NetRS: Cutting Response Latency of Distributed Key-Value Store with In-Network Replica Selection

Yi Su, Dan Feng, Yu Hua, Zhan Shi, Tingwei Zhu
Huazhong University of Science and Technology
Email: {suyi, dfeng, csyhua, zshi, twzhu}@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. Due to the dynamically changing performance of servers, the replica selection for a request will directly affect the response latency. However, in the context of key-value stores, even the state-of-the-art algorithm of replica selection still has a large room for improvement on cutting 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 takes advantage of new-generation network devices, including programmable switches and network accelerators, to select replica for requests. NetRS is able to support diverse algorithms of replica selection and adapt to the network topology of modern datacenters. Compared with the conventional scheme of clients performing replica selection for requests, NetRS could effectively cut the response latency according to our extensive evaluations. Specifically, NetRS reduces the average latency by up to 59.9%, and the 99th latency by up to 87.1%.**

I. Introduction

The distributed key-value store is a vital component of modern Web applications. For such applications, minimizing the response latency is critical due to their interactive nature. Even the poor tail latency of 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 [1]. Hence, further cutting response latency of key-value stores is of great significance.

Distributed key-value stores (e.g. Cassandra [2], Dynamo [3], Voldemort [4], Couchbase [5], etc.) generally replicate data over multiple servers to be highly available and reliable. As server performance fluctuations are the norm [6], [7] (especially in cloud environments where multiple tenants share resources), the replica selected for serving a reading request will have a direct effect on the response latency of the request. Considering that the workloads of key-value stores are commonly read dominant [8], the replica selection scheme plays an important role in cutting response latency of the system. Although redundant requests [6] (a client issues the same request to multiple replica servers, and uses the response that arrives first) can help eliminate the effects of the performance variability of servers, redundant requests cannot reduce the latency in all situations [7]. The use of redundancy is actually a trade-off between system utilization and response latency [9].

In the context of key-value stores, there are several replica selection algorithms [2], [7] from both academia and industry that address the dynamically changing performance of servers. As requests in key-value stores typically access small size data (about 1KB) [1], 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 replica 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 small size status (a few bytes).

While using clients 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 the 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. Considered that one client typically sees a small portion of the traffic, there will be lots of clients selecting replica 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 possibility of "herd behavior" is positively correlated to the number of independent RSNodes. Considered 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 [7], still has a large room for improvement in terms of cutting response latency. It is worth mentioning that the above analysis does not apply to storage systems that are not latency-critical (e.g. HDFS [10]). In such systems, 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 datacenters. Instead of using clients as

RSNodes, NetRS offloads tasks of replica selection to programmable network devices, including programmable switches (e.g. Barefoot Tofino [11], Intel's FlexPipe [12]) and network accelerators (e.g. Cavium's OCTEON [13], Netronome's NFE-3240 [14]). In addition to the control plane programmability with Software Defined Networking (SDN) switches [15], programmable network devices further enable programmability of the data plane. 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 multicore processor. Since network devices (e.g. switches) are much fewer than end-hosts in the datacenter, network devices naturally gather traffics. Due to this fact, NetRS has two advantages over the client-based replica selection. First, compared with some clients, a network device could obtain the more recent local information by gathering traffics. Then, as an RSNode, the network device is more likely to choose a better replica for a request. Second, NetRS could reduce the possibility of "herd behavior" with fewer RSNodes. It is because one network device could perform replica selection for requests from multiple clients.

Nevertheless, offloading replica selection to datacenter network is nontrivial due to the following reasons. (i) **Multiple Paths**. Modern datacenters 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 that are determined by both the request and its corresponding response, 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, consider 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 datacenter, the 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 the comprehensive considerations, including the requirements of the replica selection algorithm, the capacity of the network devices, and the network overheads due to taking extra hops.

In summary, our contributions include:

(i) **The architecture of NetRS**. In the context of datacenter networks that support multipath communications, we build 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 a flexible format of NetRS packet and customized processing pipelines of each network device. Moreover, NetRS provides support for diverse replica selection algorithms.

