The Interdisciplinary Center, Herzliya

Efi Arazi School of Computer Science

M.Sc. program - Research Track

Load Balancing Memcached Traffic Using SDN

By

Idan Moyal

M.Sc. dissertation, submitted in partial fulfillment of the requirements

for the M.Sc. degree, research track, School of Computer Science

The Interdisciplinary Center, Herzliya

December 2016

This work was carried out under the supervision of Prof. Anat Bremler-Barr from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.

Acknowledgements

This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007 2013)/ERC Grant agreement no 259085.

Abstract

Memcached is an in-memory key-value distributed caching solution, commonly used by web servers for fast content delivery. Keys with their values are distributed between Memcached servers using a consistent hashing technique, resulting in an even distribution (of keys) among the servers. However, as small number of keys are significantly more popular than others (a.k.a., hot keys), even distribution of keys may cause a significantly different request load on the Memcached servers, which, in turn, causes substantial performance degradation.

Previous solutions to this problem require complex application level solutions and extra servers. In this paper, we propose *MBalancer*–a simple L7 load balancing scheme for Memcached that can be seamlessly integrated into Memcached architectures running over *Software-Defined Networks (SDN).* In a nutshell, *MBalancer* runs as an SDN application and duplicates the hot keys to many (or all) Memcached servers. The SDN controller updates the SDN switches forwarding tables and uses SDN ready-made load balancing capabilities. Thus, no change is required to Memcached clients or servers.

Our analysis shows that with minimal overhead for storing a few extra keys, the number of requests per server is close to balanced (assuming requests for keys follows a Zipf distribution). Moreover, we have implemented *MBalancer* on a hardware-based OpenFlow switch. As *MBalancer* offloads requests from bottleneck Memcached servers, our experiments show that it achieves significant throughput boost and latency reduction.

Contents

Chapter 1

Introduction

Memcached is a very popular general-purpose caching service that is often used to boost the performance of dynamic database-driven websites. Nowadays, Memcached is used by many major web application companies, such as Facebook, Netflix, Twitter and LinkedIn. In addition, it is offered either as a *Software-as-a-Service* or as part of a *Platform-as-a-Service* by all major cloud computing companies (e.g., Amazon ElastiCache [3], Google Cloud Platform Memcache [10], Redis Labs Memcached Cloud [21]).

Memcached architecture is depicted in Figure 1.1. Popular data items are stored in the RAM of Memcached servers, allowing orders of magnitude faster query time than traditional disk-driven database queries. Data items are stored in Memcached servers by their keys, where each key is linked to a single Memcached server, using a consistent hashing algorithm. Therefore, all Memchached clients are using the same Memcached server to retrieve a specific data item. Consistent hashing algorithm ensures an even distribution of keys, but it does not take into account the number of Memcached `get`

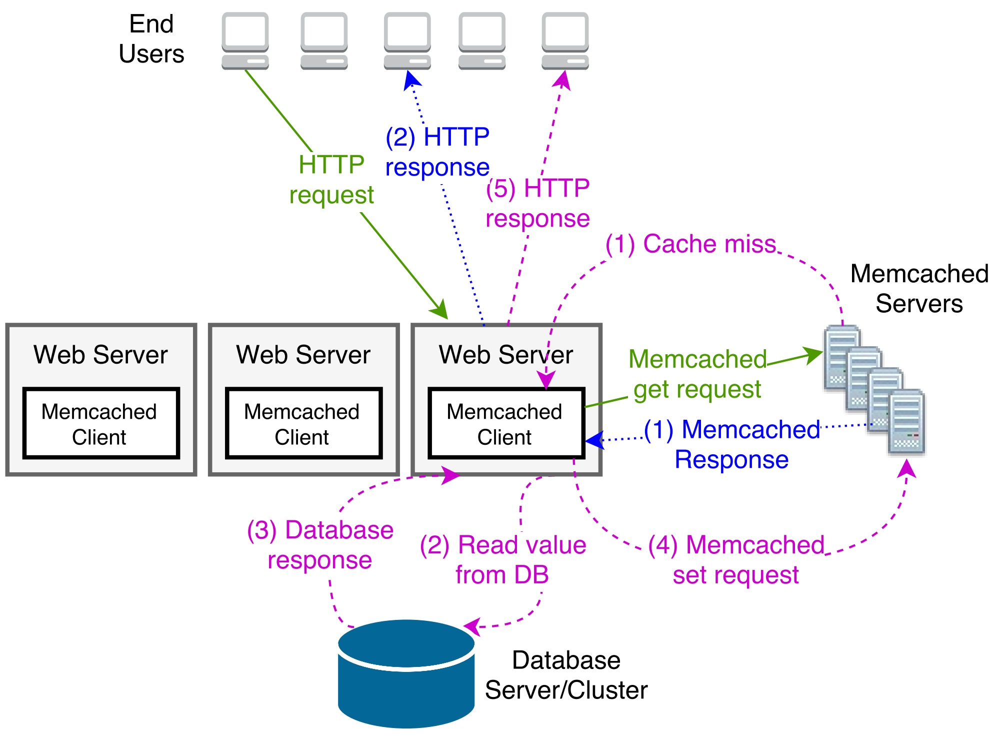


Figure 1.1: Memcached in a simple web server architecture.

requests to their corresponding data item (namely, the *key load*). It is well-known that web-items in general, and data elements stored in Memcached in particular, follow a Zipf distribution [2, 7, 9], implying that some data items are much more popular than others (a.k.a. *hot-keys*). This results in an imbalanced load on the Memchached servers, which, in turn, results in a substantial overall performance degradation.

