Load Balancing Memcached Traffic Using Software Defined Networking

Anat Bremler-Barr[†], David Hay[*], Idan Moyal[†], and Liron Schiff[‡]
[†]Dept. of Computer Science, the Interdisciplinary Center Herzliya, Israel. {bremler,moyal.idan}@idc.ac.il
[*]School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel. dhay@cs.huji.ac.il
[‡]Blavatnik School of Computer Sciences, Tel-Aviv University, Israel. schiffli@post.tau.ac.il

***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.

I. 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 [2], Google Cloud Platform Memcache [11], Redis Labs Memcached Cloud [28]).

Memcached architecture is depicted in Figure 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`

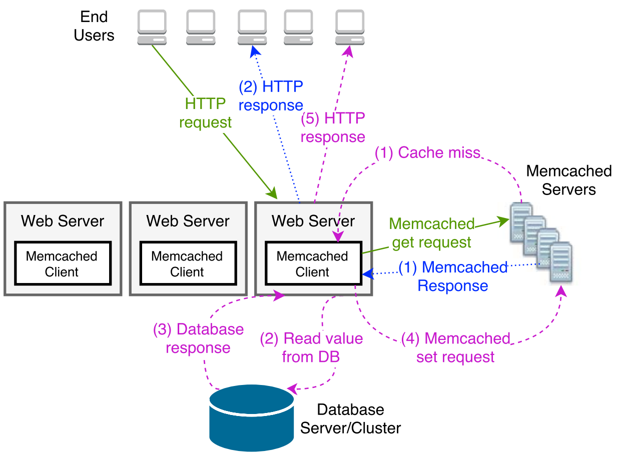


Fig. 1. Memcached in a simple web server architecture.

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 [1, 8, 10], 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` [20] and Tumblr's `memkeys` [32]). Once the hot-keys (or the loaded servers) are detected, common solutions include breaking down the popular data item to many sub-items or replicating the entire heavily-loaded Memcached server and managing 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

ISBN 978-3-901882-94-4 © 2017 IFIP

packet to one of the servers using its readily-available load balancing capabilities (namely, OpenFlow's *select groups* [21, Section 5.10].For P4 switches, we present equivalent load balancing program).

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 [5] 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, in this paper, we present implementations of *MBalancer* both over P4 switches and OpenFlow switches with the above-mentioned capabilities.

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.

II. Memcached Preliminaries

One of the main reasons Memcached is so popular is its simple, client-server–based architecture (see illustration in Figure 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.

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

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

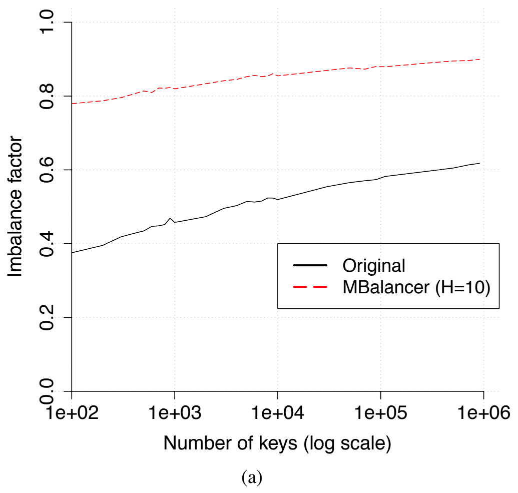
In Section V, we show that the imbalance factor indeed corresponds to the obtained throughput.

We have investigated the impact of the number of keys and the number of servers on the imbalance factor through simulations.[2] All results presented are the average of 100 runs. Figure 2(a) 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 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 2(b)). 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 2(a) and 2(b) show also that *MBalancer*, whose specific design will be explained in Section IV, achieves substantial improvement of imbalance factor, even when a only

[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.

[2]While the problem can be modelled as a weighted balls and bins problem [3], to the best of our knowledge there are no tight bounds when the weights are distributed with Zipf distribution.