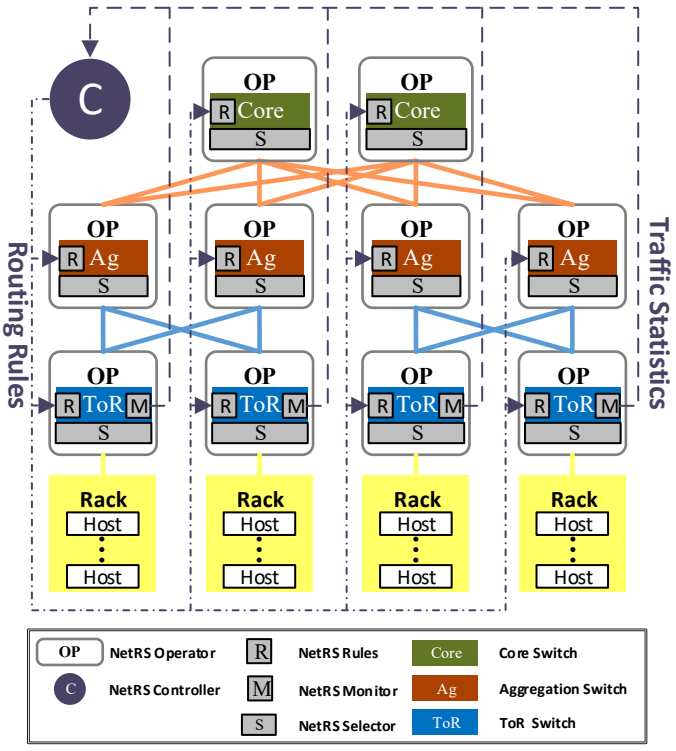

Fig. 1. An overview of the NetRS architecture.

(ii) **Algorithms of RSNodes placement**. We propose an algorithm to arrange RSNodes in the modern datacenter 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 on network devices by solving the ILP. In addition, to deal with failures of network devices, we have also developed a heuristic algorithm that is able to quickly find the alternative placement.

(iii) **System evaluation**. We evaluate NetRS using simulations in a variety of scenarios, including different numbers of clients, different demand skewness of clients, different system utilization and different service time of servers. Compared with performing replica selection by clients, NetRS reduces the mean latency by up to 59.9%, and the 99th latency by up to 87.1%.

II. Overview of NetRS

This section provides an overview of the NetRS architecture. We describe the design of NetRS and show how NetRS adapts to the network of modern datacenter and exploits programmable network devices.

The datacenter network generally uses a hierarchical topology [16]–[18] 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, and a pod also contains 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 datacenter networks, there is also a centralized SDN controller. The controller connects to all switches via low-speed links.

Programmable switches and network accelerators address different demands of in-network replica selection. On one hand, a programmable switch provides both the fast packets forwarding and the customizable pipeline of packets processing at the 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. read from the memory, write to the 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 the lower routing performance and fewer ports.

We have following considerations for designing the NetRS framework. (i) NetRS should keep things in network as much as possible. Clients and servers of the 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 the 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 datacenter 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 the network accelerator, the programmable switch 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 due to consuming too much hardware resources.

Fig. 1 shows the architecture of the NetRS framework. 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 accelerators attached to the switch. The software part generally includes NetRS rules and the NetRS selector. If the switch of a NetRS operator is a ToR switch, the operator would have the NetRS monitor. NetRS rules and monitor reside on the 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 the 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 the key-value store or (ii) use the packet to update local information if the packet is a response of the 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 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 increase in latency. The time for the system to stabilize again depends on many factors, including the speed of convergence of the replication selection algorithm, the number of new RSNodes, and the service time of the server. Thanks to the stable workload of user-facing applications [19], the NetRS controller does not need to update the RSP frequently.

III. NetRS Controller

In this section, we present the NetRS controller, which determines the NetRS operator that works as the 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 the exception handling mechanism (Section III-C). At last, we show the approach to dealing with failures of NetRS operators (Section III-D).

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, (i) finer-grained traffic groups require more efforts to find the optimal RSP due to larger solution space, and (ii) introduce more overheads to carry out the RSP in datacenter network because network devices need to deal with more cases. As a matter of 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 the 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. It is worth mentioning that the RSP and traffic groups are agnostic to clients and servers of the key-value store (Section IV).

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 use the following optimization goals to address the two factors (detailed in Section I) that have a significant impact on the effectiveness of replica selection algorithms, like C3 [7].