The *hot-keys problem* is well-known in Memcached deployments and several tools to detect hot keys are available (e.g., Etsy's `mctop` [17] and Tumblr's `memkeys` [23]). Once the hot-keys (or the loaded servers) are detected, common solutions include breaking down the popular data item to many sub-items or to replicate the entire heavily-loaded Memcached server and manage these replications using a proxy.

In this paper, we take an alternative approach and propose *MBalancer*, a simple L7 load-balancing scheme for Memcached. *MBalancer* can be seamlessly integrated into existing Memcached deployments over Software-Defined-Networks (SDN). Unlike previous solutions, it does not require either a cooperation from the Memcached client (or developer) or additional servers.

In a nutshell, *MBalancer* identifies the hot keys, which are small in number. Then, *MBalancer* duplicates them to many Memcached servers. When one of the SDN switches identifies a Memcached `get` request for a hot key, the switch sends the packet to one of the servers using its readily-available load balancing capabilities (namely, OpenFlow's *select groups* [18, Section 5.10]).

SDN switches are based on a per-packet match-action paradigm, where fixed bits of the packet header are matched against forwarding table rules to decide which action to perform on the packet (e.g., forward to specific port, rewrite header, or drop the packet). *MBalancer* uses specific locations (namely, a fixed offset) in the packets' Layer 7 header, in which Memcached's keys appear. While such a matching is not supported by OpenFlow 1.5 (which restricts the matching to L2-L4 header fields), many SDN switch vendors today extend the matching capabilities to support matching in fixed location in the payload [1]. Moreover, P4 [6] recently proposed a method for dynamically configuring the header parsing, allowing for even greater control over how matching is performed and on which parts of the header. Thus,

[1]This is in contrast to the complex general task of Deep Packet Inspection, which searches for a signature that may appear anywhere in the data.

MBalancer can also be implemented over P4 switches.

We have evaluated *MBalancer* both in simulations and in experiments, and show that in practice about 10 key duplications suffice for gaining a performance boost equivalent to adding 2-10 additional servers (the exact number depends on the specific settings). Moreover, we have shown that smartly duplicating the keys to half of the servers yields almost the same results to duplicating the keys to all the servers. In contrast, we show that moving keys between servers (without duplication) almost never helps.

We have implemented *MBalancer* and run it in a small SDN network, with a NoviFlow switch that is capable of matching fixed bits in the payload. Our experiments show balanced request load and overall throughput gain that conforms to our analysis. Furthermore, *MBalancer* significantly reduces the average latency of requests.

Chapter 2

Memcached Preliminaries

One of the main reasons Memcached is so popular is its simple, client-server–based architecture (see illustration in Figure 1.1): Memcached servers are used as a cache, storing in their memory the latest retrieved items. Memcached clients are an integral part of the web server implementation, and their basic operations are `get key`, which retrieves an object according to its key; `set key,value`, which stores the pair $\langle key, value \rangle$ in one of the servers; and `delete key`, which deletes a key and its corresponding value. Every Memcached client is initialized with a list of n Memcached servers and a consistent hashing function, *hash*; thus every Memcached request with key k is sent to server number $hash(k)$. If key k is not found in server $hash(k)$, a cache miss is encountered, and the Memcached client reads the data from the database and performs a `set` request to store the $\langle key, value \rangle$ pair in that server for future requests. When the Memcached server depletes its memory, the least recently used data item is evicted. In this basic solution, each data item is stored in exactly one server, and data items are evenly distributed among the servers.

Memcached's protocol is simple and ASCII based (it also supports a binary protocol). For instance, `get` requests are structured as follows: "get <key>\r\n", where the key is always in a fixed location in the request structure, starting from the fifth byte, and thus can be easily identified, as in, for example, "get shopping-cart-91238\r\n". We leverage this structure in our solution. The keys in Memcached are determined by the developer who is using the Memcached system, and its size is bound by 250 bytes. The value contains up to 1MB of data.

Chapter 3

Evaluation of Hot Key Problem and Remedies

Recall that keys are distributed over the Memcached servers using a consistent hashing algorithm, which assigns a key to a server uniformly at random. However, as the load on the keys follows a Zipf distribution, the overall load on the Memcached server is not uniform.

Formally, let n be the number of servers and N be the number of keys. The load on the i-th most popular key, denote by $w_i = \Pr[\text{data item has key } i]$ is $1/(i^\alpha \cdot H_N)$, where α is a parameter close to 1 (in the remainder of the paper we set $\alpha = 1$) and $H_N \approx \ln N$ is the N-th Harmonic number. As before, let *hash* be the hash function that maps keys to servers. Thus, the load on server j is $load(j) = \sum_{\{i|hash(i)=j\}} w_i$.

