An In-Network Replica Selection Framework for Latency-Critical Distributed Data Stores

Yi Su, Dan Feng, *Member, IEEE*, Yu Hua, *Senior Member, IEEE*, Zhan Shi, and Tingwei Zhu

Abstract—In distributed data stores, performance fluctuations generally occur across servers, especially when the servers are deployed in a cloud environment. Hence, the replica selected for a reading request will directly affect the response latency. However, replica selection is challenging in latency-critical data stores (e.g., key-value stores). Such data stores generally deal with small size data, and clients have to select replicas independently. Even the state-of-the-art algorithm of replica selection (C3) still has considerable room for improving the latency. According to our experiments, compared with C3, using the ideal replica selection (Oracle) reduces the 99th latency by about 34-60 percent. In this article, we propose NetRS to address the fundamental factors that prevent replica selection algorithms from being effective. NetRS is a framework that enables in-network replica selection for distributed data stores. It 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. According to our extensive evaluations, compared with the conventional scheme of clients selecting replicas for requests, NetRS reduces the mean latency by up to 50.3 percent, and the 99th latency by up to 56.3 percent. Moreover, NetRS could effectively cut the response latency even when unexpected events (e.g., workload changes, network device failures), and network congestion occur.

Index Terms—Replica selection, in-network computing, response latency, distributed storage

1 Introduction

The DISTRIBUTED data store is a vital component of modern Web applications [1], [2], [3]. For such applications, minimizing the response latency is critical due to their interactive nature. Even the poor tail latency in the data 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].

In distributed data stores, e.g., Cassandra [5] and Dynamo [6], the replica selection scheme plays an important role in cutting response latency. The workloads of such data stores are commonly read dominant [7]. For a reading request, replica selection has a direct impact on the response latency under fluctuations of server performance, which are the norm [8], [9], [10], [11].

We propose NetRS [12] to improve the replica selection in large-scale data stores. NetRS is a framework that enables the in-network replica selection in data centers. NetRS offloads tasks of replica selection to programmable network devices, including programmable switches (e.g., Barefoot Tofino [13]) and network accelerators (e.g., Cavium's OCTEON [14], Netronome's NFE-3240 [15]). In addition to the control plane programmability with Software Defined Networking (SDN) switches [16], 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 multicore processor.

Selecting replicas properly is challenging for large-scale systems, in which, the concurrency of one server is much smaller than the number of clients, and most of these end-hosts are connected via multiple switches. Considering that latency-critical data stores (e.g., key-value stores) typically deal with small size data [4] (e.g., 1KB), replica selection algorithms [5], [9] have to 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. The client-based RSNodes may reduce the effectiveness of replica selection algorithms due to the following two factors:

(i) A client is likely to select a poorly-performing server for a request due to using stale server status. As clients rely on requests and responses to update local information, the traffic flowing through a client determines the recency of its local information. Considering that one client typically sees a small portion

- *The authors are with the 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 430074, China. E-mail: {suyi, dfeng, csyhua, zshi, twzh}@hust.edu.cn.*

Manuscript received 18 Oct. 2018; revised 18 Dec. 2019; accepted 17 Feb. 2020. Date of publication 24 Feb. 2020; date of current version 7 June 2022.
(Corresponding author: Dan Feng.)
Recommended for acceptance by E. Elmroth.
Digital Object Identifier no. 10.1109/TCC.2020.2976008

of the traffic, lots of clients will selects 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) [9]. The occurrence of “herd behavior” is positively correlated to the number of independent RSNodes. Due to the large number of clients, servers are highly likely to suffer from load oscillations.

NetRS can effectively address these two factors. In data centers, as network devices (e.g., switches) are much fewer than end-hosts, network devices can automatically gather traffic. Hence, NetRS has two advantages over client-based replica selection. First, compared with clients, a 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 as one network device could select replicas for requests from multiple clients.

Offloading replica selection to network is non-trivial, there are following challenges:

(i) *Determine the RSNodes Placement*. In data centers, a request from a client needs to flow through multiple switches until arriving at the server. In fact, with 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 to optimize the overall performance and meet constraints of the system. Given the high complexity of this placement problem, we also need to quickly solve it to handle unexpected events, like workload changes, switch failures, etc.

(ii) *Keep Things in Network*. NetRS should keep things in network as much as possible. On one hand, it is much easier for clients and servers of data stores to take advantage of NetRS without knowing NetRS internals (e.g., RSNodes placement). On the other hand, NetRS could stay simple and avoid management overheads, like coordinating a large number of clients and servers when the RSNodes placement changes, network devices fail, etc.

(iii) *Accommodate Diverse Algorithms*. First, NetRS should integrate programmable network devices to meet both computational and storage requirements of diverse replica selection algorithms. Second, NetRS should provide flexible interfaces for RSNodes collecting necessary inputs of diverse algorithms. Moreover, algorithms may rank replicas according to metrics determined by both the requests and their responses, e.g., the number of pending requests. However, a request and its response may flow through different network paths (different sets of switches) in modern data centers due to redundant switches. Therefore, NetRS should be able to guarantee that one request and its corresponding response flow through the same RSNode.