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

The NetRS controller addresses *Goal 1* and *Goal 2* by minimizing the number of RSNodes. Firstly, the average traffics that flow 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. Secondly, as the probability 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 one request.
- *Constraint 2*: The utilization of each network accelerator should be limited.
- *Constraint 3*: The total amount of extra hops that requests take to go to RSNodes should be limited.

The *Constraint 1* exists because replica selection algorithms typically rely on metrics that are 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, multiple NetRS operators selecting replica for one request 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 the *Constraint 2* to make the load of each network accelerator be fit to its capacity. High utilization of a network accelerator will make requests wait a long time for replica selection. The *Constraint 3* enables the trade-off between the flexibility of making the RSP and the network overheads of taking extra hops. If the RSNode for requests of a traffic group lies 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 datacenter network.

B. Replica Selection Plan

We formalize the optimization problem of RSNodes placement as an Integer Linear Programming (ILP) problem. The NetRS controller can determine the RSP by solving the ILP problem with an optimizer (e.g. Gurobi [20], CPLEX [21]).

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 the RSNode for requests of any traffic group, then D_j will be set to 1, otherwise 0.

Suppose R is a 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. Considered the multi-tier topology of the datacenter network described in Section II, 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 traffics. We can get R according to the network topology, and use traffic statistics that collected by NetRS monitors (Section IV-D) to determine T.

Suppose a NetRS operator o_j could perform replica selection without introducing significant delay only if the utilization of its network accelerator is under U_j. Then the maximum traffic T_j^{max}, which choose 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 traffics, 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.

The description of the ILP problem is as follows.

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

$$Subjects\ to :$$

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

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

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

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

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

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

where $t(x)$ 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 connects), and $h(i,j) = t(i) - t(j)$.

Among constraints of the ILP problem, Equations (2) suggest 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, Equations (5) , (6) and (7) correspond to *Constraint 1*, *Constraint 2* and *Constraint 3*, respectively. We could cap the solving time of the ILP prblem to make trade-offs between the recalculation expense and the optimality of the RSP. Besides the 3-tier topology as shown in Fig. 1, our algorithm is also applicable to n-tier ($n \in \{1, 2, ...\}$) tree-based topologies of datacenter network.

In order to accommodate different RSPs, every switch must have a network accelerator. In fact, network accelerators are widely adopted in the datacenter network for various purposes (e.g. deep packet inspection [13], [14], firewalls [13], [14], in-network cache [16], packets sequencing [22], [23]). 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 the *Constraint 1* suggests that there should be only one RSNode among all NetRS operators in a network path. Then the 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 the Degraded Replica Selection (DRS) mechanism to handle exceptions. The DRS requires that clients of the key-value store 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 could enable 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 because the ILP problem in Section III-B is unsolvable. Given the constraints of the ILP problem, this happens depending on the distribution of loads among clients. In this case, the traffic of some traffic groups is too heavy for the NetRS operators. Hence, we have to enable the DRS for some of the traffic groups so that a feasible RSP exists for the rests. The number of traffic groups using the DRS should be as less as possible. It is because enabling the DRS for a traffic group will introduce additional RSNodes. The NetRS controller turns DRS on for groups with the highest traffic first to minimize the number of traffic groups using the DRS. (ii) A NetRS operator does not work as expected, e.g the NetRS operator is overload due to the changing of loads. The NetRS controller will enable the DRS for traffic groups that use the NetRS operator as the RSNode. (iii) The NetRS operator, which works as an RSNode, fails (Section III-D).

D. Fault Tolerance

The NetRS framework is able to handle failures of NetRS operators. We do not consider the failure of the NetRS controller because (i) it is not on the critical path of accessing the key-value store, and (ii) it could be highly available with standbys.

Algorithm 1: Rearrange traffic groups whose RSNodes have failed.

Input: G (traffic groups whose RSNodes have failed), O_g^k (NetRS operators that can be the RSNode for a traffic group g in tier k), k_0 (the tier of the failed RSNode), k_w (the bottom tier), k_{-u} (the top tier).

```
Function Rearrange(G, O_g^k, k_0, k_w, k_-u)
    for g in G do
        Enable degraded replica selection for g
    O ← EmptyList
    for i in [0] + [-1, -2, ..., -u] + [1, 2, ..., w] do
        for o in Sorted(O_g^{k_i}) do
            O.append(o)
    for g in G do
        for o in O do
            if no constraints violation for g → o then
                Set o as the RSNode for g
                break
```

