Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Advanced Topics in
Communication Networks (Fall 2018)

Group Project Report

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Authors:
Malte Brodmann
Dimitris Lekkas

Advisor: Thomas Holterbach

Supervisor: Prof. Dr. Laurent Vanbever

Submitted: Dec 16, 2019
3 Pages

Abstract

The main result of this project is a P4 implementation of the NetCache key-value store architecture. Following the major ideas we implemented most of the aspects and algorithms that were described in the paper. Additionally we also implemented simple key-value stores, a client API and a controller application as well as an environment to test our implementation. Some of the challenges included (...) Our P4 code does not target a specific programmable switch architecture but was developed and tested for BMv2 software switch. Thus we were not able to directly compare our resulting implementation with the results of the paper that were obtained using a Tofino switch. However we ran an experiment that shows that our implementation achieves good load-balancing for key-value stores connected to a switch running our P4 application. (results ...)

Contents

1 Introduction

Introduction to the problem that was solved in this project.
The main goal of this project is to give an implementation of NetCache [2]. Netcache is a new approach that presents one possible answer to the problem of load-balancing key-value stores - especially in the context of high-throughput internet services. The main contribution of the paper is a new key-value store architecture where data is cached on programmable switches to allow for dynamic load balancing. As part of our project we implemented a P4 application that enables programmable switches placed on network paths to key-value stores to cache data. Our implementation follows the major concepts and techniques presented in the paper with only minor adaptations. As the scope of the project was limited we only provide a P4 implementation for a software switch, namely the BMv2 simple switch architecture, instead of targeting an actual programmable switch (e.g. Barefoot Tofino). In addition we also implemented a simple NetCache compatible in-memory key-value store and a client API to be able to test our NetCache implementation for correctness. Finally we conducted a small experiment that is inspired by the nature of actual key-value store workloads. These workloads are usually highly skewed in practice. We examined whether our implementation achieves the dynamic load-balancing of the connected key-value stores presented in the paper.

2 Background and related work

Briefly describe background information and related papers (if any). You do not need to describe topics that were covered in the lecture, only other topics that are relevant for your project.
Many modern internet or web applications highly depend on high-throughput key-value stores. Recent studies showed that the workload for such services is oftentimes skewed resulting in a rather small set of keys which is queried very frequently whereas the majority of keys are only queried occasionally. This can lead to situations where some key-value stores can't handle the large amounts of queries they receive while others are barely utilized. An important theoretical result proves that caching can be used to improve load-balancing for key-value stores [1] . However there are two important requirements for this approach. First, for N key-value store nodes a cache must be able to store $O(NlogN)$ items. Second, the throughput of the cache must be at least the aggregate throughput of all key-value store nodes. With the recent shift from flash and disk based racks to in-memory key-value stores it becomes increasingly hard to satisfy this requirement using in-memory caches. By moving the caching layer to the network the throughput of the cache can be increased to be able to provide the aggregate throughput of all in-memory key-value stores. NetCache gives such an implementation of a cache placed on the network. (perhaps also address some of the challenges that are involved)

3 Implementation

Describe how you solved the problem and how your implementation works. Do not paste source code here.
We implemented all major aspects described in the NetCache paper. As the paper extensively covers all employed techniques and algorithms we mostly focus on the challenges we faced and aspects that were not specified precisely by the NetCache authors. In addition we briefly describe our client API, key-value store and controller implementations as well as our test setup.

3.1 Setup and network topology

For our P4 application we assumed and tested our implementation using the following network topology. The topology consists of a set of servers running our key-value store application.

They are connected to a programmable switch which runs our P4 NetCache application and also implements L2 and L3 layer forwarding. Additionally a client is directly connected to this switch as well. The client can send queries to every key-value store instance using our client API.

