Advanced Topics in IP Networks

Project: P4 – Memcached Load Balancer

# Introduction

Memcached is a very popular general-purpose caching service that is often used to boost the performance of dynamic database-driven websites. Memcached architecture is depicted in Figure 1.

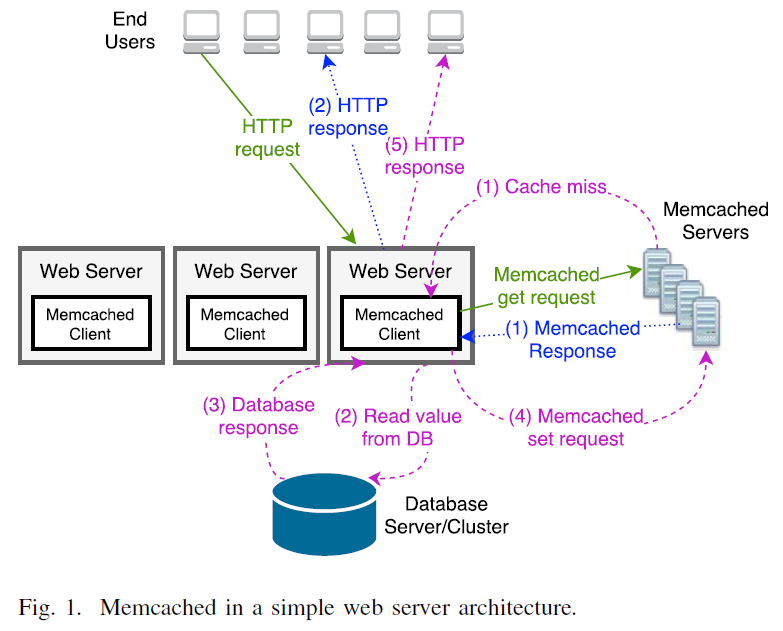
Popular data items are stored in the RAM of Memcached servers, allowing orders of magnitude faster query time than traditional disk-driven database queries. Data items are stored in Memcached servers by their keys, where each key is linked to a single Memcached server, using a consistent hashing algorithm. 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 requests to their corresponding data item (namely, the key

Figure . Memcached in a simple web server architecture.

load). It is well-known that web-items in general, and data elements stored in Memcached in particular, follow a Zipf distribution, 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. Once the hot-keys (or the loaded servers) are detected, we implement the MBalancer[1] approach as suggested by the paper this project is relying on, which is a simple L7 load-balancing scheme for Memcached.

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 P4 switches identifies a Memcached get request for a hot key, the switch sends the packet to one of the servers using a load balancing program**.

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:

In Figure 2 we see how the number of hot keys affect the imbalance factor. Mind that this is a recreation of a graph from the original paper this project is based on.

Figure . Imbalance factor as a function of the number of keys, for various number of servers

# Related Work

Facebook suggests to solve the hot key problem by replicating the busiest servers. The solution is implemented by placing Memcahced proxies between the clients to the servers, that distribute the load between the replicated servers. this solution is more expensive in CAPEX and requires extra software. Similarly, Trajano et. al. 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 analyze the performance when requests follow Zipf distribution. Pitoura et al. 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 setting. Other suggestions are to manually change the application logic. 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.

# Technical details

I’ve implemented a load balancing application for Memcached servers using P4 and predetermined routing rules. Whenever a Memcached request packet is received by the P4 switch, it parses the command and the first 16 bytes of the requested key. If the Memcached request is a “get” command, and the key matches a predetermined hot key, then the request is flagged as a request for a hot key and an internal register that counts the number of hot keys requests increments. According to the value of said register, a Memcached server is selected. Mind that the size of the register needs to match the number of servers the load balancing should happen on, for it to be round robin.

There is also a Python class handling the actual exchanges, for either “set”/ “get” operations, as well as simulated query of items that does not actually reach any server, but is useful for load calculations.

The whole setup runs within a simulation environment that is based on Mininet.

# Evaluation and discussion

Because it is run in a simulation environment, it is very limited in regards to actually measuring performance. I’ve tried various methods of creating load on the servers, but managed only to either load the P4 switch, or break the simulation. Further work could be done on a non-simulated environment with a P4 switch, to measure the actual performance gain of the solution, and see if it matches the theory of improving the throughput to be like the improvement in the imbalance factor.

The non-performance related evaluation is rather simple. First a bunch of keys are set to 2 servers. The first 10 most frequent keys are keyed in to the predefined tables of the P4 switch as potential matches of a hot key. Then queries are made to both normal keys and hot keys and we see which server responded. The responding server for hot keys should change each time.

# Conclusion

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).

# Bibliography

1. Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load Balancing Memcached Traffic Using Software Defined Networking, 2017
2. https://github.com/staveliav/MBalancer.git