In our design, the NetRS controller monitors the availability of NetRS operators. On one hand, if the failed NetRS operator is not an RSNode, the NetRS controller does nothing because the failure will not affect the replica selection of NetRS. However, the NetRS controller will prevent an unavailable NetRS operator from being an RSNode while it makes the

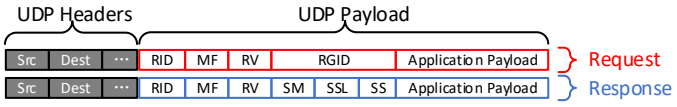

Fig. 2. Packet format of NetRS for request and response.

new RSP. As there are lots of mature mechanisms [17] that make the datacenter network be highly available, we assume that the failures do not affect packets routing. On the other hand, if the failed NetRS operator works as an RSNode for any traffic group, the controller will use the Algorithm 1 to rearrange traffic groups whose RSNodes have failed. In order to minimize impacts on the system availability, the NetRS controller will first enable degraded replica selection (Section III-C) for traffic groups whose RSNodes have failed. Then, for each traffic group g whose RSNode has failed, the controller will search for an alternative NetRS operator to work as an RSNode for g. The NetRS controller will first search in the same tier of the failed NetRS operator, then in upper tiers, finally, in lower tiers. The core idea of Algorithm 1 is to avoid introducing more NetRS operators for handling the traffic groups whose RSNodes have failed. Considered that network devices are highly reliable [24] and updating NetRS rules only takes about a few milliseconds [22], failures of NetRS operators should not significantly affect system availability.

IV. NetRS Operator

This section describes the design of the NetRS operator. We first introduce the packet format of NetRS (Section IV-A). Then, we show how a programmable switch processes packets according to NetRS rules (Section IV-B), and the working procedure of a NetRS selector that runs on the network accelerator (Section IV-C). Finally, we present the NetRS monitor's approach to collecting traffic statistics (Section IV-D).

A. Packet Format

The packet format plays an important role in propagating information. Hence clients, servers, switches and network accelerators should agree to the common format. As stateful network protocols (e.g. TCP) introduce latency overheads, recent key-value stores [16], [25] generally use stateless network protocols (e.g. UDP). In order to cut the client-perceived latency, key-value stores in production environments also exploit UDP-based network protocols for reading requests [1]. Considered that the goal of NetRS is to reduce the read latency of 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 the 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 the protocol overheads low.

NetRS packets are carried in the UDP payload. We use different packet formats for the request and the response because the information required for a request and a response is different. Using separate packet formats could reduce the bandwidth overheads of the NetRS protocol. Fig. 2 shows the packet format of the request and the response, respectively.

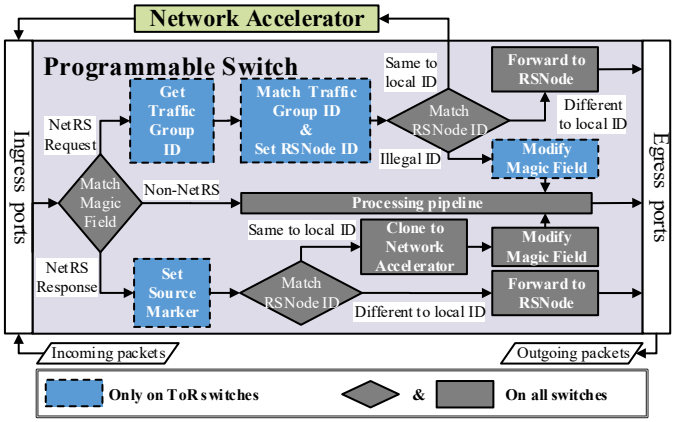

Fig. 3. NetRS rules within ingress pipelines of a programmable switch.

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 same to the value in its corresponding request. An RSNode could exploit this segment to collect request-level data. For example, the RSNode may set the retaining value of a request with the timestamp of sending the request, 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.

Segments only in the request packet are as follows:

- **RGID** (Replica Group ID): [*3 bytes*] The ID of a replica group. A NetRS operator 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 place data with consistent hash. The advantage of using RGID is to keep the headers of a packet be fixed-sized and irrelevant to the number of replicas. The fixed-sized header is more friendly for packet parsing of switches.

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.