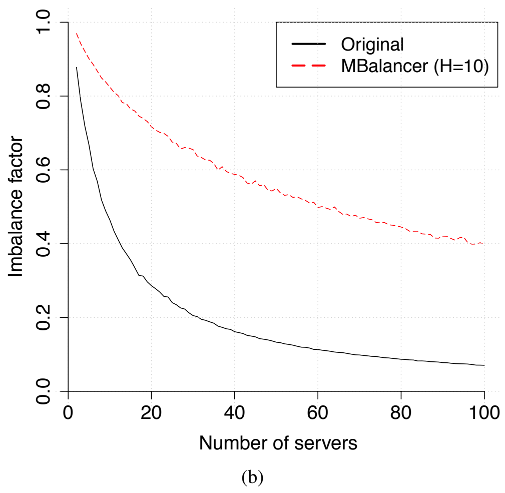


Fig. 2. The imbalance factor as a function of (a) the number of servers (for 1000 keys and 10 hot-keys), (b) the number of keys (for 10 servers and 10 hot-keys).

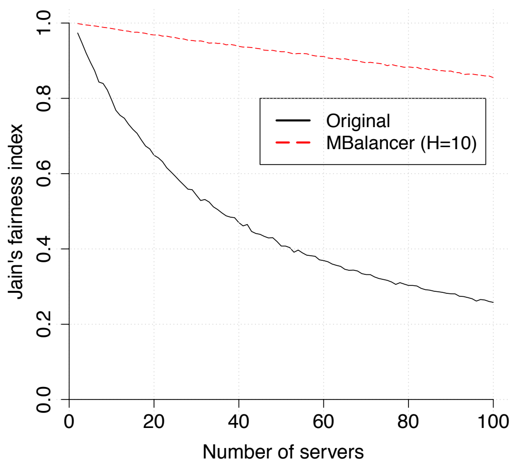


Fig. 3. Jain's fairness factor as a function of the number of servers (for 1000 keys and 10 hot keys). The closer Jain's fairness index to 1, the more balanced the load on the servers.

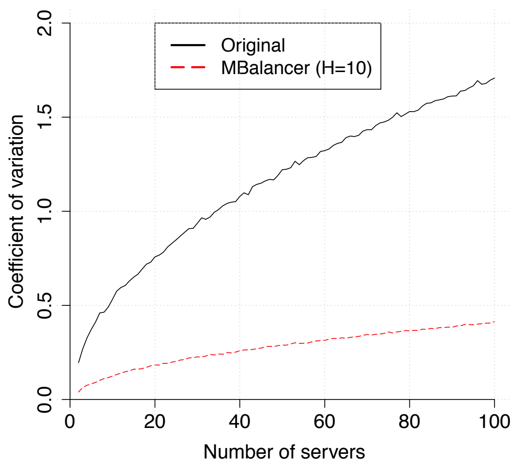


Fig. 4. Coefficient of variation as a function of the number of servers (for 1000 keys and 10 hot keys). The closer the coefficient of variation to 0, the more balanced the load on the servers.

the 10 most popular keys are being treated. The same holds also when other measures of fairness are considered: Figure 3 shows that *MBalancer* significantly improves *Jain's fairness index* [14], while Figure 4 shows the same improvement when the *coefficient of variation* is considered.

A. Comparing with Other Solutions

Facebook suggests to solve the hot key problem by replicating the busiest servers [23]. The solution is implemented by placing Memcahced proxies between the clients to the servers, that distribute the load between the replicated servers. Figure 5 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. Similarly, Trajano et. al. [31] suggest to use proxies (inside virtual machines) to replicate and cache popular keys and to load balance others; however, they increase the network latency and do not analyse the performance when requests follow Zipf distribution.

