Redis vs. Memcached

Andreas Pfurtscheller*
Aly Kamel*
{andreas.pfurtscheller,aly.kamel}@tum.de
Technical University of Munich

ABSTRACT

Today, almost 60% of the world's population actively use the Internet, with billions of users communicating with each other in real-time across the globe. In order to reach this state which we find ourselves in, it took numerous new and further developments in the field of Internet technology. In this paper we will examine the subsystems for data storage in an internet-scale distributed web application and study the requirements which arise from this high number of users. We primarily focus on the two distributed memory caching systems Memcached and Redis, both of which can be implemented as one of the building blocks to meet these demands on the architecture of a web application. We start with a brief explanation of the basic principals of caching and the technologies it is based on, followed by an introduction into the architecture of both caching systems, how they can be used to speed up requests as well as to reduce and distribute load on the application. As with all databases, we also examine their consistency behavior and look at how they manage partitions, while evaluating both systems according to the CAP theorem. Eventually, we conclude this paper with a comparison between the two caching systems.

ACM Reference Format:
Andreas Pfurtscheller and Aly Kamel. 2019. Redis vs. Memcached. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The early days of the Internet with a limited number of users and rather basic server infrastructure have long passed since its rapid growth in the early 2000s. In order to cope with this constant increase in the number of users and devices, the traditional server infrastructure had to evolve to handle the sheer number of requests, resulting in the emergence of numerous novel technologies and architectures over the past years. The basic requirements placed on almost any popular web application can be broken down as follows:

(1) keep response times short, (2) manage data influx from various origins, (3) provide access to frequently-accessed data while allowing it to change, (4) enable scaling to handle millions of requests per second, and (5) keeping data consistent across different geographical regions [24].

These challenges concern the majority of the subsystems that make up a typical web application facing a significant amount of user requests. Since today's largest Internet companies are fundamentally data-driven, the focus is primarily on those subsystems that are responsible for storing, processing and providing this data. Traditionally, these are relational database management systems (RDBMSs) that act as the main repositories for both data and associations between them [5, 7], but also NoSQL document databases have gained popularity in recent years on the back of the emergence of Big Data [4, 13].

What those systems have in common is that both RDBMS and document databases can only handle a certain number of requests at a time. Scaling the database by adding replicas is merely a shortsighted solution, since this leads to either increased response times or to inconsistencies between the replicas, which is not acceptable for databases whose principal objective is to store data in a consistent manner [8, 26]. In addition, slave nodes (i.e. replicas) of relational databases are read-only to preserve consistency, so that writes can only be made to the master node. Thus, only read but not write operations can be distributed among different instances, eventually causing a write bottleneck on the master node [10].

These limitations caused a substantial change in the storage hierarchy of web applications through the introduction of an intermediate cache layer between the application server and the underlying database [10, 24]. Caches are implemented using in-memory key value stores (KVSs), providing very high performance for read operations on data, thus allowing to significantly improve request latency. Having web applications caching their precomputed data resulting from queries to traditional RDBMS and subsequently only querying the cache, this significantly reduces the number of reads on the underlying database [4].

With diminishing prices for main memory and the demand for solutions capable of improving the scalability of read operations, Memcached was the first distributed in-memory caching system to be developed in 2004 [10]. Since then, the number of people using the Internet increased by a factor of more than four, with over 4.4 billion active users in 2019 [22, 34]. This also put greater stress on existing caching solutions, which led to the further development of Memcached, through Facebook in particular [5]. However, Memcached did not remain the sole choice for too long, as it soon faced rather sophisticated competition with the introduction of Redis in 2009 [17]. Redis addresses the contemporary demands on caching systems and implements functionalities such as native data types, replication and clustering [29].

Thereby, both caching systems try to satisfy the requirements imposed on web applications we mentioned earlier in the introduction,

*Both authors contributed equally to this work.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

especially (1) to (4). In this paper we will discuss the fundamental facets of caching prior to examining Memcached and Redis. Following a brief introduction into both systems, we compare their architectures, the capabilities they provide as well as their respective application areas.