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 both 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 first match the packet's RSNode ID segment, if the RSNode ID is different to the local ID of the switch, then the switch will forward the packet to the next hop towards the RSNode. Otherwise, if the RSNode ID is same to the local ID, the switch will perform corresponding operations based on the packet type. The packet will be forwarded to the network accelerator that runs the NetRS selector if it is a NetRS request. 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 first 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, which also makes it a non-NetRS packet. However, the packet should be recognized by NetRS monitors (Section IV-D). By cloning the packet of NetRS response, we could avoid the latency overhead of packet processing by the network accelerator.

As the RSP and traffic groups are agnostic to end-hosts, a client of the key-value store is 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 other types of switches, a ToR switch has extra NetRS rules for the NetRS request, which could (i) match the source IP of the 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 modify its magic field to make it a non-NetRS packet that is recognizable by NetRS monitors.

C. NetRS Selector

The NetRS selector is responsible for performing replica selection and maintaining corresponding local information. As the NetRS selector resides on the network accelerator attached to the programmable switch, we could implement an arbitrary replica selection algorithm in the NetRS selector 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 $f(M_{resp}), f(M_{resp}) \neq M_{req}$, where M_{resp} and M_{req} are constant values that label the NetRS response and request, respectively, and $f(\cdot)$ is an invertible function. 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 the server marks a response packet as a NetRS response, only if the packet's corresponding request had flowed through a NetRS selector. 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. Considered 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 could only be of traffic groups associated with the rack.

The NetRS monitor filters packets based on the magic field. NetRS rules ensure that the NetRS monitor could recognize responses of the 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 source marker. 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 extent of NetRS cutting response latency, including (i) the number of clients, (ii) the demand skewness of clients, (iii) the system utilization, and (iv) the service time of servers.

A. Simulation Setup

In our experiments, we use the simulator from C3 [7], 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 contains 1024 end-hosts connected via a 16-ary fat-tree (3-tier) [18].

We set major parameters in our evaluation based on the experimental parameters in C3 [7]. 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 [26] 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 [27]. For a request, the workload generator chooses an accessing key out of 10 million keys according to the Zipfian distribution (the Zipf parameter is 0.99). For each experiment, the key-value store receives 600,000 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 [16]. Specifically, the RTT (Round-Trip Time) between a switch and its attached network accelerator is 2.5us. The processing time of a network accelerator is 5us. We consider using low-end network accelerators (1 core per accelerator). The network latency is 30us between two switches that are directly connected.

We perform the simulation with the above parameters in all cases, unless otherwise noted. The clients and servers are randomly deployed across end-hosts [28], and each host only has one role [29]. 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 [7], which is state-of-the-art).

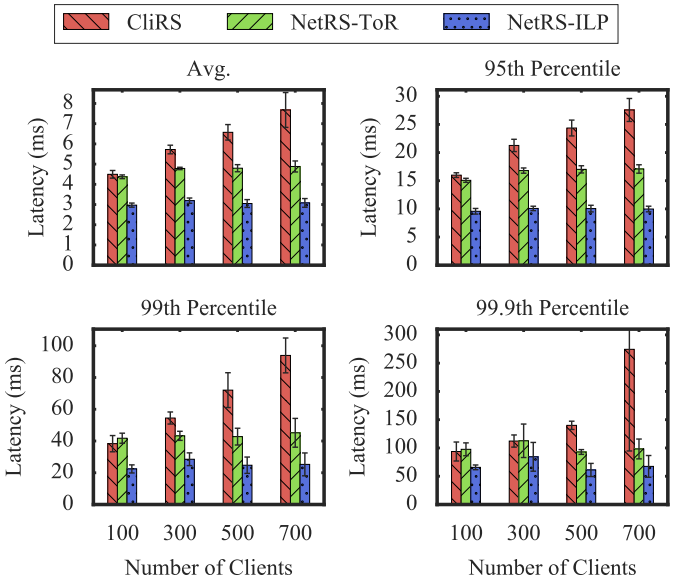

Fig. 4. The comparison of performance with varying number of clients.

- **CliRS**: A commonly used replica selection scheme in key-value stores [2]–[5]. With CliRS, clients work as RSNodes and perform replica selection for requests.
- **NetRS-ToR**: Using the NetRS framework for replica selection with a straightforward RSP, which simply specifies the NetRS operator on the Top of a Rack 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 of NetRS-ILP consists of 2 RSNodes on aggregation switches and 6 RSNodes on core switches.

