AppSwitch: Application-layer Load Balancing within a Software Switch

Eyal Cidon
Stanford University

Sean Choi
Stanford University

Sachin Katti
Stanford University

Nick McKeown
Stanford University

ABSTRACT

With the increase in adoption of SDN, programmable switches are becoming an attractive mechanism for enabling network operators to configure protocols without replacing networking hardware. This paper argues that the functionality of programmable switches can be extended beyond network operations to the application layer, in order to achieve significant end-to-end application performance improvements. To illustrate the potential of this new concept, we develop AppSwitch, a packet switch that also performs load balancing for key-value storage systems. Unlike existing key-value load balancers, which require each request to send an extra message to a proxy server, AppSwitch requires only a single message from the key-value client to the server. This results in a 2x reduction in end-to-end average latency and 2x throughput improvement. We implemented AppSwitch on PISCES, a P4 programmable version of Open vSwitch. Finally, we demonstrate that AppSwitch can be deployed transparently, without any changes to the key-value clients and servers.

CCS CONCEPTS

• **Networks → Programmable networks**;

KEYWORDS

Programmable Switches, Load Balancing, Key-Value Stores

ACM Reference format:
Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. 2017. AppSwitch: Application-layer Load Balancing within a Software Switch. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3106998

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APNet'17, August 03-04, 2017, Hong Kong, China
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3106998

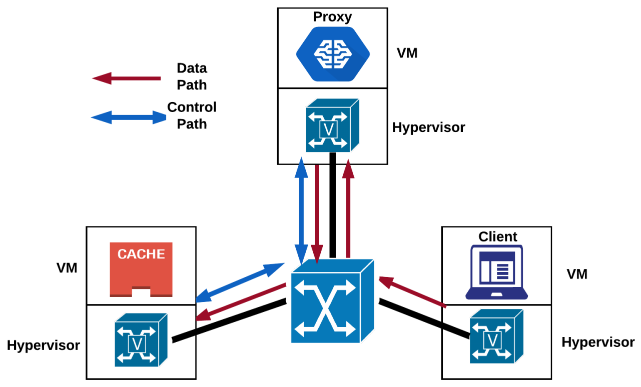
1 INTRODUCTION

Programmable switches have the potential to transform data center networking [13]. The main advantage of programmable switches is that they enable network operators to upgrade to new network protocols or modify existing protocols without having to deploy any new networking hardware or change any dataplane level software. With programmable switches, when a network operator can change the existing network behavior by simply defining the new behavior in a compatible language and compile and upload the target specific code into the target programmable switches.

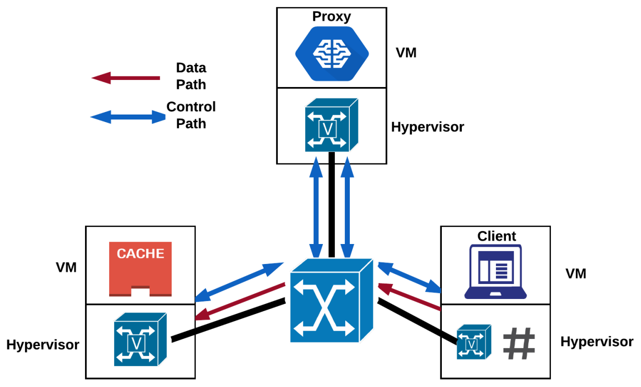
Yet, we argue that the potential of programmable switches goes far beyond the scope of the network operator. In this paper we demonstrate through an implementation and measurement that the application layer can leverage the programmability in software switches to achieve application level performance gains. AppSwitch joins other papers who also advocated placing application layer support in switches [9, 14, 19, 21, 28]. To this end, we design and implement a motivating application, AppSwitch, a key-value load balancer embedded in a switch.

Key-value stores are an essential storage layer for of modern web applications. Popular examples include LevelDB, RocksDB, Redis and Memcached [1, 2, 6, 7]. Key-value stores have a relatively simple API, based on getting and setting values using a key. These caches are typically deployed in clusters of thousands of servers, which require load balancing for routing requests among servers [24]. The load balancer is implemented as a software-based proxy [24].