We measure the expected *imbalance factor* between the servers, namely the ratio between the average load and the maximum load. Note that the imbalance factor is equivalent to the throughput when the most loaded server

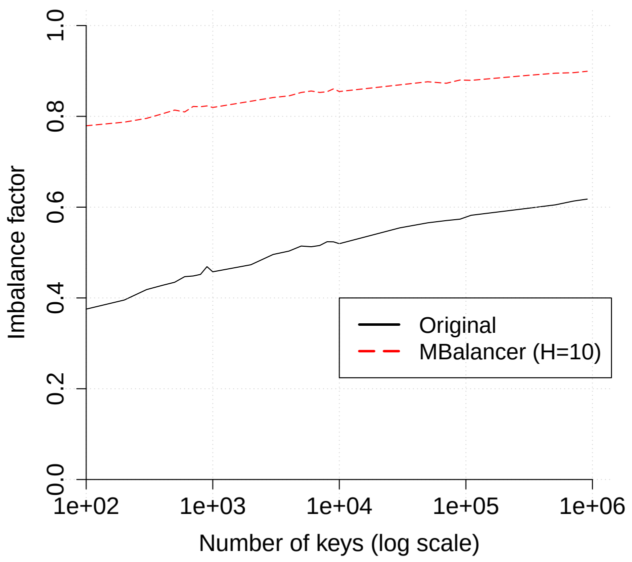


Figure 3.1: The imbalance factor as a function of the number of keys (for 10 servers).

(say, server k) becomes saturated and the rest of the servers process requests proportionally to server k's load:

$$\frac{1}{n}\sum_{j=1}^{n}\frac{load(j)}{load(k)} = \frac{1}{load(k)}\frac{\sum_{j=1}^{n} load(j)}{n}.$$

In Section 5, we show that the imbalance factor indeed corresponds to the obtained throughput.

We have investigated the impact of the number of keys and the number of servers on the imbalance factor through simulations.[1] Figure 3.1 shows that given a fixed number of servers (in this figure, 10 servers which are common in websites deployments), the imbalance factor grows logarithmically in the number of keys. As the imbalance factor runs between 40%−60%, it implies

[1]While the problem can be modelled as a weighted balls and bins problem [5], to the best of our knowledge there are no tight bounds when the weights are distributed with Zipf distribution.

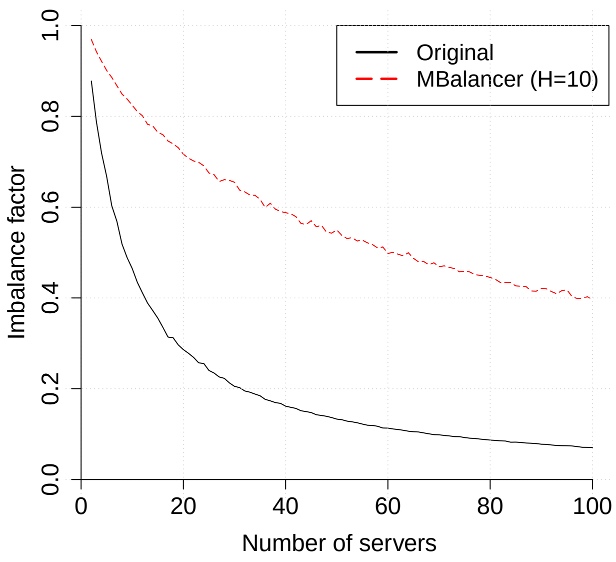


Figure 3.2: The imbalance factor as a function of the number of servers (for 1000 keys).

that half of the system resources are idled due to this imbalance. The problem becomes even more severe as the number of servers increases (see Figure 3.2). Intuitively, this is due to the fact that even if the heaviest key was placed on a server by itself, its weight is still significantly larger than the average load. This also rules out solutions that move data items between servers without duplications.

Notice that Figures 3.1 and 3.2 show also that *MBalancer*, whose specific design will be explained in Section 4, achieves substantial improvement of imbalance factor, where only the 10 most popular keys are being treated.

3.1 Comparing with Other Solutions

Facebook suggests to solve the hot key problem by replicating the busiest servers [19]. The solution is implemented by placing Memcahced proxies

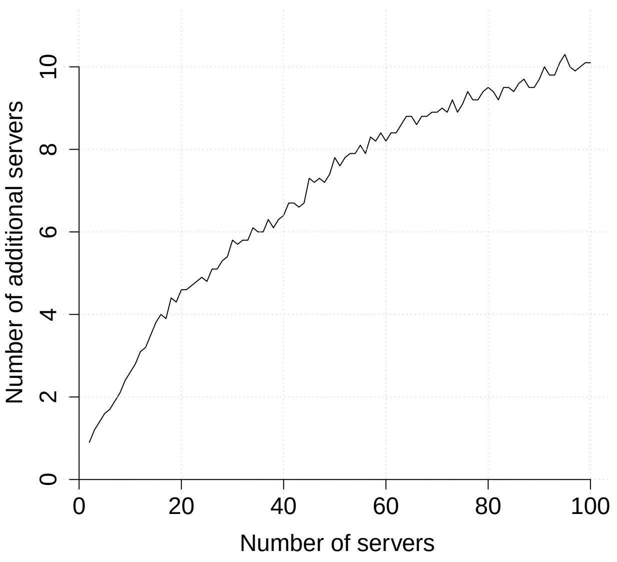


Figure 3.3: The number of additional servers (e.g., as suggested by Facebook [19]) required to achieve the same imbalance factor of MBalancer with 10 hot keys and 1000 keys.

between the clients to the servers, that distribute the load between the replicated servers. Figure 3.3 shows the number of extra servers required, so that this solution will achieve the same imbalance factor of *MBalancer*. Clearly, this solution is more expensive in CAPEX and requires extra software.

