2018 IEEE 38th International Conference on Distributed Computing Systems

NetRS: Cutting Response Latency in Distributed Key-Value Stores with In-Network Replica Selection

Yi Su, Dan Feng*, Yu Hua, Zhan Shi, Tingwei Zhu
Wuhan National Laboratory for Optoelectronics
Key Laboratory of Information Storage System, Ministry of Education of China
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
Corresponding author: Dan Feng (dfeng@hust.edu.cn)

***Abstract*—In distributed key-value stores, performance fluctuations generally occur across servers, especially when the servers are deployed in a cloud environment. Hence, the replica selected for a request will directly affect the response latency. In the context of key-value stores, even the state-of-the-art algorithm of replica selection still has considerable room for improving the response latency. In this paper, we present the fundamental factors that prevent replica selection algorithms from being effective. We address these factors by proposing NetRS, a framework that enables in-network replica selection for key-value stores. NetRS exploits emerging network devices, including programmable switches and network accelerators, to select replicas for requests. NetRS supports diverse algorithms of replica selection and is suited to the network topology of modern data centers. Compared with the conventional scheme of clients selecting replicas for requests, NetRS could effectively cut the response latency according to our extensive evaluations. Specifically, NetRS reduces the average latency by up to 48.4%, and the 99th latency by up to 68.7%.**

***Keywords*-In-network computing; Key-value store; Replica selection; Response latency;**

I. Introduction

The distributed key-value store is a vital component of modern Web applications [1]–[3]. For such applications, minimizing the response latency is critical due to their interactive nature. Even the poor tail latency in the key-value store may have a dramatic impact on user-perceived latencies because serving only one end-user request typically requires hundreds or thousands of storage accesses [4].

Distributed key-value stores (e.g. Cassandra [5], Dynamo [6], Voldemort [7], Couchbase [8], etc.) generally replicate data over multiple servers to be highly available and reliable. As server performance fluctuations are the norm [9]–[11] (especially in cloud environments where multiple tenants share resources), the replica selection has a direct impact on the response latency of reading request. Considering that the workloads of key-value stores are commonly read dominant [12], the replica selection scheme plays an important role in cutting response latency. Redundant requests [9] (a client issues the same request to multiple replica servers, and uses the response that arrives first) can also help eliminate the impact of the performance variability of servers. However, using redundant requests cannot always reduce the latency [10] since it is a trade-off between system utilization and response latency [13].

In the context of key-value stores, there are several replica selection algorithms [5], [10] that address the dynamically changing performance of servers from both academia and industry. As requests in key-value stores typically access small size data (about 1KB) [4], these replica selection algorithms work in a distributed manner to avoid the latency penalties of network communications or cross-hosts coordinations at the per-request level. With the conventional scheme, each client is one Replica Selection Node (RSNode). An RSNode *independently* selects replicas for requests based on its local information, including the data collected by itself (e.g. the number of pending requests) and/or the server status piggybacked in responses. Piggybacking data in response packets is the typical approach to delivering server status to clients. Piggybacking avoids the overheads of network protocols due to not constructing separate network packets for the status of a few bytes.

When clients are used as RSNodes, there are two factors that reduce the effectiveness of replica selection algorithms. (i) A client is likely to select a poorly-performing server for a request due to its inaccurate estimation of server status. The accuracy of the estimation depends on the recency of client's local information. As clients rely on requests and responses to update local information, the traffic flowing through a client will determine the recency of its local information. Considering that one client typically sees a small portion of the traffic, there will be lots of clients selecting replicas based on stale and limited local information. (ii) Servers may suffer from load oscillations due to "herd behavior" (multiple RSNodes simultaneously choose the same replica server for requests). The occurrence of "herd behavior" is positively correlated to the number of independent RSNodes. Due to the large number of clients in key-value stores, servers are highly likely to suffer from load oscillations. As a matter of fact, even the state-of-the-art algorithm of replica selection, C3 [10], still has considerable room for improving the response latency. It is worth mentioning that the above analysis does not apply to storage systems that are not latency-critical (e.g. HDFS [14]). In such systems,

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00024

clients send requests to centralized end-hosts (e.g. proxy servers) for better replica selection. The latency penalties are negligible because a request commonly reads several megabytes of data.

We propose NetRS to address the factors that prevent replica selection algorithms from being effective. NetRS is a framework that enables the in-network replica selection for key-value stores in data centers. Instead of using clients as RSNodes, NetRS offloads tasks of replica selection to programmable network devices, including programmable switches (e.g. Barefoot Tofino [15], Intel's FlexPipe [16]) and network accelerators (e.g. Cavium's OCTEON [17], Netronome's NFE-3240 [18]). In addition to the control plane programmability with Software Defined Networking (SDN) switches [19], programmable network devices enable the data plane programmability. Specifically, programmable switches are able to parse application-specific packet headers, match custom fields in headers and perform corresponding actions. Network accelerators can perform application-layer computations for each packet with a low-power multi-core processor. As network devices (e.g. switches) are much fewer than end-hosts in data centers, network devices can automatically gather traffic. Hence, NetRS has two advantages over client-based replica selection. First, compared with clients, network device could obtain more recent local information by gathering traffic. Then, as RSNodes, network devices are more likely to choose better replicas for requests. Second, NetRS could reduce the occurrence of "herd behavior" with fewer RSNodes because one network device could select replicas for requests from multiple clients.

Nevertheless, offloading replica selection to data center network is nontrivial due to the following reasons. (i) **Multiple Paths**. Modern data centers typically use redundant switches to provide higher robustness and performance. Thus, a request and its response may flow through different network paths (different sets of switches). However, on one hand, replica selection algorithms may rank replicas according to metrics determined by both the requests and their responses, e.g. the number of pending requests. On the other hand, unbalanced requests and responses flowing through an RSNode can result in bad replica selections. For example, there is an RSNode that sees 80% requests and 20% responses. Due to receiving only 20% responses, this RSNode will select replicas for 80% requests based on the relatively stale status of servers. Hence, NetRS should guarantee that one request and its corresponding response flow through the same RSNode. (ii) **Multiple Hops**. In a data center, a request from a client needs to flow through multiple switches until arriving at the server. Moreover, with the flexibility of SDN forwarding rules, a request could flow through any switches out of the default (shortest) network paths by taking extra hops. Although any hop can be the RSNode for a request, we should carefully determine the placement of RSNodes with comprehensive considerations, including the requirements of the replica selection algorithm, the capacity of the network devices, and the network overheads caused by extra hops.