In summary, our contributions include:

(i) *Architecture of NetRS*. We design the NetRS framework that enables in-network replica selection for distributed data 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) *Formalization of RSNodes Placement*. We use Integer Linear Programming (ILP) to formalize the RSNodes placement problem in the modern data center network with a complex topology. Our formalization comprehensively considers the requirements of replica selection algorithms, the NetRS’s utilization of each network device, and the network overheads caused by taking extra hops to RSNodes.

(iii) *Algorithms of RSNodes Placement*. We propose fast and efficient algorithms to find an approximately optimal solution for the RSNodes placement problem, which is an NP-complete problem. With these algorithms, NetRS could adaptively change the RSNodes placement in case of workload changes, network device failures, etc.

(iv) *System Evaluation*. We evaluate NetRS using simulations in a variety of scenarios. Under open-loop workloads, NetRS reduces the mean latency by up to 50.3 percent and the 99th latency by up to 56.3 percent compared with client-based replica selection. Under closed-loop workloads, NetRS improves the throughput by up to 56.2 percent compared with client-based replica selection. Moreover, NetRS can reduce latency regardless of workload changes or RSNodes failures.

2 Overview of NetRS

In this section, we describe the design of NetRS, how NetRS exploits programmable network devices, and how NetRS works in the network of modern data centers.

NetRS selects replicas using both programmable switches and network accelerators. As the related hardware is new and evolving, NetRS only relies on the basic and standard abilities of these kinds of hardware. The feasibility of exploiting these abilities are verified many times, lots of systems relying on the same abilities have been deployed successfully (e.g., deep packet inspection [14], [15], in-network cache [17], packets sequencing [18], [19], etc.).

The data center network generally uses a hierarchical topology [17], [20], [21] 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.

The programmable switch provides both the fast packets forwarding and customizable pipelines of packet processing

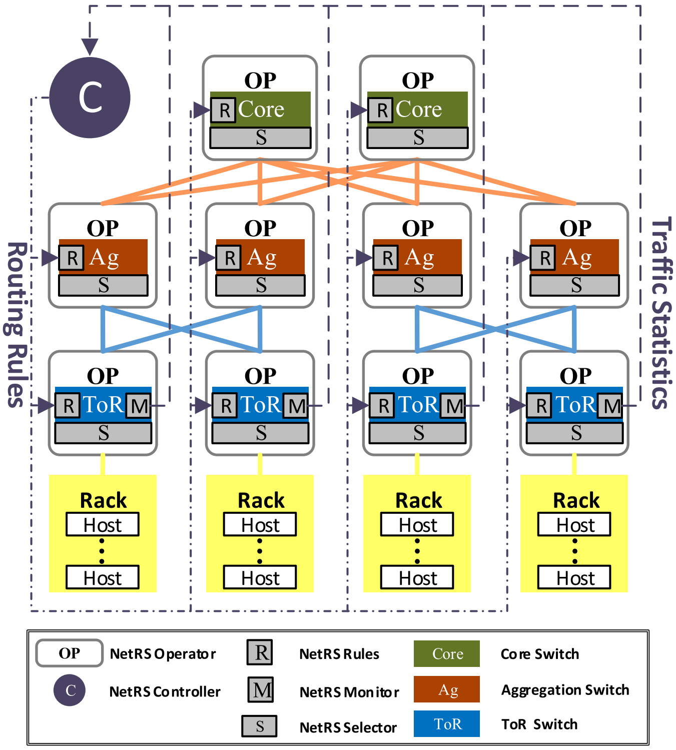


Fig. 1. An overview of the NetRS architecture.

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. The network accelerator is able to perform general computations with a multicore (or manycore) processor and several gigabytes of memory. Compared with general-purpose CPUs, network accelerators are optimized for packets processing: (i) the network accelerator supports fast Direct Memory Access (DMA) between ingress/egress ports and internal memory, (ii) it supports fast packets distribution among processing cores, (iii) it supports hardware accelerations for typical operations in packets processing, e.g., pattern matching. A network accelerator can easily saturate the bandwidth of an Ethernet port (40 Gbps-100 Gbps), and the Round-Trip Time (RTT) between a switch and its accelerator is low (about 2.5us for a 1KB packet) [17].

NetRS is a hardware/software co-design framework that enables in-network replica selection for distributed data stores. Fig. 1 shows the NetRS architecture. The basic unit of NetRS is the NetRS operator, which is a collection of hardware and software. The hardware part is similar to the hardware model proposed in IncBricks [17], which consists of a programmable switch and a network accelerator attached to the switch. A switch could belong to multiple NetRS operators if there are multiple accelerators attached to the switch. Switches are responsible for filtering NetRS related packets and forwarding them to appropriate accelerators if necessary. Accelerators are responsible for performing application-layer computations (e.g., replicas ranking). NetRS has to use such a hardware model because the programmable switch only supports simple operations, e.g., reading from memory, writing to memory, to keep the high speed of packet forwarding. 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 5.2). 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 data store or (ii) use the packet to update local information if the packet is a response of data store (detailed in Section 5.3). The NetRS monitor collects traffic statistics (detailed in Section 5.4).