B. Results and Analysis

This section provides experimental results in a variety of scenarios. We also present the corresponding analysis by quantitatively comparing the mean and tail response latency of each replica selection scheme. We use the open-loop workload, which is in line with the real-world workloads of Web applications [30]. The aggregate arriving rate of requests (A) is determined by the expected system utilization ($\frac{t^{kv}A}{N_s N_p}$), which is 90% in all experiments unless otherwise noted. 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.

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 offers the best performance. Compared with CliRS, NetRS-ILP reduces the mean latency by 33.9%-59.9% and reduces the 99th latency by 41.5%-73.1%. Compared with

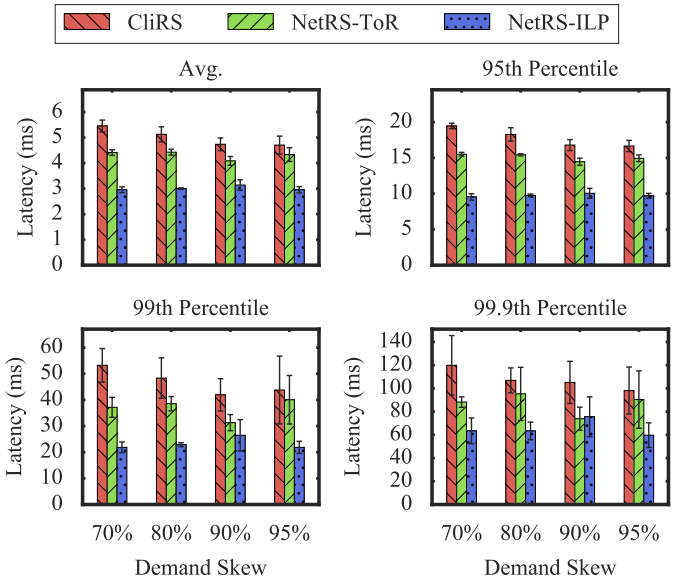

Fig. 5. The comparison of performance with varying demand skewness.

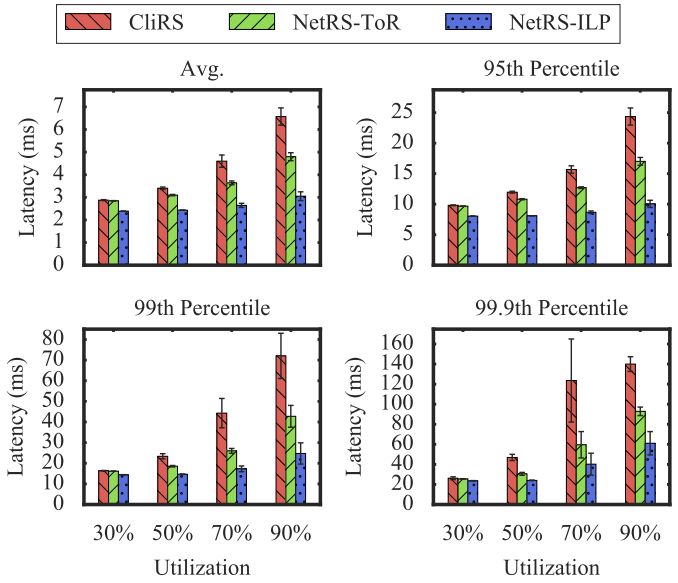

Fig. 6. The comparison of performance with varying system utilization.

NetRS-ToR, NetRS-ILP reduces the mean and 99th latency by 34.7% and 41.6% on average, respectively. (ii) With CliRS, both the mean and tail latency increase as the number of clients grows. However, the response latency roughly remains unchanged with NetRS-ILP and NetRS-ToR regardless of the number of clients. The reason behind is that, with NetRS-ILP and NetRS-ToR, the number of RSNodes is irrelevant to the number of clients. Since each client works an RSNode with CliRS, these experiments also validate our analysis that more independent RSNodes could lead to worse replica selection, which hurt the performance.