This paper is structured as follows. In Section 2, we introduce the fundamental technologies of NoSQL databases, key-value stores and caching upon which the presented systems are based. In the subsequent sections 3 and 4 we discuss the two caching systems Memcached and Redis in more detail, focusing on their architecture. Chapter 5 concludes the paper by comparing both systems and summarizing the contributions.

2 BACKGROUND

In this section, we elaborate on the fundamentals of caching and break down the technologies it is based upon, with key-value stores being the most notable one. Moreover, we provide definitions for essential terms such as ACID and BASE, two different properties for database transactions, as well as for the already mentioned CAP theorem, which also brings us to consistency behavior.

2.1 Transaction properties and consistency

In the global Internet, data has to be available within a very short amount of time at each geographic region it is requested, even if a subset of replicas, or even a whole site is not available at all [31]. This requires the data to be replicated across those different regions, so that communication latencies can be kept low. However, this is where traditional databases with strict transaction requirements like ACID (***A**tomicity, **C**onsistency, **I**solation, **D**urability*) begin to struggle, facing both decreased performance and availability in such an environment [3]. This is, because databases fulfilling the ACID transaction properties provide strong consistency using pessimistic replication techniques, which block access to replicas with stale data until they are up to date with the primary replica [31]. Whilst working with replicas running close to each other facing low communication latency (i.e. within one site), it becomes increasingly difficult with more replicas in a wide area.

2.1.1 CAP Theorem. When synchronizing replicated data, systems will have to make tradeoffs between consistency, availability and partition-tolerance. This was first observed by Rothnie and Goodman in 1977 [30] and became popular as Brewer's CAP theorem [6], which was proved by Gilbert and Lynch [11] later. The theorem states that a replicated, distributed database can meet at most two of consistency of replicas, availability of writes and partition tolerance. This means that they can only provide (1) consistency of available copies and write availability if there are no partitions (CA); (2) consistency of available copies during a partition with at most one partition available for writes (CP); or (3) write availability during a partition but inconsistencies of copies in different partitions (AP) [3, 11].

2.1.2 Eventual Consistency. With the need for high performance in today's highly-available, large-scale asynchronous systems, it is essential for them to handle network partitions and provide write availability. This is why weak mutual consistency criteria have aroused scientific interest, often referred to as optimistic replication [3, 31, 33]. This deploys new algorithms for replica coordination, which propagate changes in the background, discover conflicts after they happen and reach agreement on the final contents incrementally. This approach is also referred to as eventual consistency as it guarantees that the state of replicas will converge only eventually [27, 31].

This is a rather informal definition of eventual consistency. The work of Saito and Shapiro [31] provides a formal and much more precise definition:

> A replicated object is eventually consistent when it meets the following conditions, assuming that all replicas start from the same initial state.
> - At any moment, for each replica, there is a prefix of the [history] that is equivalent to a prefix of the [history] of every other replica. We call this a committed prefix for the replica.
> - The committed prefix of each replica grows monotonically over time.
> - All nonaborted operations in the committed prefix satisfy their preconditions.
> - For every submitted operation α, either α or $\overline{\alpha}$ [its abort] will eventually be included in the committed prefix.

This type of consistency is also contained in a weakened form of the ACID properties which were also defined by Brewer, namely BASE (***B**asically **A**vailable, **S**oft state, **E**ventual consistency*) [6]. Databases meeting these criteria (1) guarantee availability in terms of the CAP theorem, (2) are never in a certain state as it can change over time even without input, and (3) will only be consistent over time, given no system input [3, 11].

2.2 Key-Value Stores

While most traditional RDBMS are limited in performance and scalability due to the ACID requirements they comply with, database concepts like KVS implementing the weaker BASE criteria overcome these barriers, providing high availability while also maintaining performance. However, this shifts responsibility for handling data inconsistencies to the application consuming the data, which increases the complexity of their business logic [3].