Pitoura et al. [26] considered replicating items to improve load balancing in the context of range queries in distributed hash tables (DHTs). As our results, they showed that also in their context, replicating few items is enough to give good balance. However, their setting is not directly applied to our memcached setting and their solution does not provide the networking support to enable replication as required in our

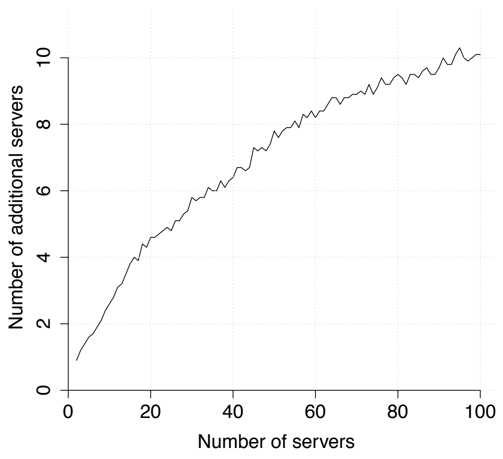


Fig. 5. The number of additional servers (e.g., as suggested by Facebook [23]) required to achieve the same imbalance factor of MBalancer with 10 hot keys and 1000 keys.

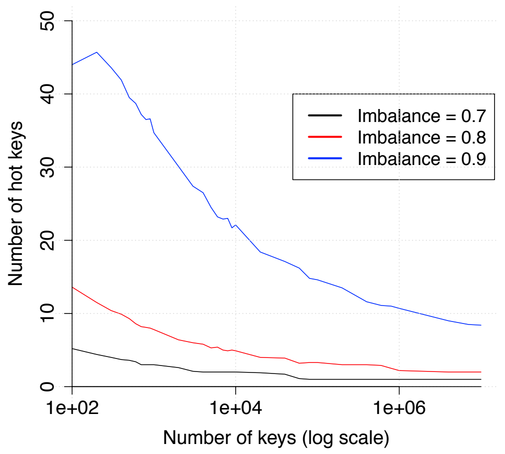


Fig. 7. The number of hot keys required to reach a certain imbalance factor threshold (for 10 servers).

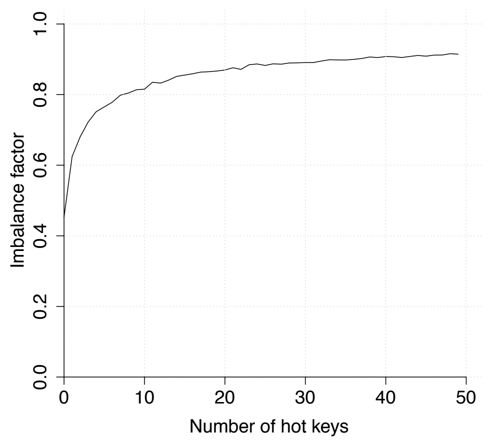


Fig. 6. The imbalance factor as a function the number of hot keys (for 10 servers and 1000 keys).

setting.

Other suggestions are to manually change the application logic [13, 20]. 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 [19], 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 [9] 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.

IV. MBalancer Overview

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 IV-D.

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, $\frac{H_{10}}{H_{1000}} = \frac{2.93}{7.49} = 0.39$). Figure 6 shows the impact of H on the imbalance factor, given a fixed number of servers and a fixed number of keys. Figure 7 shows the number of hot keys required in order to reach a certain value of an imbalance factor (for fixed number of servers). Evidently, the impact becomes negligible when $H \approx 10$, which in the specific setting we considered in the figure results in an imbalance factor of 0.8[3]. 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).

[3]Notice that by Figure 2(b), as the number of servers increases, the imbalance factor improves, even when H is left unchanged.

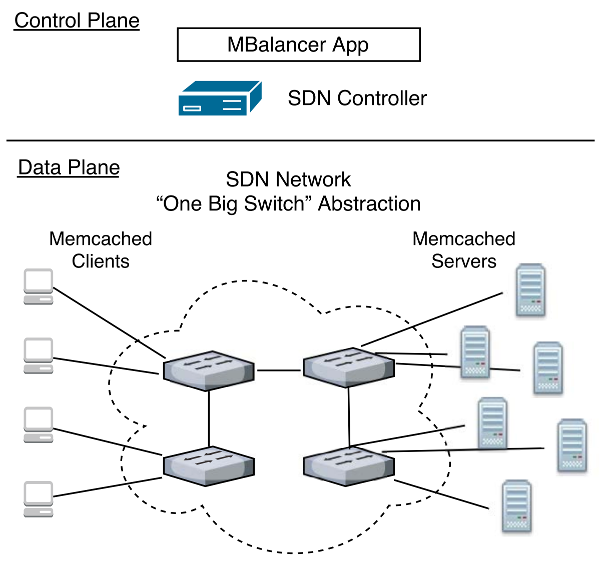


Fig. 8. The MBalancer framework.