While a load balancer is essential for scaling key-value clusters, it also incurs latency and throughput overhead, since it requires an extra message to a proxy server [20]. This extra message can almost double the network latency in cache operations [24]. In this paper, we use the example of load balancing in Memcached, a widely used key-value cache. Our ideas can be generalized to other key-value stores.



(a) Traditional key-value cache deployment



(b) Key-value cache deployment with AppSwitch

Figure 1: Key-value cache deployments in a datacenter including the control and data paths

We visualize a typical key-value cache deployment in Figure 1a. The figure depicts a cache client, a proxy and a cache server all running on virtual machines. Each virtual machine is connected to a virtual switch and the physical machine is connected via a physical switch. The figure depicts the control and data paths in the network when a client sends a cache request. The request must first be routed from the client to the proxy. The proxy performs a hash operation to find an appropriate server for the key. Only then the key is routed to the correct server.

As an alternative to current key-value load balancers, we introduce AppSwitch, a load-balancer that is implemented in a protocol independent software switch. The key idea of AppSwitch is that by introducing the key-value cache protocol header to a switch connected to the client, it is possible then to load the key hash, which was computed by the proxy, into the switch. AppSwitch removes the extra message to the proxy from the critical path of key-value lookups, while remaining transparent to the cache client and servers.

Figure 1b depicts AppSwitch's deployment. The data path does not require an extra message to the proxy, since the server that is the destination of the request is looked up by checking the key hash table at the virtual switch. The proxy is now responsible for updating the virtual switch with the up-to-date key hashes.

AppSwitch hides its load balancing capabilities from most traffic in the network, and acts as a regular networking switch that forwards packets between servers in the network. However, when it detects a key-value cache operation, it performs the required load balancing. This allows AppSwitch to be seamlessly integrated within the network without disrupting existing network flows.

We implemented AppSwitch using PISCES [27], a modified version of Open vSwitch(OvS) [25] software switch that allows the modification of the network behavior via the P4 [12] high-level domain-specific language designed to specify network behavior. Since AppSwitch's header definitions are written in P4, it can easily be installed on any P4 capable switch, including hardware switches. Furthermore, since we use the native protocol definitions of the key-value store, it does not require any modification to the key-value client and server code.

We describe the design and implementation of AppSwitch, with the goal of motivating additional work in leveraging programmable switches for application layer acceleration.

2 BACKGROUND

In this section we provide background on key-value load balancing, programmable switches and software switches. We then explain how the data path of key-value load balancers can be offloaded to a protocol independent software switch.

2.1 Key-Value Load Balancing

In order to understand how programmable switches can be leveraged for key-value load balancing, we first describe the architecture of a key-value load balancer. A key-value load balancer is responsible for distributing read and write requests to different key-value servers. There are various implementation of key-value load balancers available, ranging from a very simple implementation that performs static hash based load balancing, to sophisticated systems that dynamically change their load balancing decisions based on server load. For this paper, we compare AppSwitch with a popular key-value load balancer called McRouter [20]. McRouter is widely used in Facebook to load balance live traffic. McRouter provides four main features:

- *Key Based Routing* - McRouter routes incoming requests to the right server, using the object key contained in the request.

- *Dynamic Scaling* - McRouter continuously monitors the load of the cache servers and auto scales according to load.
- *Fault Detection* - McRouter monitors the caching servers to detect server faults. In case of a fault, McRouter needs to route all the keys associated with the server to a different non-faulty server.
- *Replication and Pooling* - McRouter pools together multiple caches and replicates the data between them for fault tolerance.

Of these four features of McRouter, only the key based routing is on the data path of client requests. The other operations involve communications between the load balancer and the cache servers. Therefore, we can offload the key based routing from the load balancer into the software switch, to save an extra network hop and reduce redundant communications between the servers and the load balancer.

2.2 Programmable Switch

Recently developed programmable switches enable simple modification of network behavior by defining new headers and operations on these headers in a high-level language. The language of choice for many application is P4 [12]. P4 allows simple declaration of packet header formats, actions can be performed on the headers, as well as the control flow that dictates which actions to be performed on the header. In addition, it allows control over data plane states and on how these states are accessed.