In summary, our contributions include:

(i) **Architecture of NetRS**. In the context of data center networks that support multipath communications, we design the NetRS framework that enables in-network replica selection for key-value stores. NetRS integrates the strengths of programmable switches and network accelerators by designing flexible formats of NetRS packets and customizing processing pipelines for each network device. Moreover, NetRS could support diverse replica selection algorithms.

(ii) **Algorithm of RSNodes placement**. We propose an algorithm to arrange RSNodes in the modern data center network with a complex topology. Our algorithm first formalizes the placement problem as an Integer Linear Programming (ILP) problem, then determines the placement of RSNodes by solving the ILP.

(iii) **System evaluation**. We evaluate NetRS using simulations in a variety of scenarios. We vary the number of clients, the demand skewness of clients, the system utilization and the service time of servers. Compared with selecting replica by clients, NetRS reduces the mean latency by up to 48.4%, and the 99th latency by up to 68.7%.

II. Overview of NetRS

This section provides an overview of NetRS architecture. We describe the design of NetRS and show how NetRS is suited to the network of modern data center and exploits programmable network devices.

The data center network generally uses a hierarchical topology [20]–[22] as shown in Fig. 1. End-hosts are commonly organized in racks (each rack contains about 20 to 40 end-hosts). End-hosts in a rack connect to a Top of Rack (ToR) switch. A ToR switch connects to multiple aggregation switches for higher robustness and performance. The directly interconnected ToR and aggregation switches fall into the same pod, as do the end-hosts that connect to the ToR switches in the pod. An aggregation switch further connects to multiple core switches. Redundant aggregation and core switches create multiple network paths between two end-hosts that are not in the same rack. Moreover, due to the wide adoption of SDN in data center networks, there is also a centralized SDN controller. The controller connects to all switches via low-speed links.

Programmable switches and network accelerators meet different demands of in-network replica selection. On one hand, a programmable switch provides both the fast packets forwarding and the customizable pipelines of packet processing at data plane. With a customized pipeline, a switch can recognize application-specific packet formats, match custom fields, and perform actions like adaptive routing and header modifications. However, in order to keep the high speed of packet forwarding, the programmable switch only supports

simple operations, e.g. reading from memory, writing to memory, etc. On the other hand, a network accelerator is able to handle complicated computations with a multicore (or manycore) processor and several gigabytes of memory. However, network accelerators fail to work as switches due to lower routing performance and fewer ports.

We have the following considerations for designing the NetRS framework. (i) NetRS should keep things in network as much as possible. Clients and servers of key-value store should be able to take advantage of NetRS without complicated interactions with NetRS components. (ii) NetRS should work in a distributed manner without coordinating network devices at per-request level. The frequent coordination is unrealistic due to introducing significant overheads. (iii) NetRS should minimize its impacts on other applications and limit its bandwidth overheads since multiple applications share the data center network.

NetRS is a hardware/software co-design framework that enables in-network replica selection for key-value stores. In brief, NetRS relies on programming switches to adaptively route packets of key-value stores and leaves application-layer computations (e.g. replicas ranking) to network accelerators. NetRS does not exclusively use either programmable switches or network accelerators for following reasons. (i) Compared with network accelerators, programmable switches could forward packets based on field matching more efficiently. (ii) Although the computing power of programmable switches may grow over time, application-layer computations on switches could block the packet forwarding of other applications.

Fig. 1 shows the NetRS architecture. NetRS consists of two kinds of components: the NetRS controller and the NetRS operator. The NetRS operator is a collection of hardware and software. The hardware part consists of a programmable switch and network accelerator(s) attached to the switch. The software part includes NetRS rules, the NetRS monitor, and the NetRS selector. While all switches have NetRS rules, the NetRS monitor only resides on the ToR switch, and the NetRS selector runs on the network accelerator. When a packet arrives, the switch determines the next hop for the packet according to NetRS rules (detailed in Section IV-B). In the case that a switch forwards the packet (or the packet's clone) to the network accelerator, the NetRS selector would (i) choose a replica server for the packet if the packet is a request of key-value store or (ii) use the packet to update local information if the packet is a response of key-value store (detailed in Section IV-C). The NetRS monitor collects traffic statistics and sends them to the NetRS controller (detailed in Section IV-D). According to the traffic statistics and pre-given constraints, the NetRS controller periodically generates a Replica Selection Plan (RSP), which is the placement of RSNodes (detailed in Section III). In order to deploy an RSP, the NetRS controller has to update NetRS rules for each NetRS operator. As

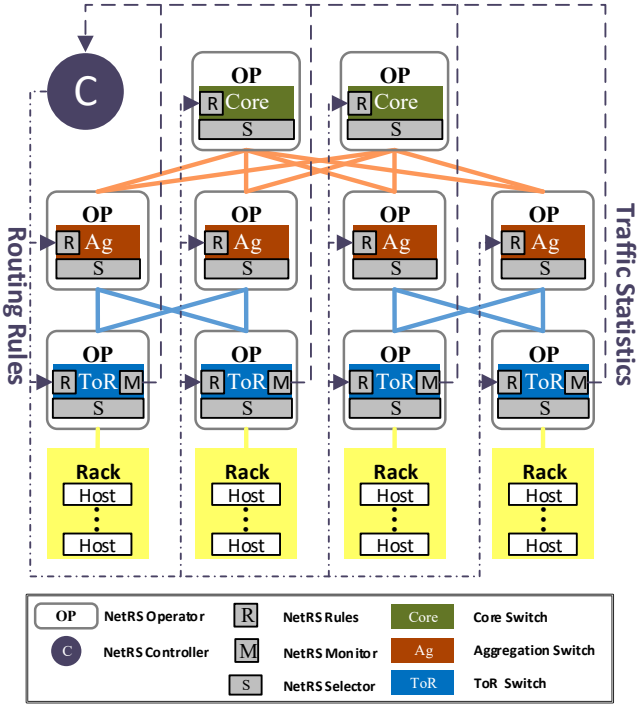

Figure 1. An overview of the NetRS architecture.

the newly introduced RSNodes have to build the view of the system status from scratch, the deployment of a new RSP may lead to a temporary latency increase. The time it takes for the system to stabilize again depends on many factors, including the rate of convergence of the replication selection algorithm, the number of new RSNodes, and the service rate of servers. Thanks to the stable workload of user-facing applications [23], the NetRS controller does not need to update RSP frequently.

III. NetRS Controller

In this section, we present the NetRS controller, which determines the NetRS operator working as Replica Selection Node (RSNode) for a request. We first state the problem of RSNodes placement (Section III-A), and then describe the algorithm solving the problem (Section III-B). The NetRS controller also ensures the high availability of NetRS with an exception handling mechanism (Section III-C).