In [15], an optimized distributed storage system is described, which handles skewed workloads (zipf requests distribution) using requests monitoring, scheduling and caching. It is described how the system outperforms a selective caching [4] (hot items replication) solution by performing load balancing between shards/servers, caching hot items in memory and making sure data is always retrieved from the fastest storage available. However, the system is a general purpose data base which in most cases may not be suitable as an alternative for Memcached. Moreover it does not specifically relate and handle the network bottleneck as *MBalancer* does.

Other suggestions are to manually change the application logic [12, 17]. For instance, break down the data of a hot key into smaller pieces, each assigned with its own key. This way each piece of data might be placed in a different Memcached server (by hashing the new key), implying several servers will participate in the retrieval of the original data, item. However, this solution is complex, as it requires extra logic in the web server for splitting data items, and for writing and reading them correctly from Memcached.

MBalancer deals with improving the performance of Memcached traffic and thus it differs from general key-value storage system designs that aim at improving the storage system performance by adding cache nodes. Specifically, recent system designs [16], which dealt with the general case of key-value storage systems, use SDN techniques and the switch hardware to enable efficient and balanced routing of UDP traffic between newly-added cache nodes and existing resource-constrained backend nodes. Despite the similarity in using the switch hardware for load balancing, these system designs come to solve a different problem, the designs are not geared for Memcached traffic and therefore involve packet header modifications, and their analysis [8] is based on a general cache model with unknown request distribution. *MBalancer*, on the other hand, does not require adding new nodes to the system (and, in fact, no change to either the client and server sides), no packet header modification (as it looks at Memcached header in L7), and its analysis is based on the fact that request distribution is zipf.

Chapter 4

MBalancer, overview and design

MBalancer partitions the Memcached servers to G groups of n/G servers each. In order to load-balance requests for hot keys, *MBalancer* duplicates each of the H most popular keys to all servers in one such group, where H is a parameter of the algorithm. For brevity, we will assume $G = 1$, and therefore, all hot keys are duplicated to all servers. This will be relaxed later in Section 4.4.

Notice that the Zipf distribution implies that even if we choose H to be small (e.g., 10 out of 1000 keys), we get a large portion of the requests (in this case, $\frac{H_{10}}{H_{1000}} = \frac{2.93}{7.49} = 0.39$). This implies that the memory required for duplicating the keys to all Memcached servers is small. Using this method, requests for hot keys are spread evenly over all Memcached servers, while the less popular keys are treated as in the original memory implementation (namely, they are mapped to a single server).

4.1 MBalancer Design

While *MBalancer* can be implemented using a proxy, we suggest an implementation using an SDN network *that does not require any software modification and, in particular, leaves both Memcached clients and Memcached servers unchanged.* The source code can be found in [1].

The *MBalancer* architecture is illustrated in Figure 4.1. For simplicity we first explain the solution where all the memcahced clients and servers are connected to a single switch. Later we explain how to relax this assumption to a general SDN network.

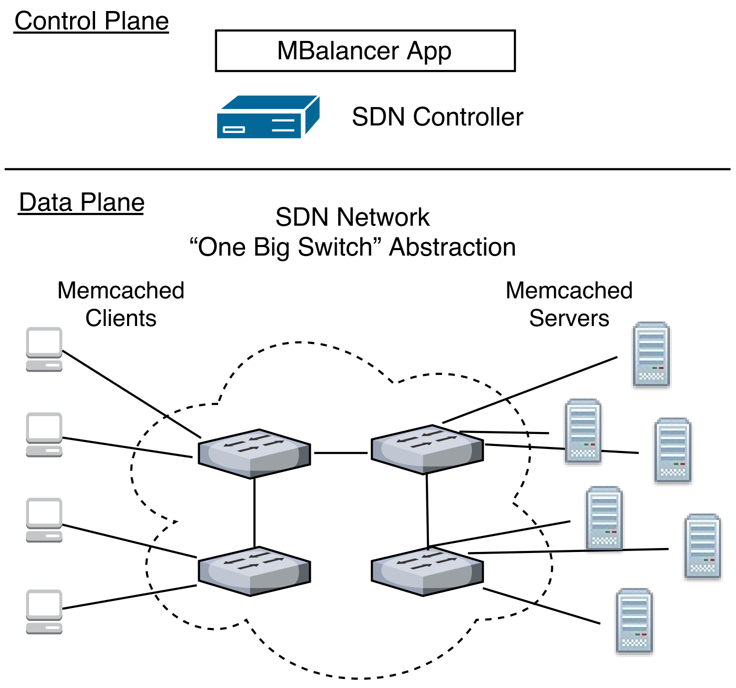


Figure 4.1: The MBalancer framework.

One of the advantages of SDN networks is a clear separation between their control and data planes. Our architecture involves actions in both planes.

Specifically, in the control plane, we have devised an SDN application, named *MBalancer*, that receives Memcahced traffic statistics from some monitoring component solution (e.g., `mctop` or `memkeys`), selects the hot keys, duplicates the keys to all servers, and configures the switches. In the data plane, the SDN switch matches hot keys from Memcached traffic and performs L7 load-balancing, as will be explained next. The rest of the keys are forwarded to their original destination by the Memcached clients as before. We note that our solution is only applicable for Memcached `get` requests over UDP. While TCP is the default in Memcached, UDP is usually preferred when the goal is better performance (cf. Facebook's solution [19]).