P4 is based on the match-action packet processing model. In the match-action model, the set of operations a switch needs to perform on a packet is described as a series of header match and action operations. A match operation consists of lookups on one of the packet headers and matching the header field on a predefined set of match rules. A action operation consists of operating a series of primitive actions based on the result of the match operation.

3 DESIGN AND IMPLEMENTATION

The key idea behind AppSwitch, is that key based routing, which is the only function of the load balancer that sits in the client's data path, can be implemented as a simple match-action model in P4 and installed in a programmable switch. When the load balancer generates a consistent key hash, the load balancer simply updates the routing rule in the modified switch, allowing the client to avoid sending messages to the proxy server.

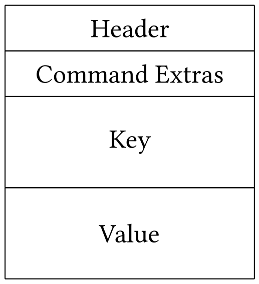
The rest of the functionality of the load balancer is not offloaded to the switch. The load balancer stills monitor the caching servers for load and for failures. When a new cache server instance is created, the load balancer generates a new hash for routing. This hash then needs to be updated in the switch. Similarly, when a fault occurs, the load balancer updates the routing tables and the client continue sending requests uninterrupted.

Our implementation of AppSwitch is specific to Memcached [2] traffic and in the future we plan to expand AppSwitch to work with any key-value store protocol. Requests in Memcached can be sent using two protocols: an ASCII protocol in which operations are described using ASCII characters, or a binary protocol, in which operations are described using a custom packet header [3]. AppSwitch load balances by utilizing the Memcached binary protocol. Unmodified Memcached binary protocol is written in P4 and installed in PISCES. Detailed descriptions of the Memcached packet and binary protocol header are available in Figure 2.

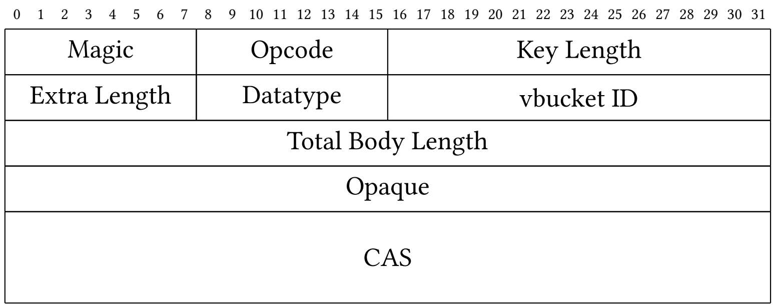
In addition to the Memcached header, we define a header that contains the key of the object currently being requested. Memcached uses variable-sized keys based on the size of the key and the value. Unfortunately, PISCES currently does not support variable-sized fields in newly defined headers. Therefore, AppSwitch currently only supports keys up to 80 bytes. However, the vast majority of keys in Memcached deployments tend to be small and therefore fit in an 80 byte field [11], which makes AppSwitch usable for most use cases. To represent keys which are smaller then 80 bytes, we pad the key header field with zeros. For performance, AppSwitch uses UDP as its main communication protocol rather than TCP protocol. TCP support can be implemented using P4 state registers [10], the switch will need to be able to answer the client's TCP handshake before the client sends the key. We believe that performance will still be improved in this case since we will still be saving sending the TCP session to the proxy server. We plan to address this in future work.

We create match-action control flows of AppSwitch for the key based load balancing. Unlike the new headers defined in P4, the match-action control flows are defined as the flow table entries in PISCES.

The load balancing operation starts by matching on the UDP header. We match on the destination port to check if the packet is a Memcached packet or not. If the packet is not a UDP packet, or if its destination port is not the Memcached port (port 11211), the packet is treated as a normal IP packet and forwarded according to its destination IP and MAC address. If the packet is indeed a Memcached packet, we then match on the magic field in the Memcached binary protocol header. This field is set to `0x80` if it is a request from a client to the server or `0x81` if it is a reply from the server to the client. If the packet is a request, we match on the packet's header field that holds the key. Then, the pre-installed flow rules are used to reperesent hashing of the key to the correct Memcached server. The output of the hash results in the IP of the server to which we send the request.