A. MBalancer Design

While *MBalancer* can be implemented using a proxy or an L7 middlebox, 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 *MBalancer* architecture is illustrated in Figure 8. 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.

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 [23]). The main challenge with adapting *MBalancer* to TCP is handing-off the connection from its original destination and the destination chosen by *MBalancer*, without requiring the switches to keep track of the state of every open connection (e.g., as in [22]),

B. 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 9 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` [24, 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 IV-C). We note that, unlike `get` operations, `set` operations typically use TCP, and therefore, their destination addresses cannot be simply rewritten as before.

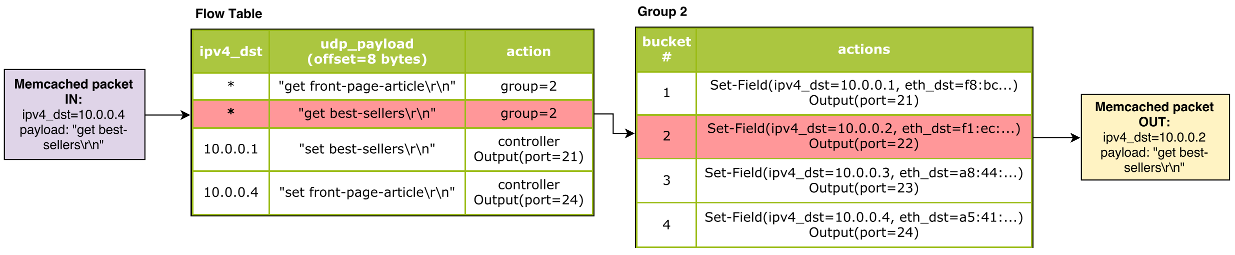
The total number of additional flow table entries is $2H + n$: one rule per hot key for `get` operation, one rule per hot key for `set` operation, and additional group configuration that requires buckets as the number of servers.

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 [16, 17] and can be also applied in our case.

C. 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



ipv4_dst	udp_payload (offset=8 bytes)	action
*	"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)
4	Set-Field(ipv4_dst=10.0.0.4, eth_dst=a5:41:...) Output(port=24)

Fig. 9. Load-balanced packet flow in SDN switch

hot keys is taken from the original Memcached server the key is stored in.[4]

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.[5]

D. 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 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 10 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.

E. P4 Implementation

P4 switches [5] parse packet fields according to user defined protocols and process packets using user defined pipeline of field matches and modifications. Our P4 program extends a simple router with the ability to parse a simplified version of memcached UDP packets and extract the queried key. Then, that key is matched against a list of heavy keys, which are configured externally. Requests to heavy keys are then balanced between a list of servers in a round robin manner.

Specifically, this is done by adding two tables to the simple router base program. The first table contains the heavy keys

[4]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.

[5]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.

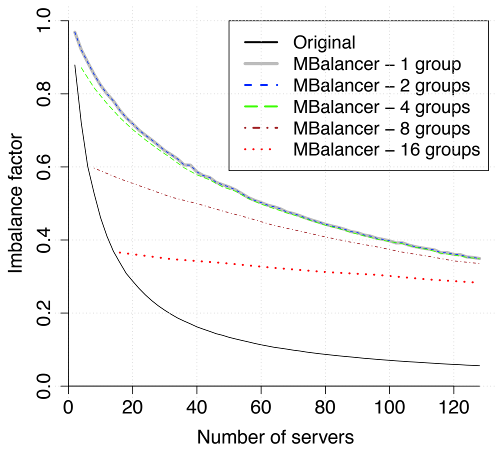


Fig. 10. The imbalance factor as a function of the number of servers, for different number of *MBalancer* groups. All settings use $H = 10$ hot keys out of 1000 keys.

and is matched to the memcached key of the packet. Upon a match, it increases a register that counts how many such heavy key matches were seen so far. The second table contains the list of servers. The current register value (as a metadata field) is used in order to choose which server to send the request, thus achieving round robin semantics.

