gain for N Partitions and 4 server threads with 95% read operations

Partitions	Socket Shift *	Stdev (Ops/s)**	Stdev (%)**
8 Partitions	624k	13446	2.16%
16 Partitions	662	9414	1.42%

* Maximum values, after saturation is reached .
** Standard Deviation of the load distribution between the 8 server threads.

Annex I

EMA CPU load

The values present in */proc/stat* are incremental values since boot time. It requires two reads in separated by a wait period in order to calculate a how much each component has used during that period.

```
> cat /proc/stat
cpu 1748231 6396 621364 15472338 59772 0 18694 0 0 0
cpu0 446662 1527 158376 3861123 11050 0 5704 0 0 0
cpu1 418243 1659 136342 3900903 22398 0 4501 0 0 0
cpu2 449180 1494 164490 3840774 15069 0 5102 0 0 0
cpu3 434145 1715 162155 3869536 11254 0 3386 0 0 0
intr 190521331 48 52354 0 0 0 [... lots more numbers ...]
ctxt 338412791
btime 1551364793
[... other parameters ...]
```

The parameters we are interested are the first four of each cpu line:

(1)user normal processes executing in user mode

(2)nice niced processes executing in user mode

(3)system processes executing in kernel mode

(4)idle twiddling thumbs

```
/**
* Global variables:
* float weight
* int n_procs
* FILE* fp3;
*Description: Read Unix /proc/stat file, parse it and calculate the Exponential Moving
Average for each cpu
```

```
8   **/
9   float* get_EMA_per_CPU(){
10
11       static float* EMA_cpu← NULL;
12       static unsigned long long int *old_fields← NULL;
13
14       if(!fp3){
15           fp3 ← fopen("/proc/stat","r");
16       }
17       if(!old_fields)
18           old_fields← calloc(4*n_procs,sizeof(unsigned long long int));
19
20      if(!EMA_cpu)
21          EMA_cpu←  calloc(n_procs,sizeof(float));
22
23      /*** Discard the first line - overall CPU metrics***/
24      fseek (fp3, 0, SEEK_SET);
25      fflush(fp3);
26      unsigned long long int tmp;
27      int ret ← fscanf (fp3, "cpu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu\n",
28                &tmp,
29                &tmp,
30                &tmp,
31                &tmp,
32                &tmp,
33                &tmp,
34                &tmp,
35                &tmp,
36                &tmp,
37                &tmp);
38      if(ret<0){
39          printf("error\n");
40      }
41
42      float avg_EMA_CPU ← 0;
43
44      for(int cpus= 0;cpus<n_procs;cpus++){
45          int cpu;
46          int retval;
47          unsigned long long int tmp2;
48          unsigned long long int fields[4];
49          retval ← fscanf (fp3, "cpu%d␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu␣%llu\n",
50                      &cpu,
51                      &fields[0],
52                      &fields[1],
53                      &fields[2],
54                      &fields[3],
55                      &tmp2,
56                      &tmp2,
57                      &tmp2,
```

```
58                          &tmp2,
59                          &tmp2,
60                          &tmp2);
61         if (retval < 5) /* Atleast 4 fields need to be read */
62         {
63            printf ("Error␣reading␣/proc/stat␣cpu␣field\n");
64         }
65
66         int user_delta ← fields[0]- old_fields[4*cpu+0];
67         int nice_delta ← fields[1]- old_fields[4*cpu+1];
68         int system_delta ← fields[2]- old_fields[4*cpu+2];
69         int idle_delta ← fields[3]- old_fields[4*cpu+3];
70
71         float current_load ← (float)(user_delta+nice_delta+system_delta)/
72         (float)(user_delta+nice_delta+system_delta+idle_delta);
73
74         EMA_cpu[cpu]← (current_load-EMA_cpu[cpu])*weight+EMA_cpu[cpu];
75
76         old_fields[4*cpu+0]← fields[0];
77         old_fields[4*cpu+1]← fields[1];
78         old_fields[4*cpu+2]← fields[2];
79         old_fields[4*cpu+3]← fields[3];
80     }
81 return EMA_cpu;
82 }
```

System Limits

The listing bellows provides the changes that need to be performed in the Unix file */etc/security/limits.conf* with root privileges, to change the maximum amount of open file descriptors and the maximum pipe size that a process can have.

```
1  # /etc/security/limits.conf
2  #
3  #Each line describes a limit for a user in the form:
4  #
5  #<domain> <type> <item> <value>
6  
7  // change the maximum number of open file descriptors
8  * soft nofile [value]
9  * hard nofile [value]
10 
11 //change the POSIX msg queue size
12 * - msgqueue [value] eg:819200
```

These changes only become active after reboot. However, after performing the changes, it is possible to enforce them through the *ulimit* command at run time.