The KVS are among the so-called NoSQL databases, which became particularly popular in recent years. While its definition "***N**ot **O**nly **SQL***" is not very descriptive regarding the capabilities of those systems, their core properties can be broken down to

(1) provide horizontal scalability, (2) partition data over multiple servers, (3) offer a callable interface or protocol, and (4) operate on weaker transaction criteria than ACID [8].

KVS are based on a rather simple data structure, which is a key-value index of all data, with the keys being strings and the values being arbitrary data depending on the implementation. This lightweight design allows faster query speeds than with conventional RDBMS [13] and allows partitioning of data across several servers by distributing those indexes.

Another consequence of this data structure is the modest number of supported commands. Among the most important ones are (1) `GET <key>`, (2) `SET <key> <value>`, (3) `DELETE <key>`, and

(4) FLUSH , which (1) return the value for the key "key", (2) set the value "value" for the key "key", (3) delete the value for the key "key", and (4) delete all key-value pairs stored in the database.

2.3 Caching

Throughout the introduction we already argued why an intermediate caching layer is needed within the data storage architecture of today's distributed web applications. Besides the already mentioned reduction of the database load, caching results of expensive computations can reduce CPU load and due to the lowered amount of database queries, also reduce the network load [35]. As a whole, this leads to faster response times of the application, fulfilling the first of the requirements on web applications [4, 8].

These fast response times are partially achieved through the use of KVS which are already faster than RDBMS on their own. For even greater increase in performance, KVS store their data in the volatile memory of the servers instead on the slower hard drives [23]. Since main memory is significantly more expensive than hard disks, it is not viable to cache the entire database of systems with several million users — something that would also slow down the cache.

To counteract cache clutter, KVS provide functionality allowing to set an expiration timestamp for key-value pairs, after which they will be invalidated and the allocated memory is freed [12].

2.3.1 Cache policies. If, however, the memory allocated to the cache is not sufficient, items must be removed using cache eviction policies [23, 35]. Choosing a sufficient policy is essential for maximizing the *cache hit* rate, since the cache will be inefficient otherwise. For instance, when a request to an item fails, also referred to as a *cache miss*, an extra call to the database is required to retrieve the data. This in its extreme form is called thrashing, which arises with a cache lacking sufficient capacity. Thereby, new items replace the already cached ones constantly, even if they are accessed frequently as well. This way, requests often result in a cache miss, which needs to be prevented [23].

One cache eviction policy is least recently used (LRU), which is also the one used by the most popular cache implementations [10, 28]. When using this policy, least recently requested items are evicted first from the cache. While it is easy to implement and efficient with uniform objects, it does not consider aspects like download latency of data [23]. Other strategies include least frequently used (LFU) and other sliding-window policies [23, 35].

2.3.2 Cache consistency. Another descriptive proper of caches is their consistency behavior. Since they are supposed to accelerate requests, a strong consistency within the system would drastically reduce response times, since it would have to wait for the synchronous replication to complete. For this reason, distributed caches build upon the BASE transaction criteria previously introduced [11, 35].

2.3.3 Cache strategies. Choosing the right strategy for how data is accessed is essential when aiming for an efficient cache deployment. There are several options to choose from, with the most notable ones being discussed in the following.

With the **Cache-Aside** strategy, the cache sits aside of the application and the database. When retrieving data, it first tries to fetch it from the cache. If this request results in a cache miss, the application will query it from the database, return it to the client and store the value in the cache. When data is updated in the database, the application has to either invalidate the corresponding cache entry, or also update the value in the cache. This approach works best for read-heavy workloads, with the system being resilient to cache failures [18, 21].

With **Read- and Write-Through** cache strategies, the cache sits in-line between the application and the database. When a request to the cache results in a miss, the data is loaded from the database, stored in the cache and returned to the application. Data is written to the cache first, which then populates the database, ensuring consistency with the database. While this strategy allows high read performance, it requires both the cache and the database to share the same data model [15, 18].