Selecting the server can be done in two ways: (a) using exact match of register value so register size is a-priori defined according to the number of servers; or (b) the second table uses ternary patterns to map the first k-bits of the register to one of the 2^k servers. While the second option is more expensive in terms of performance, it allows flexible scaling of servers. In both cases, since the register values increased by one for each packet and overlaps to zero when overflowed, we get a round robin load balancing over the servers. Note that heavy keys can be configured on the fly (by adding records to the first table), while the load-balancing system and memcached clients/servers are still working.

We have compiled the P4 program using the p4factory framework [25] and have tested it with Mininet. Figure 11 depicts an example of the configuration code that corresponds to the tables presented in Figure 9. Main parts of the source code are provided in Appendix B.

We note that our P4 program uses mostly stateless operations (i.e., operations that only read or write packet fields).

```
add_entry memcached_match 0x67:65:74:20:66:72:6f:6e:74:2d:70:61:67:65:2d:...  scatter
add_entry memcached_match 0x67:65:74:20:62:65:73:74:2d:73:65:6c:6c:65:72:0d:0a  scatter
add_entry select_heavy_dest 0 set_dest 10.0.0.1
add_entry select_heavy_dest 1 set_dest 10.0.0.2
add_entry select_heavy_dest 2 set_dest 10.0.0.3
add_entry select_heavy_dest 3 set_dest 10.0.0.4
```

Fig. 11. A configuration example that corresponds to the tables shown in Figure 9. The following commands can be applied using the `pd_cli.py` tool which is part of the `p4factory` framework.

The only stateful operation we have is needed to maintain a single counter (or, if one than one more group is used, a single counter per group). As discussed in [30] (and for the stateless operations, even in [6]), our program is feasible in today's technology and can run in hardware line rate.

V. 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.[6] We note that as the number of keys increases, *MBalancer* perfroms better (cf. Figures 2(a) and 7). 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[7]. The prototype of the *MBalancer* application was implemented in Python, as a Ryu [29] 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.

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 12 shows the effect of the imbalance factor on the (normalized) throughput, while Figure 13 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 III.

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.

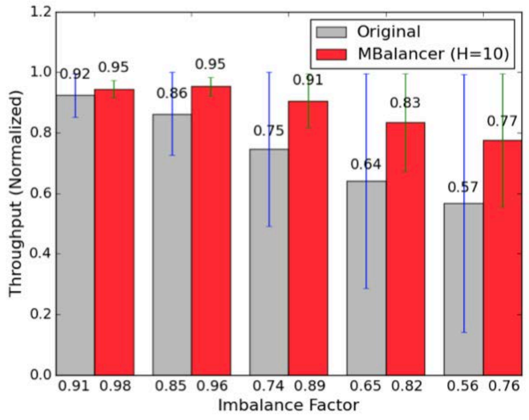


Fig. 12. The average (normalized) throughput with and without *MBalancer*. The vertical line is the normalized throughput of the servers with minimum and maximum load.

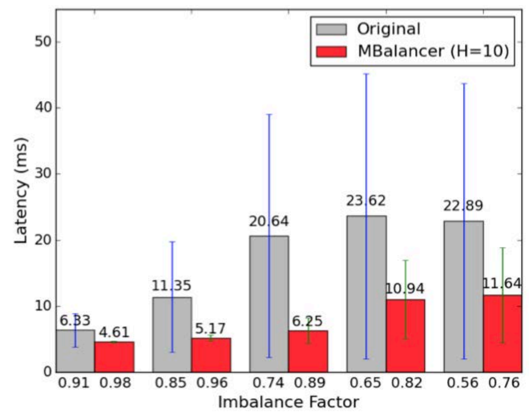


Fig. 13. The average latency with and without *MBalancer*. The vertical line is the values of the servers with minimum and maximum average latency.