A. RSNodes Placement Problem

In NetRS, we divide requests into different traffic groups. The Replica Selection Plan (RSP) specifies the NetRS operator that works as the RSNode for requests of each traffic group. The granularity of dividing requests is a key aspect of the RSP, and the typical candidates of the granularity are: (i) *request-level group* (one request as a group), (ii) *host-level group* (requests from the same host as a group), (iii) *rack-level group* (requests from the same rack as a group). Finer-grained traffic groups provide more flexibility in making the

RSP. However, finer-grained traffic groups (i) require more efforts to find the optimal RSP due to larger solution space, and (ii) introduce more overheads when carrying out RSP in data center network because of network devices dealing with more cases. In fact, for the request-level group, per-request level coordinations are unavoidable because every request introduces a new group to the RSP. Hence, NetRS does not consider the scenario of using request-level group.

In this paper, we focus on the scenario of dividing requests based on host-level groups, rack-level groups or any intervening-level groups (requests from several end-hosts in the same rack as a group). The NetRS controller determines the RSP according to statistics of each traffic group.

In order to determine the RSP, we have to solve the optimization problem of assigning each traffic group's RSNode to a NetRS operator. We set the following optimization goals to cope with the two factors (detailed in Section I) that have a significant impact on the effectiveness of replica selection algorithms, like C3 [10].

- *Goal 1*: Maximizing the recency of local information for an RSNode.
- *Goal 2*: Minimizing the occurrence of "herd behavior".

The NetRS controller achieves *Goal 1* and *Goal 2* by minimizing the number of RSNodes. First, the average traffic flowing through one RSNode will increase with fewer RSNodes. As RSNodes use requests and responses to update local information, an RSNode could obtain the more recent information on average. Second, as the occurrence of "herd behavior" has a positive correlation with the number of RSNodes, we could avoid "herd behavior" as much as possible by minimizing the number of RSNodes.

There are also constraints as follows:

- *Constraint 1*: There should be only one RSNode for each request.
- *Constraint 2*: The utilization of each network accelerator should be limited.
- *Constraint 3*: The total amount of extra hops to RSNodes that requests take should be limited.

Constraint 1 exists because replica selection algorithms typically rely on metrics correlated with decisions of replica selection (e.g. the number of a server's pending requests). Hence performing replica selection multiple times for one request could make the RSNode, whose decision is not the final one, uses incorrect input values to select replicas for following requests. Furthermore, selecting replica for each request at multiple NetRS operators introduces unnecessary latency overheads. It is because the request has to wait for replica selection multiple times while the final RSNode overwrites all previous decisions. We use *Constraint 2* to adapt the load of each network accelerator to its capacity. High utilization of a network accelerator will make requests wait a long time for replica selection. *Constraint 3* enables the trade-off between the flexibility of making RSP and the network overheads of taking extra hops. If the RSNode for requests of a traffic group is located in a NetRS operator, which is out of default network paths of these requests, the requests should take extra hops to reach the RSNode. Extra hops introduce latency overheads and occupy extra resources of the shared data center network.

B. Replica Selection Plan

We formalize the problem of RSNodes placement as an Integer Linear Programming (ILP) problem. The NetRS controller can determine RSP by solving the ILP problem with an optimizer (e.g. Gurobi [24], CPLEX [25]).

Suppose P is a binary matrix that shows the RSP. If we perform replica selection for requests of the traffic group g_i at the NetRS operator o_j, P_{ij} will be set to 1, and 0 otherwise. D is a binary vector that shows the distribution of RSNodes among all NetRS operators. If a NetRS operator o_j works as an RSNode for requests of any traffic group, then D_j will be set to 1, otherwise 0.

Suppose R is a binary matrix that describes the relationship between traffic groups and NetRS operators, for a traffic group g_i and a NetRS operator o_j, if o_j is in default network paths that are between the end-host of g_i and any end-host of another pod, R_{ij} will be set to 1, otherwise 0. In the multi-tier topology of the data center network described in Section II, suppose end-hosts of the traffic group g_i connect to the ToR switch s_{gi}. We could determine R_{ij} based on following rules: (i) if o_j is in the tier of core switches, then $R_{ij} = 1$; (ii) if o_j is in the tier of aggregation switches, $R_{ij} = 1$ only when o_j and s_{gi} are in the same pod, and $R_{ij} = 0$ otherwise; (iii) if the o_j is in the tier of ToR switches, then R_{ij} will be set to 0 except that the switch of o_j is s_{gi}, which makes $R_{ij} = 1$. T is a matrix that describes the traffic composition of each traffic group. In the multi-tier network described in Section II, we define the tier ID of a NetRS operator as the minimum number of connections between the NetRS operator and any node in the top tier (the tier of core switches is the top tier). According to the highest tier that requests flow through with the default network paths, the requests of a traffic group fall into 3 categories: the *Tier*-2 traffic (communication between end-hosts in the same rack), the *Tier*-1 traffic (communication between end-hosts in the same pod but in different racks), and the *Tier*-0 traffic (communication between end-hosts in different pods). For a traffic group g_i, T_{ik} is its *Tier*-k traffic. We can determine R according to the network topology, and get T from traffic statistics collected by NetRS monitors (Section IV-D).

Suppose a NetRS operator o_j could perform replica selection without introducing significant delay if the utilization of its network accelerator is under U_j. Then the maximum traffic T_j^{max}, which use the NetRS operator o_j as the RSNode, should be under $U_j c_j^{ac}/t_j^{ac}$, where c_j^{ac} is the number of cores in the accelerator and t_j^{ac} is the mean

service time of selecting replica. We limit the total amount of extra hops by a constant E. When calculating the number of extra hops, we consider the difference of total forwarding times between going through the RSNode and going directly to the server. For example, for *Tier*-2 traffic, if the RSNode lies in the tier of core switches, then the amount of extra hops for one request is 4 (a request will be forwarded once to get to the server with the default network path, and going to the RSNode makes it be forwarded 5 times, hence the extra hops of the request is $4 = 5 - 1$). We can get T_j^{max} and E from system administrators, who determine the values based on their demands.

Suppose $t(x)$ is a function that returns the tier ID of a NetRS operator o_x or a traffic group g_x (the g_x's tier ID is same to the tier ID of the NetRS operator, to which end-hosts of g_x directly connects), and $h(i,j) = t(i) - t(j)$.