(a) Memcached packet structure



(b) Memcached binary protocol header, important fields include: *Magic*, specifies request or response packet and *Opcode* specifies the operation to be executed

Figure 2: Memcached packet structure and headers

Finally we implement a controller system for AppSwitch based on McRouter. Since McRouter does not include SDN capabilities and thus cannot install and modify rules in switches, we integrate our controller with Ryu[8], a python based open source SDN controller, to dynamically install new flow rules to route newly added keys to the correct switch.

4 EVALUATIONS

We demonstrate the potential performance gains of AppSwitch with a synthetic workload. In the experiment we send key-value commands with small value sizes that fit into a single packet. It was shown in [11] that the vast majority of key-value requests are smaller than 1Kbyte.

However, the gains we demonstrate are still valid for a large, multi-packet request. This is because in the unmodified key value store system the client has to send the entire multi-packet request since the client does not keep and track the actual server destinations [4].

The main performance metrics that we measured are throughput and latency measurements on cache operations. We compared AppSwitch's performance to McRouter.

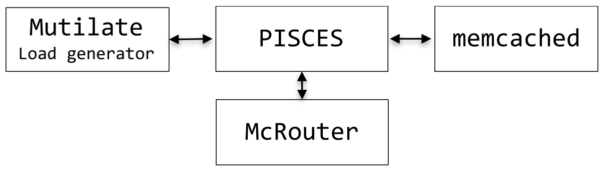


Figure 3: Experimental Setup Topology

4.1 Experimental Setup

Our experimental setup consists of four machines. Each machine takes the role of either a Memcached client, a McRouter proxy, a Memcached server or a software switch. Each machine is configured as a virtual machine with 2 CPUs and with 8GB of RAM running Ubuntu 14.04. All three machines have a single network connection to the software switch. The overview of the setup can be seen in Figure 3.

We configured the client to send Memcached traffic via a traffic generator called Mutilate [11]. Mutilate is a load generator designed for high requests rates, tail-latency measurements and realistic request streams. For the Memcached server, we configure it with the copy of Memcached version 1.4.33. For the McRouter machine, we configure it with release 22.0, and we configured it to only load balance to the single Memcached server. In this way we simulate the performance of a proxy machine that helps locate the Memcached server for the client request. Finally, on the switch VM, we install a PISCES software switch [27] and configure it's forwarding table.

4.2 Experimental Result

We run two different experiments with the given setup. Our first experiment is the baseline experiment, where the query from the client is routed through McRouter to locate the correct Memcached server which will serve the query. We configure Mutilate to send queries that consist of 50% of SET requests and 50% of GET requests. When the queries arrive at the Memcached server, it simply serves the requests that it receives. Our second experiment evaluates the performance of AppSwitch. Instead of forwarding the request to McRouter, we install a forwarding rule that look into the Memcached header and forward the request directly to the correct Memcached server.

The results of the baseline experiment and the AppSwitch experiment is provided in Table 1. As we can see, AppSwitch performs on average roughly 2X better and has an even greater speedup on on tail latencies. This improvement in performance is due to two main reasons: The first is that we are skipping the network messages between McRouter and the switch on the datapath. In the entire system, there

	Avg. QPS	Avg. latency (ms)	95th latency (ms)	99th latency (ms)
Baseline	631.0	1696.7	2025.4	4122.5
McRouter	1156.8	967.8	1368.2	1547.7

Table 1: Queries per Second (QPS) and latency comparison between using McRouter based setup and AppSwitch.

are total of eight messages between different servers and the switch. By removing McRouter from the datapath, we save four messages on the datapath, two from the client the McRouter and two which go between McRouter and the Memcached server on reply. Therefore, we can see that there are about 2X improvements in the number of messages. Another reason for the improvement is that McRouter does other operations based on the request beyond load balancing, such as maintaining fault tolerance, which add more overhead.