3 MEMCACHED

Memcached was developed in 2003 by Brad Fitzpatrick in order to scale up his own social media service LiveJournal [10]. It has been deemed a reliable way to speed up requests, which is why it gained popularity among developers. It is currently deployed by many large-scale cooperations including Facebook [24], Youtube [9] and Wikipedia [19]. Since Memcached is an open-source system, libraries for multiple languages such as Python, Java, C and PHP have been developed. Cloud hosting providers like Microsoft Azure [1] and Amazon Web Services [2] also offer it as a service on their systems.

3.1 Architecture

Memcached is based on a client-server architecture as shown in Figure 1. Multiple clients, like PHP web servers, use Memcached to cache their data in-memory [20]. The data is split into multiple KVS which are each contained in a separate Memcached server. A consistent hashing algorithm is applied to map the keys to servers, so that the clients know where to store and access them. Since Memcached uses client-side partitioning, Memcached servers do not need to communicate with each other. The connection protocol can be set to either UDP, TCP or Unix sockets. Memcached allows up to 250B long keys and up to 1MB long string values. Keys can be optionally set with an expiration date, otherwise they will not expire.

Increasing the storage of Memcached means more keys can be kept in the cache which leads to fewer cache misses and higher performance. Adding a new server not only enlarges the overall capacity of the system, but also helps reduce CPU load on individual nodes. Since a consistent hashing algorithm is used, only K/N keys require a remap on average, with K being the number of keys and N the number of hash slots [14].

3.2 Commands

Most commands run in $O(1)$ and get executed atomically. However, if multiple commands are issued simultaneously, race conditions may occur. A possible option to avoid that is by using the CAS (Check and Set) operation. When a client reads a value and tries to update it, the CAS command could be used instead of the usual SET. As the name implies, CAS first checks if the value of the key to

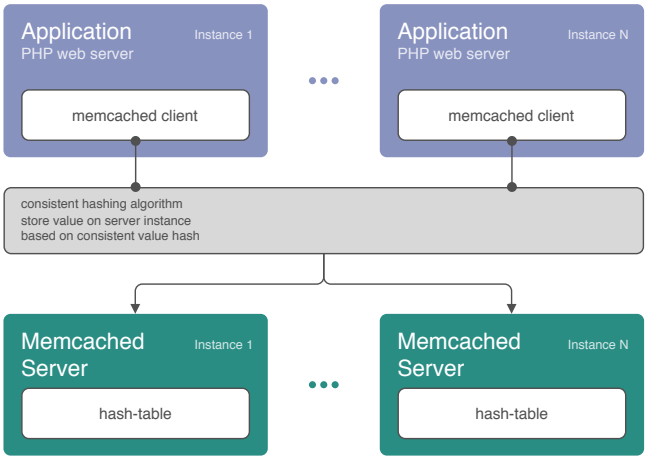

Figure 1: PHP clients deploying a Memcached cache

be updated has changed since it has last been read. The value gets only updated if no other client has altered the it. This ensures that an overwrite does not happen.

3.3 Storage retrieval

While Memcached is running, a crawler periodically scans all the keys and discards any that have expired. Additionally, whenever a client performs a `GET` operation on an expired key, its storage space gets reclaimed and the value is fetched from the database.

If there is no space remaining in the cache, an improved LRU algorithm is applied, in order to free up space. At first, the keys near the end of the tail are examined whether any of them has expired. One of the expired ones gets evicted and is replaced by the new key. If none of them have expired, the basic LRU algorithm explained in Section 2 is applied. With this inexpensive modification, the least recently used key can be kept in the cache, since it might still be in the current working set.