3.2 NetCache Protocol

3.2.1 Packet format

The authors of NetCache proposed an application-layer protocol that is used to send and answer key-value store queries in the context of NetCache. The major header fields of the protocol are OP, SEQ, KEY and VALUE. OP denotes the corresponding operation of the query (e.g. Get, Put or Delete), SEQ represents a sequence number, KEY is the key of a key-value pair stored in a key-value store and VALUE is a variable length field storing the respective value. All of our application components implement this protocol using UDP for Get/Read queries and TCP for Put/Write and Delete queries with some minor adaptations. First, as the authors did not explain how they implemented the variable-length value field we fixed the length of the value field to 128 bytes. Thus we allow for values with a length of at most 128 bytes. We achieve variable-length values by adding zero-bytes to all values that do not exceed this limit. Another option could consist of introducing a new protocol header field that states the length of the value. However we decided against this approach to not change to NetCache protocol on a large scale. Second, we introduce a few new values for the operation entry of the NetCache header. These represent specific states that are important for cache coherence will be explained later in section ...

3.2.2 Network Routing

On top of NetCache the P4 application we implemented also supports L2 and L3 layer forwarding. All corresponding forwarding tables are populated assuming the topology described above.

3.3 Query Handling

After receiving a packet the switch parses it using the corresponding L2, L3 and L4 layer parsers. A packet is parsed as a NetCache packet when the respective UDP or TCP packet contains a special port number we reserved for the NetCache protocol as either a source or destination port. Additionally the payload of this packet must contain the NetCache packet header values. We reserve one byte for OP, four bytes for SEQ, 16 bytes for KEY and 128 bytes for the VALUE field. Therefore a packet must have a payload length of 149 bytes in order to be correctly parsed as a NetCache packet.

Packets that were not parsed as NetCache packets will simply be forwarded using the corresponding routing tables. The following sections describe how NetCache packets will be handled by our switch application. The switch application will perform different actions depending on the operation of the NetCache query. By simply looking at the OP header field the switch is able to distinguish between the different types of queries.

3.3.1 Get / Read queries

When receiving a Read query the switch first has to determine whether it cached the queried key. If yes, the switch can directly reply to this query otherwise it has to forward the query to key-value store. To do so, we follow the approach described in the paper. A match-action table matching on the NetCache KEY header field is used to determine whether the respective item was cached by the switch or not. The corresponding table is maintained by the controller, i.e.

the controller inserts entries or removes entries from the table whenever a new item is inserted or evicted from the cache. This is done using the SimpleSwitchAPI. If there is no matching key in the table the switch will simply continue to forward the packet. If the switch cached the queried item we additionally have to check if the respective cache entry is valid. This is important to ensure cache coherence (see section ...). Following the descriptions of the paper we use a register array that contains a bit for every possible slot where a cache item can be stored. The bit represents the validity of the corresponding cache items. If the read to the corresponding register returned the information that the cache item is invalid the switch will only be able to forward the query. Otherwise we know that the switch cached the queried item and should directly respond the client. Therefore we swap all L2, L3 and L4 source and destination header fields and route the packet to the respective egress port. While all of the above happened in the ingress part, it is the egress parts responsibility to insert the value of the cached item into the VALUE header field. If a item is cached by the switch applying the match-action table in the ingress part also sets some metadata. This metadata described where the corresponding entry is located in the switch and how the value can be retrieved. Using this metadata we append the missing information to the VALUE header field when the queried item is cached and valid. We will cover the details on how this is done in the section about memory management. This completes all actions to be performed for handling Read queries.

3.3.2 Put / Write queries

3.3.3 Delete queries

3.4 Client API

We implemented a simple client API in python that can be used to send NetCache queries to a set of key-value stores.

3.5 Key-Value Store

4 Evaluation

Describe how you tested your implementation and summarize the results.
As we are only able to test our implementation using software switches in the scope of this project we are not able to directly reproduce the results of the paper. However a correct implementation should still be able to achieve good load-balancing for all key-value stores with high probability. Thus we generated zipf-distributed workloads for our implementation using our client API. (...)

5 Conclusion

A brief conclusion about how you solved the project.
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

References

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

[2] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 121–136.

A Group organization

Briefly describe what each team member was contributing to the project

Malte Brodmann Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Dimitris Lekkas Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.