4.2 MBalancer Data Plane Configuration

For simplicity, we begin by explaining the switch configuration assuming that the H heaviest keys were already duplicated to all Memcached servers.

Figure 4.2 shows the switch configuration. Hot keys are identified by a flow table rule with payload matching for the hot key. The rules added to the switch rely on the fact that the key in a Memcached `get` request packet is in a fixed location (specifically, offset of 12 bytes) and ends with \r\n.

Once a packet with a hot key is identified, it is forwarded to an *OpenFlow group* of type ***`select`*** [20, Section 5.10]. Each such group contains a list of *action buckets* (in our case, each action bucket is responsible of redirecting packets to a different Memcached server). Each packet is processed by a single bucket in the group, based on a switch-computed selection algorithm that implements equal load sharing (e.g. hash on some user-configured tuple, simple round robin, or basing on the bucket weight).

In order to redirect a packet, the action bucket rewrites the destination MAC and IP addresses fields with the new addresses and forwards the packet to its new destination. We note that, in this situation, when the Memcached server responds to the client, the client receives a packet where the source IP address is different from the address it expects. In UDP, this is not a problem, as Memcached UDP clients are using unique identifiers (added to their `get` requests) to correlate between Memcached requests and responses, and are not validating the source IP address of the response packet.

Finally, an additional rule per hot key is added to the switch for capturing `set` requests for hot keys (update). These rules action is set to duplicate the packet and forward it to the *MBalancer* application in addition to the original Memcached server; note that this rule contains the original Memcached server destination (further explained in section 4.3). We note that, unlike `get` operations, `set` operations typically use TCP, and therefore, their destination addresses cannot be simply rewritten as before.

The total number of additional flow table entries is $2H + n$: one rule per hot key for `get` operation, one rule per hot key for `set` operation, and additional group configuration that requires buckets as the number of servers.

Multi-Switch SDN Network: In order to apply hot keys redirection rules in an SDN network that contains multiple switches, it is needed to place the rewrite rule only once at each path between each client and Memcached server. Then, the packet should be marked (with one bit) in order to avoid loops. Several methods have been proposed to cope with such issues in multi-switch SDN networks [13, 14] and can be also applied in our case.

4.3 MBalancer Application Tasks

MBalancer decides which are the H hot keys according to a monitoring information. It then decides if it should interfere with the regular Memcached activity. If so, it performs hot keys duplication to the Memcached servers and configures the flow table in the switch.

MBalancer application performs hot keys duplication using a Memcached client running within the *MBalancer* application. This client is configured to perform an explicit `set` operation to each of the relevant servers. The value of the hot keys is taken from the original Memcached server the key is stored in.[1]

As mentioned before *MBalancer* application is notified whenever a `set` operation for a hot key occurs and gets a copy of the packet. Then, it initiates a `set` operation to all relevant Memcached servers but the original one.[2]

4.4 MBalancer with More than One Group

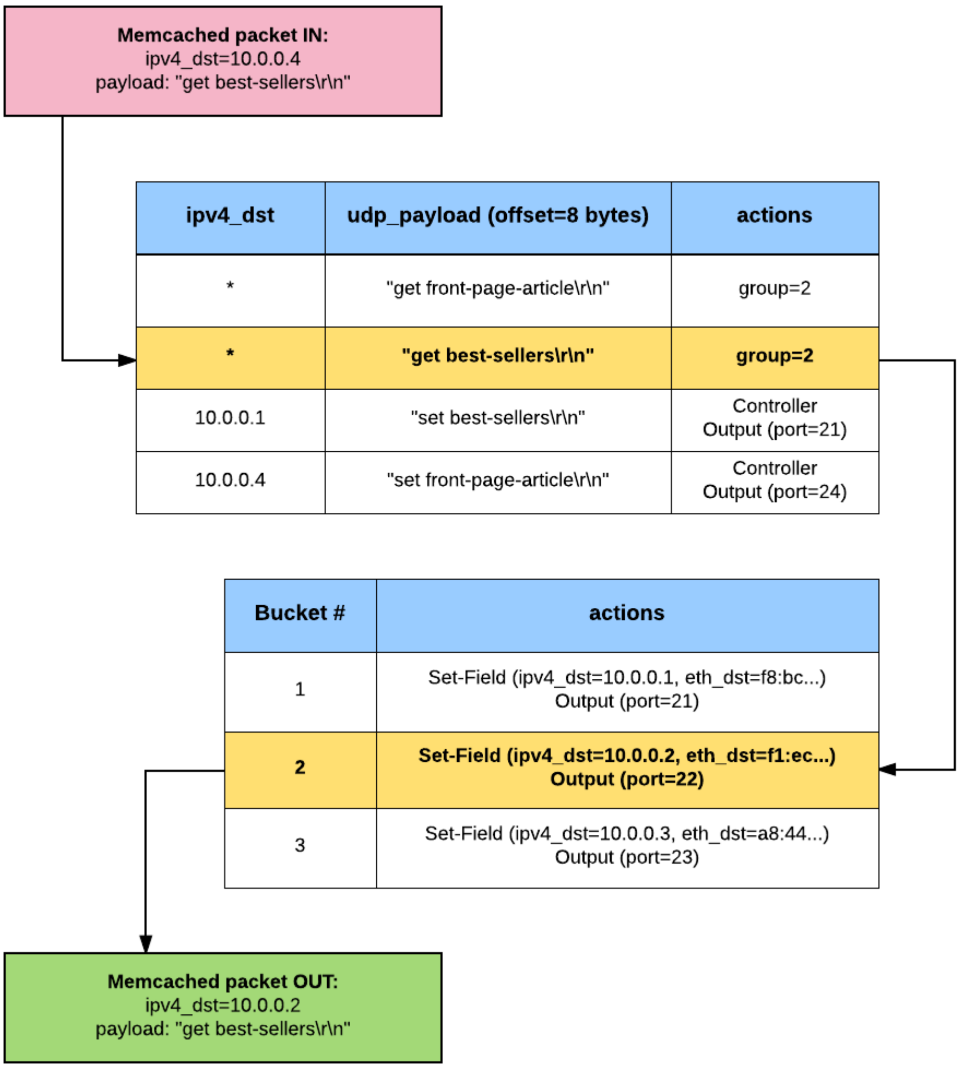
As mentioned before, *MBalancer* partitions the Memcached servers to G groups of n/G servers each. For $G > 1$, first a group is selected and then the hot key is duplicated to all the servers in this group. Notice that the load resulting from hot keys is the same for all servers within a group. Therefore, upon adding a hot key, the selected group is naturally the one with the least