The description of the ILP problem is as follows.

$$Minimize : \sum D_j \tag{1}$$

$$Subjects\ to :$$

$$\forall i, \forall j : P_{ij} \in \{0,1\}, D_j \in \{0,1\} \tag{2}$$

$$\forall i, \forall j : D_j - P_{ij} \geq 0 \tag{3}$$

$$\forall i, \forall j : R_{ij} - P_{ij} \geq 0 \tag{4}$$

$$\forall i : \sum P_{ij} = 1 \tag{5}$$

$$\forall j : \sum (P_{ij} \sum_{k=0}^{t(i)} [T_{ik}]) \leq T_j^{max} \tag{6}$$

$$\sum (P_{ij} \sum_{k=0}^{h(i,j)-1} [2(h(i,j)+k)T_{i(t(i)-k)}]) \leq E \tag{7}$$

Among the constraints of the ILP problem, Equation (2) suggests that P and D contain only binary elements, Equation (3) guarantees that a NetRS operator is considered as an RSNode if it selects replica for any traffic group, Equation (4) reduces the solution space by forbidding a request to flow from the tier to its lower tier before the request reaching its RSNode. Such restriction help to avoid extra hops that form loops between tiers. Finally, Equation (5) , (6) and (7) correspond to *Constraint 1*, *Constraint 2* and *Constraint 3*, respectively. We could get a suboptimal solution to the ILP problem by terminating the solving process early. Hence, by limiting the solving time, we can make a trade-off between the recalculation expense and the optimality of the RSP. Besides the 3-tier topology shown in Fig. 1, our algorithm is applicable to n-tier ($n \in \{1, 2, ...\}$) tree-based topologies of data center network.

In order to accommodate different RSPs, every programmable switch must have a network accelerator. In fact, programmable switches and accelerators are widely adopted in the data center network for various purposes (e.g. deep packet inspection [17], [18], firewalls [17], [18], in-network cache [20], packets sequencing [26], [27]). Due to using a separate traffic threshold T_j^{max} for each network accelerator, our algorithm of RSNodes placement could adapt to the scenarios of sharing accelerators with other applications. NetRS could effectively exploit the underloaded accelerators by setting higher traffic thresholds for them. In the scenario of NetRS using dedicated accelerators, we could cut the network cost of NetRS by connecting one accelerator to multiple switches. Such configuration is feasible as *Constraint 1* suggests that there should be only one RSNode among all NetRS operators in a network path. In this scenario, Equation (6) will change to $\forall J : \sum_{j \in J} \sum (P_{ij} \sum_{k=0}^{t(i)} [T_{ik}]) \leq T_J^{max}$, where J is a set of switches connected to the same accelerator.

C. Exception Handling Mechanism

NetRS uses a mechanism of Degraded Replica Selection (DRS) to handle exceptions. The DRS requires that clients of the key-value store should provide a target replica for each request as a backup. If the DRS for a request is enabled by the NetRS controller, NetRS will route the request to the backup replica provided by the client. The NetRS controller enables the DRS for each traffic group independently by updating NetRS rules of NetRS operators without interactions with end-hosts. Currently, the DRS is necessary for the following scenarios. (i) No feasible RSP exists. If there are some traffic groups that we cannot find NetRS operators as their RSNodes without violating the constraints, the ILP problem in Section III-B would have no feasible solution. In this case, we could enable the DRS for some traffic groups so that a feasible RSP exists for the rests. Considering that enabling DRS for a traffic group will lead to additional RSNodes, the number of traffic groups using the DRS should be as small as possible. Moreover, the traffic of a group using DRS should be high to prevent clients from selecting poorly-performing replica servers, which hurts the tail latency. Hence, the NetRS controller turns DRS on for groups with the highest traffic. (ii) A NetRS operator does not work as expected, e.g the NetRS operator is overloaded due to load changes. The NetRS controller will enable the DRS for traffic groups that use the NetRS operator as RSNode. (iii) The NetRS operator working as an RSNode fails.

IV. NetRS Operator

This section describes the NetRS operator. We first introduce the packet format of NetRS (Section IV-A). Then, we show the processing pipeline of a programmable switch according to NetRS rules (Section IV-B), and the working procedure of a NetRS selector running on the network accelerator (Section IV-C). Finally, we present how NetRS monitors collect traffic statistics (Section IV-D).

A. Packet Format

The packet format plays an important role in propagating information. Clients, servers, switches and network accel-

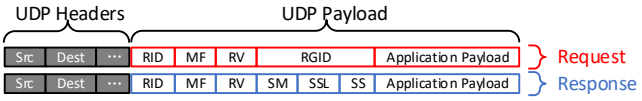

Figure 2. Packet format of NetRS for the request and the response.

erators should agree to the common format. As stateful network protocols (e.g. TCP) introduce latency overheads, recent key-value stores [20], [28] generally use stateless network protocols (e.g. UDP). Key-value stores in production environments also exploit UDP-based network protocols to cut latency overheads for reading requests [4]. Considering that the goal of NetRS is to reduce the read latency in key-value stores, we design the packet format of NetRS in the context of UDP-based network protocols. Moreover, network devices could parse packets more efficiently with UDP protocol due to not maintaining per-flow state. There are two design requirements for the packet format. (i) It should be flexible and adapt to diverse replica selection algorithms. (ii) It should keep protocol overheads low.

NetRS packets are carried in the UDP payload. In order to reduce bandwidth overheads of NetRS protocol, we use separate packet formats for request and response to carry different information. Fig. 2 shows the packet format of request and response, respectively. The request and response packet have following common segments:

- **RID** (RSNode ID): [*2 bytes*] The ID of a NetRS operator, which works as the RSNode for a request or the corresponding request of a response.
- **MF** (Magic Field): [*6 bytes*] A label that used by switches to determine the type of a packet.
- **RV** (Retaining Value): [*2 bytes*] A value set by the RSNode for a request, and the value in a response will be the same with the value in its corresponding request. An RSNode could exploit this segment to collect request-level data. For example, an RSNodes may set the retaining value of a request using the timestamp of the request sending, and then the RSNode will know the response latency of the request when its corresponding response arrives. The usage of this segment depends on the needs of the replica selection algorithm.
- **Application Payload**: [*variable bytes*] The content of a request or a response.

The segment only in the request packet is as follows:

- **RGID** (Replica Group ID): [*3 bytes*] The ID of a replica group. A NetRS selector could obtain the replica candidates for a request by querying its local database of replica groups with the RGID. The size of the database should be small because key-value stores typically use consistent hashing to place data. The advantage of using RGID is to keep the headers of a packet fixed-sized and irrelevant to the number of replicas. The fixed-sized headers are more friendly for switches to parse packets.