VI. Conclusion

In this paper, we presented *MBalancer*: an efficient framework for solving Memcached's hot keys problem by efficiently load balancing its requests. We note that an SDN-based load-balancing was introduced in several works such as HULA [18], B4 [15], and SWAN [12]. These works focus on centralized load balancing for wide area network connecting their data center (namely, load balancing traffic routes for traffic engineering). In this work, on the other hand, we deal with specific (distributed) application and have control on the endpoints of

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

[7]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.

the traffic (i.e., we can change the mapping of keys to servers and can duplicate keys).

On a broader perspective, *MBalancer* demonstrates the ability of switches and routers in SDN environment to solve problems that were traditionally done by middleboxes (namely, in our case, load balancers and proxies). Especially as contemporary switches and routers can look at the payload of the packet and are able to modify the packet before forwarding it, we believe that it is important to investigate the fine line between having middleboxes as separate entities and delegating middlebox capabilities to the switches (which exist in the network anyway). We note that our solution does not require any change to existing switch hardware, and relies on very limited payload matching capabilities (namely, in fixed location in the L7 header). We note that several other works calls for *application-aware data-plane* [22], where extensions to OpenVSwitch (OVS) are suggested to support even more sophisticated operations (e.g., even adding DPI module to OpenVSwitch, as was suggested in [7]). Similar payload matching capabilities were also used in [33] to improve performance of YouTube streaming using SDN.

Acknowledgments: This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n^o 259085, the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the Israeli Centers of Research Excellence (I-CORE) pro- gram (Center No. 4/11).

References

[1] 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.

[2] Amazon. Amazon elasticache.

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

[4] P. Berenbrink, F. Meyer auf der Heide, and K. Schröder. Allocating weighted jobs in parallel. In *ACM SPAA*, pages 302–310, 1997.

[5] 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.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.

[7] C. ChoongHee, L. JungBok, K. Eun-Do, and R. Jeong-dong. A sophisticated packet forwarding scheme with deep packet inspection in an openflow switch. In *International Conference on Software Networking*, pages 1–5, May 2016.

[8] 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.

[9] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, page 23, 2011.

[10] 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.

[11] Google. Google cloud platoform memcache.

[12] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. *ACM SIGCOMM Computer Communication Review*, 43(4):15–26, 2013.

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

[14] R. Jain, D.-M. Chiu, and W. R. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*, volume 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[15] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[16] 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.

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

[18] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. ACM Symposium on SDN Research*, 2016.

[19] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *USENIX NSDI 16*, 2016.

[20] 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.

[21] 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.

[22] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. V. Lakshman. Application-aware data plane processing in SDN. In *ACM SIGCOMM HotSDN*, pages 13–18, 2014.

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

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

[25] P4 Language Consortium. P4 model repository (p4factory). https://github.com/p4lang/p4factory.

[26] T. Pitoura, N. Ntarmos, and P. Triantafillou. *Replication, Load Balancing and Efficient Range Query Processing in DHTs*, pages 131–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[27] M. Raab and A. Steger. *"Balls into Bins" — A Simple and Tight Analysis*, pages 159–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[28] RedisLabs. Redislabs memcached cloud.

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

[30] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, 2016.

[31] A. F. Trajano and M. P. Fernandez. Two-phase load balancing of in-memory key-value storages using network functions virtualization (nfv). *Journal of Network and Computer Applications*, 69:1 – 13, 2016.

[32] Tumblr. Tumblr memkeys. https://github.com/tumblr/memkeys.

[33] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer. Dynamic application-aware resource management using software-defined networking: Implementation prospects and challenges. In *IEEE NOMS*, pages 1–6, May 2014.