4 REDIS

Developer Salvatore Sanfilippo wanted to scale up his startup in 2009. He wanted a system, which allows (1) fast data access while (2) maintaining durability [25]. Another important feature for him was the possibility of using (3) complex data types as values [32]. For these reasons he decided to develop his own caching system, Redis (***Re**mote **Di**ctionary **S**erver).* Redis spread out since and got deployed by Twitter [36], Instagram [16] and Craigslist [37] among others. Redis clients are available to a great selection of languages [29]. Redis also provides other services such as a native messaging queue and the ability to execute Lua scripts, that will not get discussed in this paper though. In this section we elaborate on how Redis was able to reach the previously stated goals.

4.1 Persistence

The main use case of a cache is to achieve high read and write performance by storing frequently used data during the time the system is running. Redis expands on this idea by allowing a cache to persist the data it has previously accumulated, so that it does not have to get repopulated after shutdown. If the cache storage S is being fully capitalized, persisting data decreases the amount of cache misses by up to S at every system start. This would also allow Redis to overtake the role of a traditional database in some scenarios, since data can be permanently retained. Redis provides two options to persist data: *Snapshotting* and *AOF (Append-Only File)* mode. Both methods have their advantages, which is why they are often combined together. There are even plans to merge both modes into one in the future.

When running in *Snapshotting* mode, the system creates a snapshot of the system and is able to restore the data from it. This can be set up to occur periodically - by default every few minutes - and/or after a certain number of writes has been reached. The resulting *RDB file* contains all the information necessary to reconstruct the stored data in a compact manner. This allows it to be easily transported across data centers.

By using the *AOF-Mode*, the focus is on achieving even higher durability. Whenever a client issues a `SET` command, a log is created and appended at the end of the *AOF* file. By tracing back the file, the cached data can be reconstructed. If the maximum file size is approached, the AOF file gets rewritten in the background by removing all logs of a key except the most recent one. Since *"last writer wins"* all older logs are irrelevant. Journaling is a lot faster than creating snapshots, which is why it is possible to note down writes more frequently - by default every second. As a result, in the worse case scenario only one second of data updates are lost.

4.2 Transactions

Simple Redis commands run in under 1 millisecond. If a command fails, Redis will continue to process the subsequent ones, even though that may result into inconsistencies. In order to avoid that and maintain atomicity while executing a sequence of commands, Redis introduced transactions. A chain of operations wrapped between `MULTI` and `EXEC` gets treated as a single command. Transactions could then be used as building blocks to implement more abstract concepts such as critical regions.

4.3 Architecture

A basic Redis deployment works very similar to that of Memcached, as seen in Figure 2. The LRU algorithm is also utilized to manage storage eviction. One main difference is that values stored in the hash tables are not necessarily strings. Redis allows values to be strings, lists, sets, hashes, bitmaps and more. These types are provided with basic operations such as item insertion and removal, all of which implemented to run as fast as possible. Although types could be simulated in Memcached by using serialization, that would require more effort on the programmer's side.

Just like with Memcached, this architecture does not tolerate partition, since every server contains a different subset of the data. In the following, we will discuss approaches, which allow the system to be scaled up.

4.3.1 Replication. Whenever a system needs to be scaled up, the first intuition is to increase the number of servers. However, this would only work if the requests are equally distributed among the servers. If a small dataset is being heavily accessed, a single

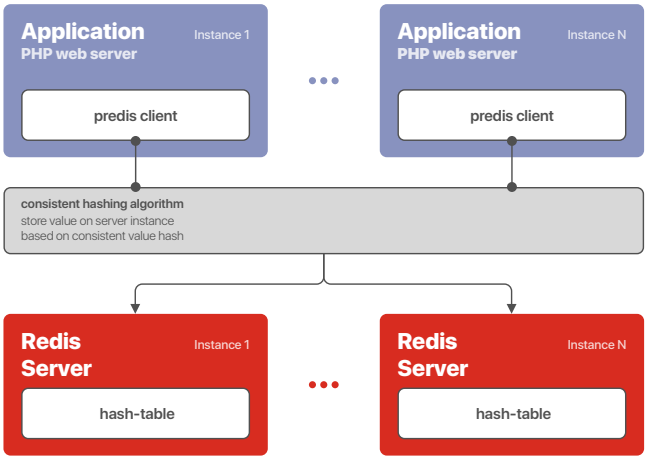

Figure 2: PHP application deploying a Redis cache

node will end up being the bottleneck of the system, no matter how much data is split across servers.