[1]In arbitrary web application architecture, the value of a key can be a database record, or a computation result which might not be a database record. Thus, the value is taken from the Memcached server and not directly from the database.

[2]As the hot key rule that matches the original `set` operation contains also the original destination address, it will not be matched with the `set` operations initiated by *MBalancer* application, and therefore, will not creates loops.

such load. *MBalancer*'s data plane is changed to support G groups and not one, and forwarding the hot key to the right group in its corresponding rule. Because each Memcached server is in only one group, the total number of rules ($2H + n$) is unchanged. On the other hand, the memory (in the Memcached servers) used for duplicating hot keys is reduced by a factor of G.

Figure 4.3 compares the imbalance factor when using more than one group. Notice that the values for $G = 1$ and $G = 2$ are indistinguishable in this setting, implying one can save half of the memory overhead *for free.* For larger G, there is some imbalance factor penalty that decreases as the number of servers (or, equivalently, the size of the groups) increases.



Memcached packet IN:
ipv4_dst=10.0.0.4
payload: "get best-sellers\r\n"

ipv4_dst	udp_payload (offset=8 bytes)	actions
*	"get front-page-article\r\n"	group=2
*****	**"get best-sellers\r\n"**	**group=2**
10.0.0.1	"set best-sellers\r\n"	Controller Output (port=21)
10.0.0.4	"set front-page-article\r\n"	Controller Output (port=24)

Bucket #	actions
1	Set-Field (ipv4_dst=10.0.0.1, eth_dst=f8:bc...) Output (port=21)
2	**Set-Field (ipv4_dst=10.0.0.2, eth_dst=f1:ec...) Output (port=22)**
3	Set-Field (ipv4_dst=10.0.0.3, eth_dst=a8:44...) Output (port=23)

Memcached packet OUT:
ipv4_dst=10.0.0.2
payload: "get best-sellers\r\n"

Figure 4.2: Load-balanced packet flow in SDN switch

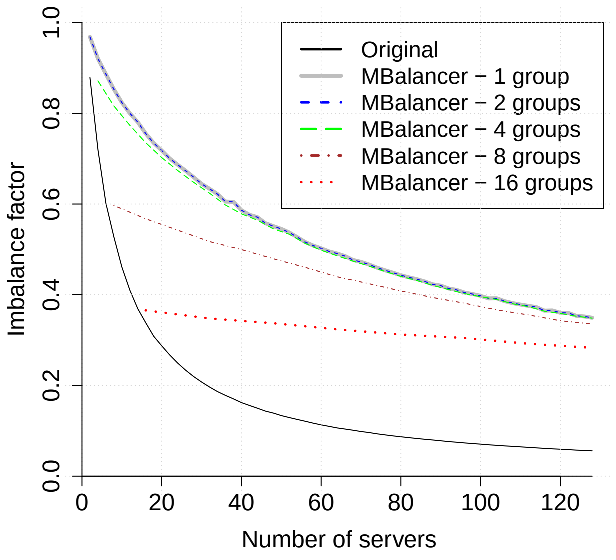


Figure 4.3: The imbalance factor as a function of the number of servers, for different number of *MBalancer* groups. All settings use $H = 10$ hot keys out of 1000 keys.

Chapter 5

Experimental Results

In this section we present the results of an experiment we have conducted over an SDN network. We have used 64 memcached clients, running on 2 servers that issued `get` requests for 1000 keys to two Memcached servers, which stored values of 100KB size.[1] The clients and the servers were connected to a single switch using a 1GBit/s interface. We used a NoviKit 250 SDN switch with payload matching capabilities[2]. The prototype of the *MBalancer* application was implemented in Python, as a Ryu [22] SDN controller application.

The `get` requests were issued over UDP, with a timeout of 1 second. In common web server implementations, `get` requests are blocking until a response has been received; only then another request is made. Thus, in our configuration if a web sever did not receive a response from Memcached it waited for 1 second before issuing the second request.