Segments only in the response packet are as follows:

- **SM** (Source Marker): [*4 bytes*] A value indicating the network location from which a response comes.
- **SSL** (Server Status Length): [*2 bytes*] The length of the piggybacked status of the server in a response.
- **SS** (Server Status): [*variable bytes*] The piggybacked status of the server in a response.

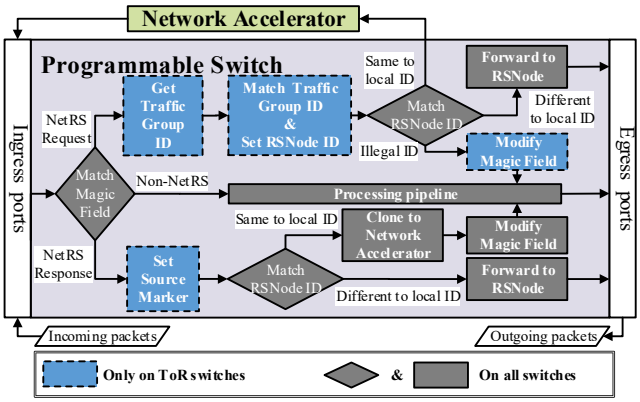

Figure 3. NetRS rules within ingress pipelines of a programmable switch.

B. NetRS Rules

The NetRS controller updates NetRS rules of each NetRS operator based on the periodically generated RSP. Each NetRS operator relies on its NetRS rules to forward a packet to the right place. The processing pipeline of a programmable switch includes two stages: ingress processing and egress processing. NetRS rules are a part of the ingress processing pipeline. Fig. 3 shows the procedure of ingress processing according to NetRS rules. Packets fall into 3 categories, including the non-NetRS packet, the NetRS request, and the NetRS response. The switch uses the segment of the magic field in a packet to determine the type of the packet. A non-NetRS packet will directly enter the regular ingress processing pipeline and go towards its target server. A switch only applies NetRS rules to the NetRS packet, including the NetRS request and response.

The NetRS controller assigns a unique ID (a positive integer) to each NetRS operator and uses this ID to represent each NetRS operator in the RSP. The NetRS operator stores its ID locally in the programmable switch. The segment of RSNode ID in a NetRS packet stores the ID of a NetRS operator that works as the RSNode. When a NetRS packet arrives, the programmable switch will firstly match the packet's RSNode ID segment. If the RSNode ID is different to the local ID in the switch, then the switch will forward the packet to the next hop towards the RSNode. Otherwise, if the RSNode ID is the same with the local ID, the switch will perform corresponding operations based on packet type. If the packet is a NetRS request, it will be forwarded to

the network accelerator which runs the NetRS selector. The network accelerator will transform the NetRS request to a non-NetRS packet, and send the packet back to the switch (Section IV-C). Otherwise, if the packet is a NetRS response, the switch will firstly send a clone of the packet to the network accelerator, and then push the packet to the regular pipeline of ingress processing with a modified magic field of M_{mon}, which also labels it as a non-NetRS packet. The magic field of M_{mon} makes the packet recognizable by NetRS monitors (Section IV-D). By cloning the packet of NetRS response, we could avoid the latency overhead of network accelerator processing the packet.

As the RSP and traffic groups are agnostic to end-hosts, clients of key-value stores are unable to determine the RSNode ID for a NetRS request. With the network topology described in Section II, NetRS uses ToR switches to set the RSNode ID for each NetRS request. Compared with switches of other types, a ToR switch has extra NetRS rules for NetRS requests, which could (i) match the source IP of a packet and get the traffic group ID, and (ii) set the RSNode ID according to the traffic group ID. For the NetRS response, the ToR switch has NetRS rules to set the segment of source marker, which is required by the NetRS monitor (Section IV-D). A NetRS response does not need to obtain the RSNode ID from the ToR switch because the server will copy the RSNode ID from its corresponding request to the packet of NetRS response.

In order to enable the degraded replica selection for a traffic group (Section III-C), the NetRS controller just tells the corresponding NetRS operator to set an illegal RSNode ID (e.g. -1) to packets of the traffic group. If a NetRS packet has an illegal RSNode ID, the ToR switch will label it as a non-NetRS packet by setting $f(M_{mon})$ to its magic field, where $f(\cdot)$ is an invertible function.

C. NetRS Selector

The NetRS selector is responsible for performing replica selection and maintaining corresponding local information. Due to residing on network accelerators, the NetRS selector could use an arbitrary replica selection algorithm without considering the limitations of programmable switches.

For a NetRS request, the NetRS selector determines the target replica server for the packet based on local information. When a NetRS request arrives from the co-located switch, the NetRS selector will first extract the Replica Group ID from the packet. Then the NetRS selector looks up the local database to determine replica candidates and selects a replica from the candidates. The NetRS selector will rebuild the packet with the selected replica server and the necessary retaining value. Moreover, while rebuilding the packet, the NetRS selector also specifies the magic field to $f(M_{resp}), f(M_{resp}) \neq M_{req}, M_{resp}$, where M_{req} and M_{resp} are constant values that label the NetRS request and response, respectively. The server will set the magic field in the NetRS response to $f^{-1}(m)$, where m is the magic field value of the corresponding request. This mechanism guarantees that (i) the server marks a response packet as a NetRS response, only if the packet's corresponding request had flowed through a NetRS selector; (ii) the NetRS monitor could recognize the response of a request using degraded replica selection. Finally, the NetRS selector will send the rebuilt packet to the switch.

For a NetRS response, the NetRS selector will update local information according to the piggybacked information in the packet and then abandon the packet.

D. NetRS Monitor

The NetRS monitor is in charge of collecting statistics on traffic composition of each traffic group. We deploy the NetRS monitor as a bunch of match-action rules in egress pipelines of the ToR switch.

We should answer two questions for designing the NetRS monitor. First, when should the data collection happen (the time point that a packet enters or leaves the network)? Second, what kind of packets (requests or responses) should the NetRS monitor concern? In NetRS, we choose to collect data when a response leaves the network. The reasons are as follows. (i) A request does not carry the replica selected by NetRS when it first enters the network. (ii) For a ToR switch, requests leaving the network may be of any traffic group, so are responses that first enters the network. Considering that each traffic group requires separate match-action rules (counters), collecting such packets will introduce too many burdens to a switch. In comparison, responses leaving the network are of traffic groups associated with the rack.

The NetRS monitor filters packets based on the magic field. NetRS rules ensure that the NetRS monitor can recognize responses of key-value store. When a response enters the egress pipeline of a ToR switch, the NetRS monitor first determines the traffic group based on its destination IP. Then, the monitor updates the corresponding counter according to the source marker. Each ToR switch has a unique source marker that depends on its network location. A source marker contains two components: the pod ID and the rack ID. A ToR switch could determine whether a packet is from the same pod and/or the same rack by comparing the source marker in the packet to the local one. It is worth mentioning that the source IP of a packet may work as the source marker if the IP has the pattern indicating the network location of an end-host.