Redis resolves these problems by introducing redundancy to its design, as shown in Figure 3. The main server nodes are declared as `Masters` and are each assigned with a number of `Slaves` which contain a copy of their data. Whenever a Master node fails, one of its Slaves gets promoted to Master and replaces it. In order to maintain consistency, data can only be written to the Master node, which then asynchronously replicates its data to its Slaves. The number of replications the Master has to perform is equal to the amount of its Slaves multiplied by the number of write operations issued to it. Since this would scale up poorly, it is possible to expand the hierarchy downwards by adding `Subslaves` to the systems. Each node is only responsible for the slaves directly beneath it, so that way a fraction of the load is lifted from the Master.

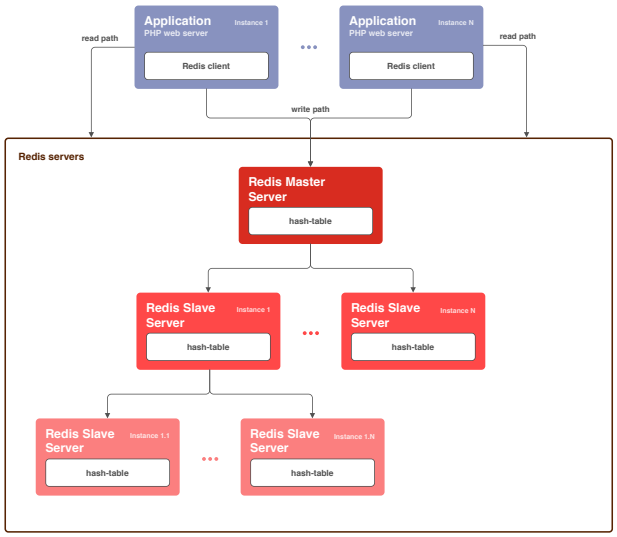

Figure 3: PHP application deploying a Redis cache with replication

4.3.2 Clustering. As previously mentioned, replication boosts read performance. This, however, occurs at the cost of the write performance, since data has to be constantly replicated. Furthermore, if only one Master is being used, it will get quickly overloaded by `SET` requests. For that reason, data partitioning is extremely important.

In order to shard data effectively, Redis has introduced the principle of clustering. There, server nodes are connected together to build the Cluster (Figure 4). Each node inside owns a subset of 16384 hash slots. Each key gets mapped to a certain slot by using a cyclic redundancy check (CRC16) hash. Whenever a client issues a command, one of the cluster nodes redirects it to the node responsible for that key. Nodes communicate via gossip protocol, which means there is no central router that the whole system depends on. The Cluster works best with between three and 1000 nodes.

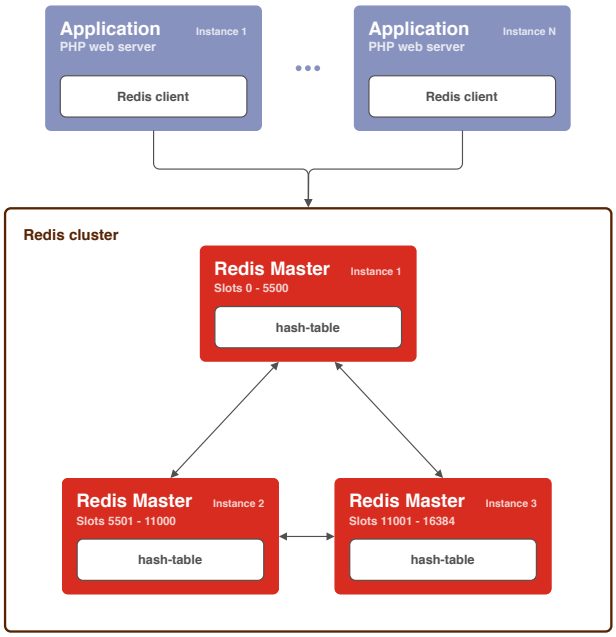

Figure 4: PHP application deploying a Redis Cluster (without replication)