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. Although CliRS and NetRS-ILP still provide the worst and best performance, respectively, the reduction in response latency due to the NetRS framework tends to decrease as the demand skewness increases. For example, in the scenario of no demand skewness shown in Fig. 4 (500 clients), NetRS-ILP reduces the mean and 99th latency by 53.6% and 65.7%, respectively. However, the reduction in the mean and 99th latency are by 45.8% and 58.9% when the demand skewness is 70%. For the demand skewness of 90%, NetRS-ILP introduces only 33.7% reduction in the mean latency and 36.9% reduction in the 99th latency. There are the following reasons for this phenomenon. 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 will be reduced to the number of high-demand clients thanks to the skewed demand. On the other hand, the demand skewness will not benefit the NetRS framework because high-demand clients spread over different network locations. The demand skewness of clients only has a limited impact on the traffic skewness of switches.

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 ($t^{kv}A$) ranging from 30% to 90%. Compared with CliRS, NetRS-ILP reduces the mean latency by 17.0%-53.6% and reduces the 99th latency by 12.0%-65.7%. Compared with NetRS-ToR, NetRS-ILP reduces the mean and 99th latency by 16.1%-36.4% and 10.8%-42.1%, 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 the scheme that offers worse performance, a scheme introduces more reduction in response latency in the region of higher utilization. The reason behind is that the severe contention of resources will amplify the impact of bad replica selection on the response latency.

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, we find that NetRS-ILP generally introduces less reduction in mean latency with a lower service time of servers. For example, NetRS-ILP reduces the mean latency by 23.8% with the 0.1ms service time, and by 53.6% with the 4ms service time. However, there is no such correlation in the tail latency or with the NetRS-ToR scheme. The reason behind is that, with NetRS-ILP, a packet may take extra hops to reach its RSNode. Extra hops introduce latency overheads (30us per extra hop). The impact of these overheads on the mean latency could be more significant in the context of lower service time. It is because the mean latency is generally

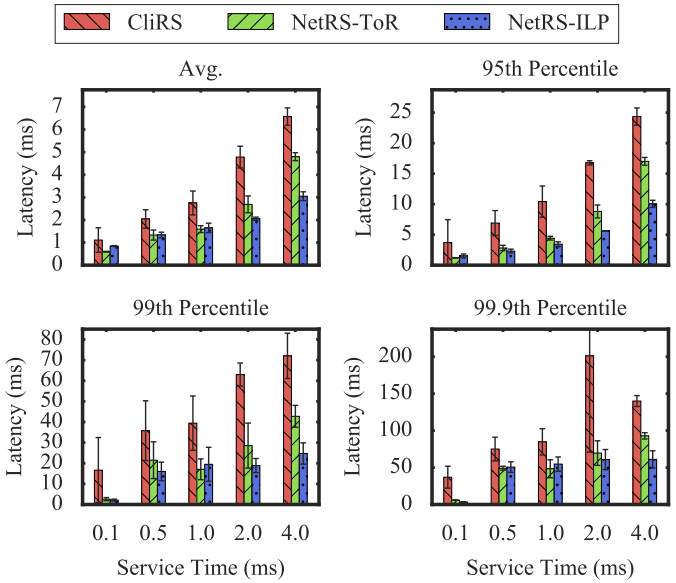

Fig. 7. The comparison of performance with varying service time of servers.

comparable with the service time of servers. However, the tail latency is typically orders of magnitude greater than the mean latency, the latency overheads due to taking extra hops are negligible in terms of the tail latency.

In summary, according to the experimental results shown in Fig. 4 , 5 , 6 and 7, the NetRS framework could effectively cut the response latency of key-value stores compared with selecting replica by clients. In addition, these results also show that our ILP-based algorithm of RSNodes placement makes a significant contribution to the latency reduction of NetRS.

VI. Related Work

A. Load Balancing

Load balancing has been studied extensively. One of the most important problems in load balancing is to choose the proper server(s) to fulfill a request (or task) in the context of the time-varying performance of servers. In general, the approaches to addressing the problem fall into two categories: redundant requests and replica selection. On one hand, redundant requests are used pervasively to cut response latency. Google presents their "tail-tolerant" techniques [6] in large-scale systems, which reissue requests to cut latency with cross-server cancellations to reduce overheads. Vulimiri et al. [9] suggest that the use of redundant requests is a trade-off between response latency and system utilization. Shah et al. [31] and Gardner et al. [32] provide theoretical analyses on using redundant requests to cut latency. On the other hand, replica selection is also an indispensable part of distributed systems. Mitzenmacher [33] 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 is the default load balance strategy of Cassandra, which selects replica based on the history of reading latencies and I/O loads. C3 [7] 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 various replica selection algorithms via performing replica selection in the datacenter network.