V. Evaluation

We conduct simulation-based experiments to extensively evaluate the NetRS framework. Our evaluation reveals the impact of different factors on the effectiveness of NetRS cutting response latency.

A. Simulation Setup

In our experiments, we use the simulator from C3 [10], which simulates clients and servers of key-value stores. In order to evaluate the NetRS framework, we extend the simulator to simulate network devices. The simulated network is a 16-ary fat-tree (3-tier) [22] containing 1024 end-hosts.

We set major parameters in our evaluation based on the experimental parameters in C3 [10]. Specifically, the service time of a server follows exponential distribution while the mean value (t^{kv}) is 4ms. Each server could process $N_p (N_p = 4)$ requests in parallel. The performance of each server fluctuates independently with an interval of 50ms. The fluctuation follows the bimodal distribution [29] with the range parameter $d = 3$ (in each fluctuation interval, the mean service time could be either t^{kv} or t^{kv}/d with equal possibility). Keys are distributed across $N_s (N_s = 100)$ servers according to consistent hashing with a replication factor of 3. There are 200 workload generators in total, and each workload generator creates reading requests based on the Poisson process, which could approximate the request arrival process of Web applications with reasonably small errors [30]. For a request, the workload generator chooses an accessing key out of 100 million keys according to the Zipfian distribution (the Zipf parameter is 0.99). For each experiment, the key-value store receives 6 million requests. By default, there are 500 clients sending requests without the demand skewness (in other words, the number of requests issued by each client is evenly distributed).

We set the parameters of network devices based on the measurements of real-world programmable switches and network accelerators in the paper of IncBricks [20]. Specifically, the RTT (Round-Trip Time) between a switch and its attached network accelerator is 2.5us. We consider using low-end network accelerators. Each accelerator has 1 core and the processing time is 5us. The network latency between two switches that are directly connected is 30us.

We perform the simulation with the above parameters in all cases, unless otherwise noted. The clients and servers are randomly deployed across end-hosts [31], and each host only has one role [32]. We repeat every experiment 3 times with different deployments of clients and servers. We compare the following schemes (in all schemes, RSNodes select replica using the C3 algorithm [10], which is state-of-the-art).

- **CliRS**: A commonly used replica selection scheme in key-value stores [5]–[8]. With CliRS, clients work as RSNodes and perform replica selection for requests.
- **CliRS-R95**: For primary requests, CliRS-R95 is the same as CliRS. However, if a primary request has been outstanding for more than the 95th-percentile expected latency, the client will send a redundant request [9].
- **NetRS-ToR**: Using the NetRS framework for replica selection with a straightforward RSP, which simply specifies the NetRS operator co-located with the rack's ToR switch as the RSNode for requests from the rack.
- **NetRS-ILP**: Using the NetRS framework for replica selection with an RSP determined by solving the ILP problem of RSNodes placement. For a request, the replica selection happens at the NetRS operator specified by the RSP. As an example, an RSP from NetRS-ILP consists of 6 RSNodes on aggregation switches and 1 RSNode on a core switch.

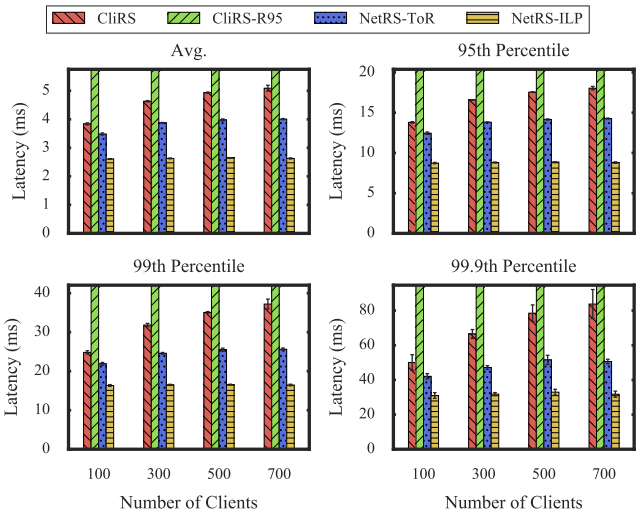

Figure 4. The comparison of performance with varying number of clients.

B. Results and Analysis

This section provides experimental results in a variety of scenarios. We use the open-loop workload, which is in line with the real-world workloads of Web applications [33]. The aggregate arriving rate of requests (A) corresponds the 90% system utilization ($\frac{t^{kv}A}{N_s N_p}$), which is low considering the performance fluctuation ($\frac{2}{1+d}\frac{t^{kv}A}{N_s N_p} = 45\%$). In our deployment, $U = 50\%$ and $E = 20\%A$, where U is the maximum utilization of a network accelerator, and E is the maximum amount of extra hops.

In most cases, using CliRS-R95 will result in a dramatic increase in response latency. It is because the extra loads of redundancy will make a small portion of servers overloaded due to the skewed workloads. Fig. 4 , 5 , 6 and 7 do not show bars exceeding the respective latency thresholds.

1) Impact of the number of clients: Fig. 4 shows the response latency comparison of all schemes when the number of clients ranges from 100 to 700. We observe the following things. (i) Both NetRS-ILP and NetRS-ToR outperform CliRS, and NetRS-ILP shows the best performance. Compared with CliRS, NetRS-ILP reduces the mean latency by 32.0%-48.4% and the 99th latency by 34.2%-55.8%. Compared with NetRS-ToR, NetRS-ILP reduces the mean latency by 31.3% and the 99th latency by 32.3% on average. (ii) With CliRS, both the mean and tail latency increase as

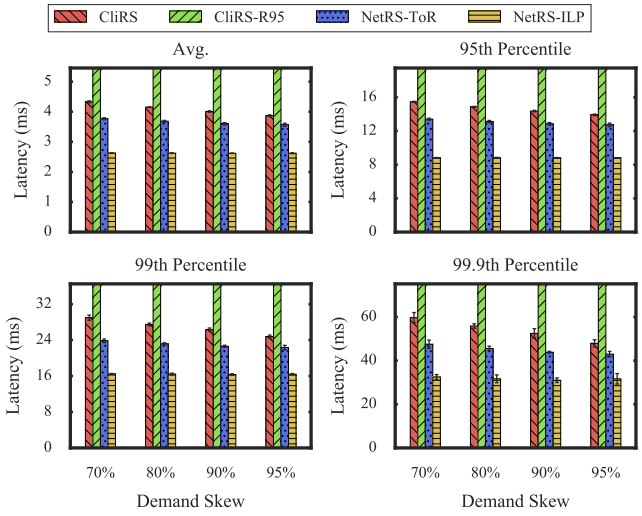

Figure 5. The comparison of performance with varying demand skewness.