With this basic architecture, the Cluster relies on all nodes being available, since each of them contains a different segment of the data set. This means all nodes are single points of failure (SPOF) and if one of them fails, the whole system collapses. For that reason, clustering is usually combined with replication (Figure 5). Nodes receive incoming requests and redirect them to either a Master node or one of its Slaves, if a `GET` command was issued. With that we eliminate the disadvantages of clustering and replication while maintaining all their benefits.

5 SUMMARY

To conclude our paper, we analyze Memcached and Redis per Brewer's CAP theorem, which was explained above. Then we will compare them with each other and reflect on their importance in large-scale internet distributed systems.

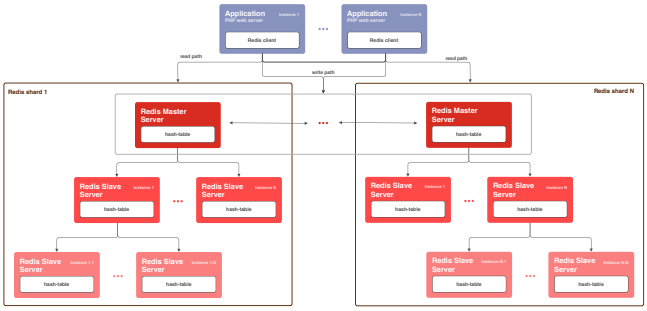

Figure 5: PHP application deploying a Redis Cluster

According to the CAP theorem, both Memcached and Redis deployed without data replication would be CA systems. Strong consistency is achieved since only one instance of each key-value pair exists. Otherwise, if Redis Cluster with replication is used instead, we would have a system with only partition tolerance to a certain extent. If a Master fails and has no available Slave to replace him, the system will crash. Also, a replacement requires some time, making the system only basically available. After write requests Slaves need to get updated, which is why this system is eventually consistent, because replication occurs asynchronously.

By using Memcached or Redis the goals listed in the introduction can be reached. While Memcached is built to be as simple as possible, Redis provides a lot of features to help developers customize their system based on their individual needs. Memcached can be set up to include replication and clustering, however these would need to be manually developed, like what Facebook did [24].

Currently, near real-time responses on the Internet are of upmost importance in order to satisfy users, which is why any distributed system with a large enough user base requires a plan to achieve high performance, which could be solved by a caching system, as we have demonstrated in this paper.

REFERENCES

[1] 2014. Memcached Cloud available in the Azure Store. https://azure.microsoft.com/en-in/updates/memcached-cloud-available-in-the-azure-store/

[2] Jeff Barr. 2011. Amazon ElastiCache – Distributed In-Memory Caching. https://aws.amazon.com/blogs/aws/amazon-elasticache-distributed-in-memory-caching/

[3] Philip A. Bernstein and Sudipto Das. 2013. Rethinking Eventual Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 923–928. https://doi.org/10.1145/2463676.2465339

[4] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have Your Data and Query It Too: From Key-Value Caching to Big Data Management. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 239–251. https://doi.org/10.1145/2882903.2904443

[5] Dhruba Borthakur. 2013. Petabyte Scale Databases and Storage Systems at Facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1267–1268. https://doi.org/10.1145/2463676.2463713

[6] Eric A. Brewer. 2000. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 7–. https://doi.org/10.1145/343477.343502

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[8] Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* 39, 4 (May 2011), 12–27. https://doi.org/10.1145/1978915.1978919

[9] Cuong Do. 2007. YouTube Scalability. (06 2007). https://www.youtube.com/watch?v=w5WVu624fY8 Seattle conference on scalability.

