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 ElasticCache [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 Memcached 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 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 Memcached 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
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).\textit{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 \footnote{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.}. Moreover, P4 \footnote{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.} 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 \textit{MBalancer} both over P4 switches and OpenFlow switches with the above-mentioned capabilities.

We have evaluated \textit{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 \textit{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, \textit{MBalancer} significantly reduces the average latency of requests.

\section{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 \texttt{get key}, which retrieves an object according to its key; \texttt{set key,value}, which stores the pair \texttt{(key,value)} in one of the servers; and \texttt{delete key}, which deletes a key and its corresponding value. Every Memcached client is initialized with a list of \textit{n} Memcached servers and a consistent hashing function, \texttt{hash}; thus every Memcached request with key \texttt{k} is sent to server number \texttt{hash(k)}. If key \texttt{k} is not found in server \texttt{hash(k)}, a cache miss is encountered, and the Memcached client reads the data from the database and performs a \texttt{set} request to store the \texttt{(key,value)} 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, \texttt{get} requests are structured as follows: "\texttt{get <key>\r\n}\texttt{\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, "\texttt{get shopping-cart-91238\r\n}\texttt{\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.

\section{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 \textit{n} be the number of servers and \textit{N} be the number of keys. The load on the \textit{i}-th most popular key, denote by \textit{w}_i = Pr[data item has key \textit{i}] is \frac{1}{i^{\alpha} \cdot H_N} approximate \footnote{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.}, where \alpha is a parameter close to 1 (in the remainder of the paper we set \alpha = 1) and \textit{H}_N \approx \ln \textit{N} is the \textit{N}-th Harmonic number. As before, let \texttt{hash} be the hash function that maps keys to servers. Thus, the load on server \textit{j} is \texttt{load}(j) = \sum_{i=1}^{n} \texttt{load}(\texttt{hash}(i)=j) \textit{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 \textit{k}) becomes saturated and the rest of the servers process requests proportionally to server \textit{k}'s load:

\[
\frac{1}{n} \sum_{j=1}^{n} \texttt{load}(j) = \frac{1}{n} \sum_{j=1}^{n} \texttt{load}(j) \approx \frac{\texttt{load}(k)}{n}.
\]

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. 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 \textit{MBalancer}, whose specific design will be explained in Section IV, achieves substantial improvement of imbalance factor, even when a only
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.

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 Memcached 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
Other suggestions are to manually change the application logic \cite{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 differs from general key-value storage system designs that aim at improving the storage system performance by adding cache nodes. Specifically, recent system designs \cite{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 \cite{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, $H/1000 = 2.93 \approx 0.39$). 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).

\(^{3}\text{Notice that by Figure 2(b), as the number of servers increases, the imbalance factor improves, even when } H \text{ is left unchanged.}\)
we have devised an SDN application, named MBalancer, that involves actions in both planes. Specifically, in the control plane, the MBalancer application performs hot keys duplication 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 \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

---

**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 memcached 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 Memcached traffic statistics from some monitoring component solution (e.g., nctop 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.

Fig. 8. The MBalancer framework.
hot keys is taken from the original Memcached server the key is stored in.\textsuperscript{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.\textsuperscript{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

\textsuperscript{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.

\textsuperscript{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.

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),
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. We note that as the number of keys increases, MBalancer performs 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. 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 server 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 servers, 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.

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...
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 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° 259085, the HIUJ 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) program (Center No. 4/11).

REFERENCES


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
A. Memcached Message Header Definitions

```c
#define MEMCACHED_KEY_SIZE 128
header_type memcached_key_size_t {
    fields {
        space : 8;
        key : MEMCACHED_KEY_SIZE;
    }
}

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

```c
header memcached_udp_frame_header_t {
    memcached_udp_frame_header;
}
header memcached_simple_frame_t {
    memcached_simple_frame;
}
```

C. Control Program Extension

```c
#define SELECTOR_TYPE exact
#define REG_WIDTH 1
header_type memcached_metadata_t {
    fields {
        selector : REG_WIDTH;
    }
}

#define REG_WIDTH 1
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);
}

```

APPENDIX B

P4 SOURCE CODE

A. Memcached Message Header Definitions

- `MEMCACHED_KEY_SIZE` is defined as 128.
- The `memcached_key_size_t` structure contains fields for space, key, and `MEMCACHED_KEY_SIZE`.

B. Memcache Packet Parsing Configurations

- `memcached_udp_frame_header_t` includes fields for requestID, seqNum, totalDatagramsNum, and reserved.
- `memcached_simple_frame_t` includes fields for command3chars, space, and key.

C. Control Program Extension

- `SELECTOR_TYPE` is defined as `exact`.
- `REG_WIDTH` is set to 1.
- The `memcached_metadata_t` structure contains a selector field.
- The `scatter` action reads from `memcached_metadata.selector`, adds 1 to the field, and writes 0 to `heavy_reqs`.

The document discusses the weight distribution and boundedness of max bin for various key distributions. It introduces a model where weighted balls are thrown into bins, and each key is hashed to one of the bins. The maximum loaded server is considered as the maximum weight, and the gap from the average is discussed. The document also presents simulations for weighted balls into bins, where the maximum loaded server is at least \( \frac{1}{\ln N} \) more than the average bin. It further discusses the maximum loaded server in the Zipf key distribution case and the maximum load in the uniform distribution case. The document concludes with a discussion on various variables and their behavior in different scenarios.