5 DISCUSSION

We believe AppSwitch is just one example of the application-layer benefits that programmable switches can provide.

In the immediate future, we plan to generalize AppSwitch to work with any key-value store. Then, in order to scale our system to multiple switches, it will need to provide consistency guarantees. In the longer term, we plan to explore load balancing in other contexts besides key-value stores, and embedding additional application layer functionality, such as node discovery and distributed storage in programmable switches.

5.1 Supporting Other Key-Value Stores

The current implementation of AppSwitch only works with Memcached, yet the idea of offloading the key hash to the switch is a general one and can be applied to other key-value protocols. We would like AppSwitch to support any existing or future key-value store.

A straightforward way to support this is by adding the protocol headers for various key-value stores in P4, compiling them into a programmable switch that will understand the protocols and perform the cache server lookup based on the information in the header. One issue that arises from this approach is that most current implementations of P4 programmable switches, such as PISCES, require recompilation of the entire switch to run different versions of P4 programs. This means that in order to dynamically support multiple key-value stores in a single switch, we have to face either downtime in order to switch between multiple P4 programs or come up with a generic protocol that can be applied to multiple key-value stores.

In order to generalize the header structure, we plan to add two new headers to AppSwitch which will replace the Memcached specific protocol headers. The first of these headers will be a header which the client attaches to each request. The fields in the header will be pointers detailing where in the packet the key starts and ends. The header will also include a preamble so that it will be detectable at the switch. The second header included in the parser will be a list of arbitrary bytes, in which the real key-value protocol headers and the key will reside. The proposed headers are described in Figure 4 as P4 code. This can either be attached in the client software, as a kernel module or in the software switch by parsing the headers to compute the set of headers to replace.

```
header_type key_location {
        fields {
                preamble : 8;
                key_start_byte : 32;
                key_end_byte : 32;
        }
}

header_type general_key {
        fields {
                byte_1 : 8;
                byte_2 : 8;
                ....
                byte_n : 8;
        }
}
```

Figure 4: Proposed P4 header for general application load-balancing.

5.2 State Management Across Switches

Another challenge in designing the AppSwitch proxy is updating state across multiple switches. Large scale key-value store deployments comprise of thousands of servers, with a large amount of clients. The number of switches these clusters require is linearly dependent on the number of clients, since each client running in the data center must be connected to a switch. Especially once AppSwitch supports other key-value stores that have strict consistency guarantees, it will need to provide consistent updates across these switches.

Our main insight is that the state management in AppSwitch is similar to traditional SDN controllers. Since we route packets based on the keys of the message, changes in load balancing decisions are translated to route changes on the switch. This is no different then the requirement from an SDN controller to change the routing state of the switches based on link state. This insight allows us to use the vast research into SDN consistent state transitions into our system [15–17, 22, 23, 26, 30]. Kinetic [26] defines different types of network update consistency guarantees. In general, some form of per-packet consistency needs to be provided, which means that every packet in the network is treated as if it is being routed before the update or after the update.

5.3 Generalizing Load Balancing

In the last section we described how AppSwitch might be expanded to load balance any key-value store. But an even broader opportunity would be to load balance based on any arbitrary key. In fact the keys do not need originate just from from key-value stores. The general key header described in Figure 4 can be used on any set of arbitrary bytes and thus can be expanded to any application. This will allow us to potentially expand AppSwitch to handle layer-7 load balancing independently from the application.

The general key based routing abstraction can be further expanded to the general node discovery problem which is common in many distributed systems. In node discovery, the network needs to find a server based on some memory address.

6 RELATED WORK

AppSwitch joins with other papers who have made a call to utilize programmable switches and network for application layer support [9, 14, 19, 21, 28]. We will first discuss some other alternatives used in load balancing and then discuss some of the previous systems that implement non-network related applications on a network switch.

6.1 Load Balancing Systems

There is a lot of existing work on load balancing using a proxy server, dedicated hardware or NFV instances. The design of these systems is quite similar. Almost all of them require the clients to send the request to the load balancer first, then the load balancer will either tell the client where to resend the request or route the request to the desired server themselves. Prominent recent examples include Facebook's Memcached load balancer, McRouter [20] and NetKV [29]. AppSwitch improves upon these load balancer proxies by offloading the key hash to the switch, and thus reducing the number of hops in the key-value data path, and increasing performance.