Mayflower [34] selects the replica server and the network path collaboratively using the SDN technique for distributed filesystems, e.g. HDFS [10]. Mayflower deals with 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 scenario of key-value stores. In this scenario, replica selection should be lightweight and performed in a distributed manner because the average request size is about 1KB.

B. In-Network Computing

As network devices evolve to be more and more powerful, a lot of recent works try to improve the performance of datacenter applications via offloading tasks to the network. SMIND [35] proposes the in-network deduplication to reduce the network transmission. Camdoop [36] and NetAgg [37] propose to perform aggregation operations in network for applications that follow a partition/aggregation pattern, e.g. data analysis, search engines. The in-network aggregation helps eliminate the network congestion, which occurs due to multiple servers sending data to one aggregation server. NOPaxos [22] avoids overheads of application-level consensus protocols, e.g. Paxos, via serializing all requests with a sequencer in the network. Eris [23] extends NOPaxos to transactions with a more sophisticated sequencing mechanism.

In-network computing is also helpful to enhance key-value stores. NetKV [38] introduces a network middlebox that enables adaptive replication. SwitchKV [25] balances loads of backend 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 [16] builds an in-network cache system that works as a cache layer for key-value stores. Different from NetRS that 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 datacenters. NetRS exploits programmable switches and network accelerators to aggregate tasks of replica selection. Compared with selecting replica by clients, NetRS introduces a significant reduction in response latency. NetRS is able to support various replica selection algorithms with the flexible format of NetRS packet and the processing pipelines at each network device. We design the architecture of NetRS that adapts to the network topology in modern datacenters and formalize the problem of RSNodes placement with ILP. Moreover, NetRS could be highly available through a mechanism that handles exceptions, e.g. failures of network devices.

REFERENCES

[1] 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 *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 385–398.

[2] "Apache Cassandra," http://cassandra.apache.org/, Dec. 2017.

[3] 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 *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[4] "Voldemort," http://www.project-voldemort.com/voldemort/, Dec. 2017.

[5] "Couchbase," https://www.couchbase.com/products/server, Dec. 2017.

[6] J. Dean and L. A. Barroso, "The Tail at Scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[7] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, May 2015, pp. 513–527.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 53–64.

[9] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low Latency via Redundancy," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 283–294.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1–10.

[11] "Barefoot Tofino," https://barefootnetworks.com/, Dec. 2017.

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

[13] "OCTEON MIPS64," http://www.cavium.com/OCTEON_MIPS64.html, Dec. 2017.

[14] "Netronome nfe-3240," https://www.netronome.com/products/nfe/, Dec. 2017.

[15] 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, Jan. 2015.

[16] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 795–809.

[17] 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," *Commun. ACM*, vol. 59, no. 9, pp. 88–97, Aug. 2016.

[18] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74.

[19] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, Nov. 2016, pp. 755–770.

[20] "Gurobi optimization," http://www.gurobi.com/, Dec. 2017.

[21] "Cplex optimizer," https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer, Dec. 2017.

[22] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 467–483.

[23] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 104–120.

[24] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 350–361.

[25] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be Fast, Cheap and in Control with SwitchKV," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 31–44.

[26] 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, Sep. 2010.

[27] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power Management of Online Data-intensive Services," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 319–330.

[28] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280.

[29] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 123–137.

[30] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[31] N. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency ?" in *2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Oct. 2013, pp. 731–738.

[32] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing Latency via Redundant Requests: Exact Analysis," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '15. New York, NY, USA: ACM, 2015, pp. 347–360.

[33] 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, Oct. 2001.

[34] S. Rizvi, X. Li, B. Wong, F. Kazhamiaka, and B. Cassell, "Mayflower: Improving Distributed Filesystem Performance Through SDN/Filesystem Co-Design," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 384–394.

[35] Y. Hua, X. Liu, and D. Feng, "Smart In-network Deduplication for Storage-aware SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 509–510.

[36] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting In-network Aggregation for Big Data Applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3.

[37] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 249–262.

[38] W. Zhang, T. Wood, and J. Hwang, "NetKV: Scalable, Self-Managing, Load Balancing as a Network Function," in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, Jul. 2016, pp. 5–14.

[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 *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. ACM, 2017, pp. 121–136.