Appendix A
Analytical Model

In this appendix we give some more details about the analysis of the hot keys problem using a balls and bins model. Specifically, we analyze the expected maximum load to a server considering the following model: requests are made to N distinct keys where keys are requested according to a key weight distribution $D = \{w_k\}_{i\in[N]}$, such that $\sum_{i\in[N]} w_k = 1$ and $w_k \geq w_{k+1}$ (i.e., w_k is the weight of the k-heaviest key). Each key is hashed to one out of n servers, and all requests to that key are sent to that server.

We observe that this model can be approximated using the weighted balls into bins games model [3], which considers m balls with weights $\{w_k\}_{i\in[N]}$ uniformly thrown into n bins. In case all weights are one then the heaviest bin is with high probablity loaded with $m/n + \Theta\left(\sqrt{m \log n/n}\right)$ balls (and total weight) [27]. For the general case, the maximum loaded bin is at least
$max_{i\in[N]}(\{w_k\})$ higher than the average bin [4].

Considering the Zipf key distribution with parameter $\alpha = 1$, the (normalized) weight of the k-heaviest key is $w_k := \frac{1}{k\sum_{k=1}^{N}\frac{1}{k}}$, which can be approximated by $w_k := \frac{1}{k \ln N}$. When considered as weights of N balls and considering the n servers as bins in a balls-into-bins game, we get that the maximum loaded server has weight of $\frac{1}{\ln N}$ more than the average which is $\frac{1}{n}$.

In order to compare this to the maximum load in the uniform key distribution ($w_k := \frac{1}{N}$), we use the maximum load in the all one weight balls-into-bins game (with N balls and n *bins*) and we factor it by $\frac{1}{N}$. We get $1/n + \Theta(1) \cdot \frac{\sqrt{\log n}}{\sqrt{N}\sqrt{n}}$. When considering the gap from the average ($\frac{1}{n}$), this is asymptotically lower than in the Zipf case, i.e., $\Theta(1) \cdot \frac{\sqrt{\log n}}{\sqrt{N}\sqrt{n}} << \frac{1}{\ln N}$ when n or N are big enough.

As we mentioned before, these bounds are only asymptotic and are not tight. Real-life behavior of hot keys under various variables was presented through simulations in Section III.

Appendix B
P4 source code

A. Memcached Message Header Definitions

```
#define MEMCACHED_KEY_SIZE 128

header_type memcached_udp_frame_header_t{
  fields {
    requestID : 16;
    seqNum : 16;
    totalDatagramsNum : 16;
    reserved : 16;
  }
}


header_type memcached_simple_frame_t {
  fields {
    command3chars : 24;
    space : 8;
    key : MEMCACHED_KEY_SIZE;
  }
}
```

B. Memcache Packet Parsing Configurations

```
header memcached_udp_frame_header_t
    memcached_udp_frame_header;
header memcached_simple_frame_t
    memcached_simple_frame;
```

```
parser  parse_memcached_udp_frame{
    extract(memcached_udp_frame_header);
    return parse_memcached_simple_frame;
}

parser parse_memcached_simple_frame{
    extract(memcached_simple_frame);
    return ingress;
}

header udp_t udp;

parser parse_udp {
    extract(udp);
    return parse_memcached_udp_frame;
}
```

C. Control Program Extension

```
#define SELECTOR_TYPE exact
   // either exact or ternary
#define REG_WIDTH 1
  // if exact then log2(servers_num)

header_type memcached_metadata_t {
  fields {
    selector : REG_WIDTH;
  }
}

register heavy_reqs {
  width : REG_WIDTH;
  instance_count : 1;
}

metadata memcached_metadata_t
    memcached_metadata;

action scatter() {
  register_read
    (memcached_metadata.selector,
     heavy_reqs, 0);
  add_to_field
    (memcached_metadata.selector, 1);
  register_write(heavy_reqs,0,
    memcached_metadata.selector);
}

table memcached_match {
  reads {
    memcached_simple_frame.command3chars
        : exact;
    memcached_simple_frame.key : exact;
  }
  actions {
    scatter;
  }
  size: 1024;
}

table select_heavy_dest{
  reads {
    memcached_metadata.selector
        : SELECTOR_TYPE;
  }
  actions {
    set_dest;
  }
  size: 1024;
}
```