[1]We have used only two Memcached servers due to a limitation of equipment in our lab.

[2]We note that while Memcached's maximum key size is 256 bytes long, NoviKit 250 switch payload matching capabilities supports only keys up to 80 bytes long.

We ran an experiment with over 100,000 `get` requests 100 times. Since we had only two Memcached severs, we used a skewed hash function in order to create settings with different imbalance factors (recall that as the number of servers increases, the imbalance factor decreases). Figure 5.1 shows the effect of the imbalance factor on the (normalized) throughput, while Figure 5.2 shows the effect on latency. All experiments run with $H = 10$ and $G = 1$ (which in our case is only 1MB extra RAM). Clearly, it shows a significant boost in performance, matching the results of Section 3.

Moreover, as expected, the (normalized) throughput matches the imbalance factor. This is because the clients do not continue to the next request until a response for the previous request is received. Thus, the loaded servers become the system bottleneck and have a high impact on the throughput.

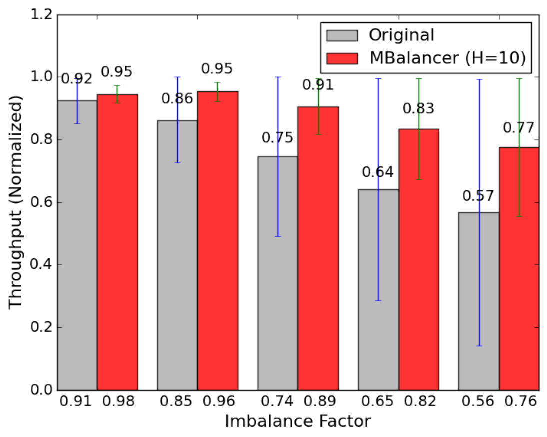


Figure 5.1: The average (normalized) throughput with and without *MBalancer*. The vertical line is the normalized throughput of the servers with minimum and maximum load.

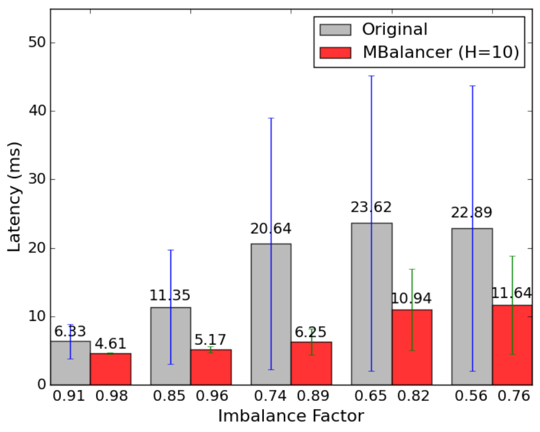


Figure 5.2: The average latency with and without *MBalancer*. The vertical line is the values of the servers with minimum and maximum average latency.

Chapter 6

Conclusions

In this paper, we presented *MBalancer*: an efficient framework for solving Memcached's hot keys problem by efficiently load balancing its requests. The framework also demonstrates the power of Software-Defined- Networking to solve higher level application issues.

In this paper we present *MBalancer*, a solution for the memcached hot-keys problem based on SDN. The goal of our solution is to show how SDN, specifically the ability to match packets according to their payload, can be used to solve higher level application issues. The key challenge of the research, was to define and formalize the hot keys problem and in the technical aspect, the deployment of OpenFlow rules for matching and redirecting memcached packets, in order to perform L7 load balancing in the switch. Using *MBalancer*, we demonstrate how it is possible to overcome the hot keys problem using SDN. We show that the problem is indeed solved when applying our solution to a web application architecture with memcached, where the problem occurs and we show that the overhead of our solution is minimal, meaning it can be applied to memcached clusters containing hundreds of servers and thousands of keys.

6.1 Limitations and Discussion

In this section we discuss a few limitations in our work:

1. Memcached get operations over UDP are used for achieving better performance, over TCP, with respect to the known limitations of UDP. In our research we demonstrate a UDP based solution, but the majority of memcached deployments in the world operate over TCP. We describe the challenge of implementing a TCP based solution in section 6.2.

2. Payload matching is not yet a part of the OpenFlow specification. *MBalancer* relies on a specific switch model, which features the ability to perform matching based on packet payload, using OpenFlow's experimenter API. We believe that payload matching will be added to next versions of the OpenFlow specification.

3. The memcached key length is limited by the switch's maximum table size. While memcached's maximum key size is 256 bytes, *MBalancer*, using the NoviKit 250 switch, can only support keys up to 80 bytes long. As switches improve, we believe new switches will support larger tables.

4. *MBalancer* does not support memcached multi-get operations, where a request contains several keys to retrieve in a single request. This is because memcached packets payload for such operations, are in the following format: "get <key-1> <key-2> <key-3>\r\n" Supporting multi-get operations in *MBalancer*, requires more advanced payload matching capabilities and possibly changes to the memcached protocol.

6.2 Future Work

In this section we list a few points we believe would be both beneficial and challenging to research:

1. ***MBalancer* and TCP**: Most memcached deployments in the industry are operating over TCP. We believe it would be a great benefit to make *MBalancer* support memcached over TCP. The challenge for supporting this capability, is the need to perform TCP splicing in order to hand-off a TCP connection from one server, the original destination, and the load balanced destination. TCP connection splicing as described in [11] is not enough, as after the first TCP connection hand-off, it is a problem to perform additional hand-offs because of the need to compensate the sequence and acknowledgement numbers and there's no way to keep a state for every connection in the switch.