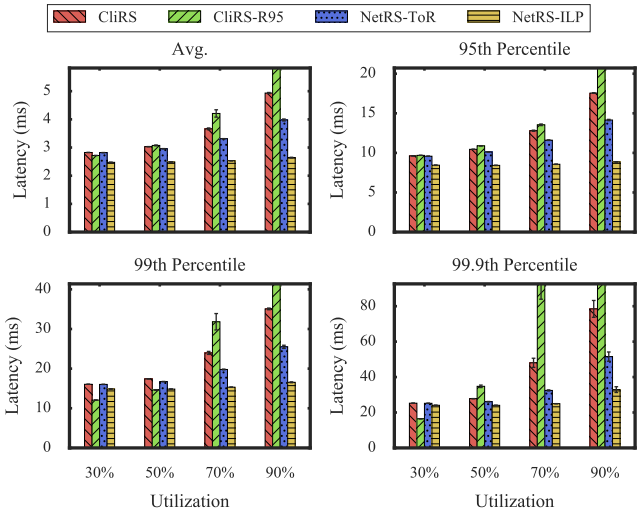

Figure 6. The comparison of performance with varying system utilization.

the number of clients grows. However, the response latency roughly remains unchanged with NetRS-ILP and NetRS-ToR regardless of changes in the number of clients. The underlying reason is that, with NetRS-ILP and NetRS-ToR, the number of RSNodes is irrelevant to the number of clients. Since each client works as an RSNode with CliRS, these experiments also validate our analysis that more independent RSNodes could lead to worse replica selection, which leads to performance penalties.

2) Impact of the demand skewness: Fig. 5 depicts the influence of demand skewness on response latency with different schemes of replica selection. In the experiments, we measure the demand skewness with the percentage of requests issued by 20% clients. Excluding CliRS-R95, CliRS and NetRS-ILP still provide the worst and best performance, respectively. However, as the demand skewness increases, the latency reduction introduced by the NetRS framework tends to decrease. For example, in the scenario of no demand skewness shown in Fig. 4 (500 clients), NetRS-ILP reduces the mean and 99th latency by 46.4% and 52.8%, respectively. However, when the demand skewness is 70%, NetRS-ILP reduces the mean and 99th latency by 39.2% and 43.1%, respectively. For the demand skewness of 95%, NetRS-ILP introduces only 32.2% reduction in the mean latency and 33.8% reduction in the 99th latency. The reasons for this phenomenon are as follows. On one hand, CliRS could provide lower response latency with heavier demand skewness due to high-demand clients dominating the system performance. With CliRS, the number of RSNodes would be reduced to the number of high-demand clients thanks to the skewed demand. On the other hand, since high-demand clients spread over different network locations, the demand skewness of clients does not imply the traffic skewness of switches. Therefore, the NetRS framework could benefit only a little from the demand skewness.

3) Impact of the system utilization: Fig. 6 shows the impact of the system utilization on response latency for all schemes of replica selection. We run the experiments with the system utilization ($\frac{t^{kv}A}{N_s N_p}$) ranging from 30% to 90%. Compared with CliRS, NetRS-ILP reduces the mean latency by 12.4%-46.4% and reduces the 99th latency by 7.4%-52.8%. Compared with NetRS-ToR, NetRS-ILP reduces the mean and 99th latency by 12.2%-33.5% and 7.2%-35.1%, respectively. We have the following observations. (i) With all schemes, the response latency increases as the system utilization grows. It is because the higher utilization suggests the more severe contention of resources and the longer queueing latency, which none of these schemes could avoid. (ii) Compared with CliRS and NetRS-ToR, NetRS-ILP introduces more reduction in response latency in the region of higher utilization. The underlying reason is that the severe contention of resources will amplify the impact of bad replica selection on the response latency. (iii) CliRS-R95 outperforms other schemes in cutting the tail latency when the utilization is low. With low utilization, the impact of extra loads due to issuing redundant requests is negligible.

4) Impact of the service time: Fig. 7 depicts the response latency comparison of all schemes when the server's mean service time varies from 0.1ms to 4ms. It is obvious that the response latency would be shorter if the server could process requests faster regardless of the replica selection scheme. These experiments aim to reveal the difference of the NetRS-based schemes on cutting the response latency with different service time. Compared with CliRS, NetRS-ILP introduces fewer reductions in mean latency with a lower service time. However, there is no such phenomenon in terms of the tail latency or with NetRS-ToR. The reasons are as follows, (i) NetRS-ILP introduces the latency overheads of taking extra

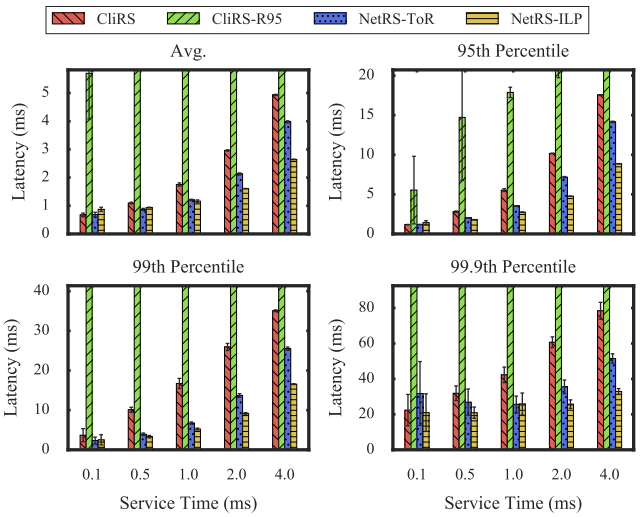

Figure 7. The comparison of performance with varying service time.

hops, and (ii) NetRS-ILP leads to the longer waiting time in RSNodes due to higher utilization of RSNodes. Considering that the mean latency is comparable to the service time, these latency overheads are significant compared with the mean latency when the service time is low. Since the tail latency is typically orders of magnitude greater than the service time, the latency overheads are negligible for the tail latency.

In summary, (i) the NetRS framework could effectively cut the response latency in key-value stores compared with selecting replica by clients; (ii) our ILP-based algorithm of RSNodes placement makes a significant contribution to the latency reduction of NetRS; (iii) redundant requests are only suitable for scenarios of low utilization.