6.2 Other Switch-Based Systems

Another related work is SwitchKV [19]. SwitchKV is a key-value store that utilizes existing switches to provide load balancing. While SwitchKV also offloads load balancing to the switch, there are major differences in its deployment model. In SwitchKV object keys must be encoded in the destination MAC address field in the Ethernet header. This encoding might be problematic when SwitchKV is deployed in a data center, which also includes non key-value traffic, since it will be difficult to detect cache related traffic. AppSwitch performs key-value load balancing transparently to the rest of the network, by simply adding new key-value protocol headers to protocol independent switches. Additionally, SwitchKV's design requires deploying a brand new key-value store from scratch. This requires replacing existing key value stores with a new communication protocol, server code and controller, which would entail replacing existing key-value deployments across thousands of nodes. AppSwitch integrates seamlessly with existing and future key-value store solutions, requiring only changes to the proxy to properly run.

There are several existing projects on embedding application layer functionality into programmable switches. Examples range from simple ones like placing a NAT on a switch [9] to UnivMon [21] which use P4 for network monitoring, and implementing the Paxos algorithm in a programmable switch [14]. There are many more projects presented in OpenNFP [5] There are even older software switch implementations, which allowed for middle-box functionality such as the Click modular router [18].

7 SUMMARY

In this paper we demonstrated how to use programmable switches to accelerate application layer protocols, by introducing AppSwitch a key-value store load balancer built in a programmable switch. By leveraging switch programmability AppSwitch load balances key-value store request directly in the network switches, thus removing the requirement for the client to send requests directly to a load balancer proxy. AppSwitch can be integrated seamlessly into the network without any modification to the client. Through experiments, we demonstrated that AppSwitch can introduce a significant performance benefit to the majority of key-value store. Finally we presented an idea on how to generalize AppSwitch to other key-value stores and additional applications.

REFERENCES

[1] LevelDB. leveldb.org.
[2] Memcached. memcached.org.
[3] Memcached binary protocol. https://github.com/memcached/memcached/wiki/BinaryProtocolRevamped.

[4] Memcached message protocol. https://github.com/memcached/memcached/blob/master/doc/protocol.txt.
[5] Opennfp. http://open-nfp.org/.
[6] Redis. redis.io.
[7] RocksDB. rocksdb.org.
[8] Ryu sdn framework. https://osrg.github.io/ryu/.
[9] Simple p4 nat. http://p4.org/p4/lets-get-started/.
[10] The p4 language specification. 2016. Version 1.1.0.
[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.
[12] 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 CCR*, 44(3):87–95, July 2014.
[13] 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 *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.
[14] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.
[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.
[16] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.
[17] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark. Kinetic: Verifiable dynamic network control. In *NSDI*, pages 59–72, 2015.
[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.
[19] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX NSDI*, pages 31–44, Santa Clara, CA, Mar. 2016. USENIX Association.
[20] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://code.facebook.com/posts/296442737213493/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments.
[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.
[22] J. McClurg, H. Hojjat, P. Černỳ, and N. Foster. Efficient synthesis of network updates. In *ACM SIGPLAN Notices*, volume 50, pages 196–207. ACM, 2015.
[23] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 21:1–21:14, New York, NY, USA, 2015. ACM.
[24] 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 *Presented as part of the 10th USENIX NSDI*, pages 385–398, Lombard, IL, 2013. USENIX.
[25] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX NSDI*, pages 117–130, Oakland, CA, May 2015. USENIX Association.
[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *SIGCOMM CCR*, 42(4):323–334, Aug. 2012.
[27] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.
[28] Z. Wu and H. V. Madhyastha. Rethinking cloud service marketplaces. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 134–140, New York, NY, USA, 2016. ACM.
[29] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, June 2014. USENIX Association.
[30] W. Zhou, D. K. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, pages 73–85, 2015.