2. ***MBalancer* Implementation Using P4 [6]**: P4, is a high-level language for programming protocol-independent packet processors. Using P4, it is possible to decoratively express how packets should be processed by the SDN switch. One of the required parts of a P4 program is a parser which parses incoming packets into user defined data structures. It is possible to include packets payload as a part of the parsing process and then match according to it. Using P4, it should potentially be possible to implement *MBalancer* in a generic way, which will work with every switch. We believe that P4 will turn to be the standard packet programming language and therefore would like to make *MBalancer* work using P4.

Bibliography

[1] Mbalancer source code: `https://github.com/idanmo/mbalancer`.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Int'l Conf. on Parallel and Distributed Information Systems*, pages 92–103, Dec 1996.

[3] Amazon. Amazon elasticache.

[4] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM, 2011.

[5] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, Dec. 2008.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[7] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 373–385, 2013.

[8] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM, 2011.

[9] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura. Caching memcached at reconfigurable network interface. In *IEEE FPL*, pages 1–6, 2014.

[10] Google. Google cloud platoform memcache.

[11] M. Hesham, H. Fang, and M. Sarit. Application-aware data plane processing in sdn. 2014.

[12] InterWorx. Locating Performance-Degrading Hot Keys In Memcached. `http://www.interworx.com/community/locating-performance-degrading-hot-keys-in-memcached/`.

[13] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software-defined networks. In *ACM CoNEXT*, pages 13–24, 2013.

[14] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *IEEE INFOCOM*, pages 545–549, 2013.

[15] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 60, 2016.

[16] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), Santa Clara, CA*, 2016.

[17] Marcus Barczak. mctop—a tool for analyzing memcache get traffic. Code As A Craft, December 2012. `https://codeascraft.com/2012/12/13/mctop-a-tool-for-analyzing-memcache-get-traffic`.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX NSDI*, pages 385–398, 2013.

[20] Open Networking Foundation. Openflow switch specification - version 1.5.0, December 2014.

[21] RedisLabs. Redislabs memcached cloud.

[22] Ryu. Ryu SDN framework, 2015. `http://osrg.github.io/ryu`.

[23] Tumblr. Tumblr memkeys. `https://github.com/tumblr/memkeys`.

תקציר

Memcached הוא פתרון זיכרון מטמון מבוזר השומר מידע בזיכרון ע״י מפתח וערכו. השימוש בו מאוד נפוץ בשרתי תוכן באינטרנט על מנת להחזיר תוכן באופן מהיר. המפתחות והערכים מפוזרים מעל מספר שרתי Memcached ע״י שימוש בפונקציית גיבוב עקבית, המבטיחה פיזור אחיד של המפתחות בין השרתים. מאידך, לרוב ישנם מפתחות שהם פופולריים יותר מאחרים בצורה ניכרת (ידועים כ-מפתחות חמים), ולמרות הפיזור האחיד של המפתחות בין השרתים, פיזור הבקשות בין השרתים אינו אחיד, וגורם להיווצרות עומס על שרתים מסוימים, אשר פוגע בתפקודם באופן משמעותי עד כדי בעיות ביצועים.

פתרונות קודמים לבעיה הצריכו פתרונות אפליקטיביים מסובכים ושרתים נוספים. בעבודה זו, אנו מציעים את MBalancer- פתרון איזון עומסים פשוט מעל השכבה השביעית עבור Memcached אשר ניתן להטמיע בקלות בארכיטקטורות Memcached שרצות מעל רשתות מונחות תוכנה, (SDN) Software Defined Networking. בקצרה, MBalancer רץ כאפליקציית SDN ומשכפל מפתחות חמים למספר שרתי Memcached (או כולם). בקר ה-SDN מעדכן את טבלאות החוקים במתגי ה-SDN ומשתמש ביכולות הקיימות ב-SDN לביצוע איזון עומסים. ולפיכך, אין צורך לשנות דבר בקליינטי ושרתי ה- Memcached.

ע״פ האנליזה שביצענו, אנו מראים שעם תקורה מינימלית עבור שמירת מספר מצומצם של מפתחות נוספים, מספר הבקשות לכל שרת קרוב למאוזן (בהנחה שהתפלגות הבקשות עבור המפתחות היא התפלגות Zipf). בנוסף, ממשנו את MBalancer במתג OpenFlow מבוסס חומרה. בהינתן ש-MBalancer מוריד את הצוואר בקבוק משרתי Memcached עמוסים, הניסוי שביצענו מראה שיפור משמעותי בתפוקה ובזמן תגובה מהשרתים.

עבודה זו בוצעה בהדרכתה של פרופ׳ ענת ברמלר בר מבי״ס אפי ארזי למדעי המחשב, המרכז הבינתחומי, הרצליה.

IDC HERZLIYA

Efi Arazi School of Computer Science

המרכז הבינתחומי בהרצליה

בית-ספר אפי ארזי למדעי המחשב

התכנית לתואר שני (M.Sc.) - מסלול מחקרי

איזון עומסים לבקשות Memcached ברשתות מונחות תוכנה

מאת

עידן מויאל

עבודת תיזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc. במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

דצמבר 2016