VI. Related Work

Tolerating Performance Variability: The approaches to dealing with the time-varying performance of servers fall into two categories: redundant requests and replica selection. On one hand, redundant requests are used pervasively to reduce response latency. Google proposes to reissue requests to reduce latency and use cross-server cancellations to reduce redundancy overheads [9]. Vulimiri et al. [13] suggest that the use of redundant requests is a trade-off between response latency and system utilization. Shah et al. [34] and Gardner et al. [35] provide theoretical analyses on using redundant requests to reduce latency. On the other hand, replica selection is also an indispensable part of distributed systems. Mitzenmacher [36] proposes the "power of two choices" algorithm, which sends a request to the server with a shorter queue out of two randomly chosen servers. Dynamic Snitching [5] is the default replica selection strategy of Cassandra, which selects replica based on the history of reading latencies and I/O loads. C3 [10] is the state-of-the-art algorithm of replica selection, which could effectively reduce tail latency compared with other algorithms. These works are orthogonal to NetRS. NetRS focuses on improving the effectiveness of diverse replica selection algorithms via performing replica selection in data center network.

Mayflower [37] selects the replica server and the network path collaboratively using the SDN technique for distributed filesystems, e.g. HDFS [14]. Mayflower considers the scenario that a request commonly reads several megabytes or even gigabytes of data. Hence, a client could connect a centralized server for replica/network path selection for each request. NetRS addresses the replica selection problem under the scenario of key-value stores. Due to the small size request (about 1KB), replica selection should be lightweight and performed in a distributed manner.

In-Network Computing: In-network computing is widely used to enhance the performance of key-value stores. NetKV [38] introduces a network middlebox that enables adaptive replication. SwitchKV [28] balances loads of back-end servers by routing hot requests to a high-performance cache with SDN. NetCache [39] shares the same goal with SwitchKV, however, NetCache cache hot data within programmable switches instead of a dedicated cache layer. IncBricks [20] builds an in-network cache system that works as a cache layer for key-value stores. Different from NetRS, which focuses on dealing with the performance fluctuation of servers, these works leverage in-network computing to address the problem of workload skewness for key-value stores.

VII. Conclusion

This paper presents NetRS, a framework that enables in-network replica selection for key-value stores in data centers. NetRS exploits programmable switches and network accelerators to aggregate tasks of replica selection. Compared with selecting replica by clients, NetRS significantly reduces the response latency. NetRS could support various replica selection algorithms with the flexible format of NetRS packet and the customized processing pipelines of each network device. We formalize the problem of RSNodes placement with ILP in the context of modern data center network with complex topology. Moreover, NetRS could be highly available through a mechanism that handles exceptions, e.g. failures of network devices.

Acknowledgments

This work is supported by NSFC No. U1705261, No.61772222, No.61772212, Shenzhen Research Funding of Science and Technology - Fundamental Research (Free exploration) JCYJ20170307172447622. We thank our shepherd Zhenhua Li and the anonymous reviewers for their constructive comments. We also thank Yajing Zhang for helping us improve the presentation.

REFERENCES

[1] H. Xiao, Z. Li, E. Zhai, T. Xu, Y. Li, Y. Liu, Q. Zhang, and Y. Liu, "Towards Web-Based Delta Synchronization for Cloud Storage Services," in *USENIX FAST*, 2018.

[2] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai, "Efficient Batched Synchronization in Dropbox-Like Cloud Storage Services," in *Springer Middleware*, 2013.

[3] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, "Towards Network-level Efficiency for Cloud Storage Services," in *ACM IMC*, 2014.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, and P. Saab, "Scaling memcache at facebook," in *USENIX NSDI*, 2013.

[5] "Apache Cassandra Database," http://cassandra.apache.org, 2017.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *ACM SOSP*, 2007.

[7] "Project Voldemort: A Distributed Database," http://www.project-voldemort.com/voldemort, 2017.

[8] "Couchbase Data Platform: Couchbase Server," https://www.couchbase.com/products/server, 2017.

[9] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[10] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *USENIX NSDI*, 2015.

[11] Y. Hua, "Cheetah: An Efficient Flat Addressing Scheme for Fast Query Services in Cloud Computing," in *IEEE INFOCOM*, 2016.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *ACM SIGMETRICS*, 2012.

[13] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low Latency via Redundancy," in *ACM CoNEXT*, 2013.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE MSST*, 2010.

[15] "Barefoot Tofino: P4-programmable Ethernet Switch ASICs," https://barefootnetworks.com/products/brief-tofino/, 2017.

[16] "Intel Ethernet Switch FM6000 Series, white paper," 2013.

[17] "Octeon Multi-Core MIPS64 Processor Family," https://www.cavium.com/octeon-mips64.html, 2017.

[18] "Netronome NFE-3240 Family Appliance Adapters," https://www.netronome.com/products/nfe/, 2017.

[19] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[20] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *ACM ASPLOS*, 2017.

[21] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," *Communications of the ACM*, vol. 59, no. 9, pp. 88–97, 2016.

[22] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *ACM SIGCOMM*, 2008.

[23] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, "History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters," in *USENIX OSDI*, 2016.

[24] "Gurobi Optimizer: State of the Art Mathematical Programming Solver," http://www.gurobi.com/products/gurobi-optimizer, 2017.

[25] "Cplex Optimizer: High-performance Mathematical Programming Solver," https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer, 2017.

[26] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering," in *USENIX OSDI*, 2016.

[27] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control," in *ACM SOSP*, 2017.

[28] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be Fast, Cheap and in Control with SwitchKV," in *Usenix NSDI*, 2016.

[29] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, 2010.

[30] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power Management of Online Data-intensive Services," in *ACM ISCA*, 2011.

[31] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM IMC*, 2010.

[32] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *ACM SIGCOMM*, 2015.

[33] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *IEEE IISWC*, 2016.

[34] N. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency ?" in *Annual Allerton Conference on Communication, Control, and Computing*, 2013.

[35] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing Latency via Redundant Requests: Exact Analysis," in *ACM SIGMETRICS*, 2015.

[36] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[37] S. Rizvi, X. Li, B. Wong, F. Kazhamiaka, and B. Cassell, "Mayflower: Improving Distributed Filesystem Performance Through SDN/Filesystem Co-Design," in *IEEE ICDCS*, 2016.

[38] W. Zhang, T. Wood, and J. Hwang, "NetKV: Scalable, Self-Managing, Load Balancing as a Network Function," in *IEEE ICAC*, 2016.

[39] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *ACM SOSP*, 2017.