[10] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[11] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[12] Parikshit Gopalan, Howard Karloff, Aranyak Mehta, Milena Mihail, and Nisheeth Vishnoi. 2002. Caching with Expiration Times. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 540–547. http://dl.acm.org/citation.cfm?id=545381.545454

[13] Jing Han, Haihong E, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *Proc. 6th Int. Conf. Pervasive Computing and Applications*. 363–366. https://doi.org/10.1109/ICPCA.2011.6106531

[14] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. 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)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[15] Iqbal Khan. 2018. Using Read-through and Write-through in Distributed Cache. https://dzone.com/articles/using-read-through-amp-write-through-in-distribute

[16] Mike Krieger. 2011. Storing hundreds of millions of simple key-value pairs in Redis. https://instagram-engineering.com/storing-hundreds-of-millions-of-simple-key-value-pairs-in-redis-1091ae80f74c

[17] Reuven M. Lerner. 2010. At the Forge - Redis. *Linux Journal* (2010). https://www.linuxjournal.com/article/10836.

[18] Umer Mansoor. 2017. Caching Strategies and How to Choose the Right One. https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/

[19] MediaWiki. 2019. Manual:Memcached — MediaWiki, The Free Wiki Engine. https://www.mediawiki.org/w/index.php?title=Manual:Memcached&oldid=3250276

[20] Memcached Development Team. 2019. Memcached Wiki. https://github.com/memcached/memcached/wiki

[21] Microsoft Azure. 2018. Cache-Aside pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside

[22] Julia Murphy and Max Roser. 2019. Internet. *Our World in Data* (2019). https://ourworldindata.org/internet.

[23] Pranay Nanda, Shamsher Singh, and GL Saini. 2015. A Review of Web Caching Techniques and Caching Algorithms for Effective and Improved Caching. *International Journal of Computer Applications* 128, 10 (2015), 41–45.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 385–398. http://dl.acm.org/citation.cfm?id=2482626.2482663

[25] Jordan Novet. 2016. A conversation with Salvatore Sanfilippo, creator of the open-source database Redis. *VentureBeat* (June 2016). https://venturebeat.com/2016/06/19/redis-creator/

[26] Fatma Özcan, Nesime Tatbul, Daniel J. Abadi, Marcel Kornacker, C. Mohan, Karthik Ramasamy, and Janet Wiener. 2014. Are We Experiencing a Big Data Bubble?. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1407–1408. https://doi.org/10.1145/2588555.2618215

[27] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition*. Springer. https://doi.org/10.1007/978-1-4419-8834-8

[28] Matti Paksula. 2010. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science* (2010).

[29] Redis Development Team. 2019. Redis Documentation. https://redis.io/documentation

[30] James B. Rothnie and Nathan Goodman. 1977. A Survey of Research and Development in Distributed Database Management. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3 (VLDB '77)*. VLDB Endowment, 48–62. http://dl.acm.org/citation.cfm?id=1286580.1286585

[31] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *ACM Comput. Surv.* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[32] Salvatore Sanfilippo. 2009. Redis. https://news.ycombinator.com/item?id=494649

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[34] Statista Research Department. 2019. Global digital population 2019. https://www.statista.com/statistics/617136/digital-population-worldwide/
[35] Jia Wang. 1999. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Comput. Commun. Rev.* 29, 5 (Oct. 1999), 36–46. https://doi.org/10.1145/505696.505701
[36] Yao Yu. 2014. Scaling Redis at Twitter. https://www.youtube.com/watch?v=rP9EKvWt0zo
[37] Jeremy Zawodny. 2011. Redis Sharding at Craigslist. https://blog.zawodny.com/2011/02/26/redis-sharding-at-craigslist/