NetRS aims to cut the response latency in data stores by making full use of accelerators in the data center network. In data centers, multiple applications share the links, switches and network accelerators. NetRS should minimize their impacts on other applications to efficiently share network devices with them. Therefore, NetRS should be able to limit its bandwidth overheads and its utilization of accelerators. According to these constraints, the network topology and traffic statistics collected by NetRS monitors, a NetRS controller finds a placement of RSNodes that maximizes the overall system performance (detailed in Section 3). In NetRS, we call the placement of RSNodes as the Replica Selection Plan (RSP). Given an RSP, the NetRS controller first turns the RSP into NetRS rules and updates the NetRS rules on corresponding switches (detailed in Section 4). Then switches will route packets according to the placement of RSNodes determined by the NetRS controller.

3 RSNodes Placement Problem

This section states the problem of RSNodes placement in modern data centers. We provide a detailed description of the problem, then formalize the problem using Integer Linear Programming (ILP). At last, we propose a heuristic algorithm that finds approximately optimal solutions.

3.1 Problem Description

We consider the problem by first dividing requests into different traffic groups. Then 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 an 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 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 placement problem of RSNodes is an optimization problem of assigning each traffic group's RSNode to a NetRS operator. We have the following optimization goals to cope with the two factors (detailed in Section 1) that hurt the effectiveness of replica selection algorithms.

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

There are also constraints as follows:

- *Constraint 1*: There should be only one RSNode for each request.
- *Constraint 2*: NetRS's 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 a 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 prevent NetRS from overusing each network accelerator. On one hand, high utilization of a network accelerator will make requests wait a long time for replica selection. On the other hand, considering that multiple applications share network accelerators, NetRS should not use a network accelerator excessively. *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.

3.2 Problem Formalization

We formalize the problem of RSNodes placement as an Integer Linear Programming (ILP) problem. In order to achieve *Goal 1* and *Goal 2*, we minimize the number of merged RSNodes in the ILP problem. A merged RSNode is a union of traffic group's RSNodes located in the same NetRS operator. Each merged RSNode independently selects replicas and RSNodes in one merged RSNode share local information. There are two advantages to using fewer merged RSNodes. First, the average traffic flowing through one merged RSNode will increase. As RSNodes use requests and responses to update local information, a merged RSNode could obtain more recent information on average. Second, as the occurrence of "herd behavior" has a positive correlation with the number of independent RSNodes, we could avoid "herd behavior" as much as possible by minimizing the number of merged RSNodes.

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 merged 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 2, 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 located in s_{gi}, which makes $R_{ij} = 1$. We can determine R according to the network topology.

Suppose T is a matrix that describes the traffic composition of each traffic group. In the multi-tier network described in Section 2, 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 get T from traffic statistics collected by NetRS monitors (Section 5.4).

Suppose a NetRS operator o_j could perform replica selection without introducing significant delay or impacts on other applications if NetRS's utilization of its 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. Due to using a separate traffic threshold T_j^{max} for each NetRS operator, our formalization of RSNodes placement is suitable for scenarios of sharing accelerators with other applications. NetRS could effectively exploit the underloaded accelerators by setting higher traffic thresholds for them.

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 policies of resource allocation for applications and the system running status.

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 as 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.

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

$$Subjects\ to:$$

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

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

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

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

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

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

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 a 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. Besides the 3-tier topology shown in Fig. 1, the formalization is applicable to n-tier ($n \in \{1, 2, \ldots\}$) tree-based topologies of data center network.

3.3 Heuristic Algorithm

In modern data centers, there are typically hundreds of switches and thousands of hosts. For large-scale systems, it would take hours to solve the ILP problem, which is an NP-complete problem. However, In practice, we need to determine the RSP in a timely manner to handle unexpected events (e.g., workload changes). Hence, we propose a heuristic algorithm to quickly find an approximately optimal RSP. Our algorithm uses a step-by-step strategy to determine the RSNode for each traffic group sequentially.

We design the heuristic algorithm according to the observation that NetRS operators located in upper tiers (operators having smaller tier ID) are able to aggregate more traffic. Hence, in order to minimize the number of merged RSNodes, our algorithm increases the chance of merging RSNodes by trying to migrate RSNodes from a lower tier to an upper tier. The algorithm also performs horizontal and downward migration to avoid violations of constraints. Due to constraints, we cannot migrate all RSNodes to the upper tier. The algorithm should postpone the upward migration of RSNodes that potentially provides fewer benefits on merging RSNodes. (i) For an RSNode, if the upper tier traffic of its corresponding traffic group is less than a threshold t_{up}, we should postpone its upward migration because migrating the RSNode could introduce too many extra hops.

Considering that the total amount of extra hops is limited, we may migrate more RSNodes to the upper tier by not migrating this one. With a smaller t_{up}, the algorithm may make better decisions at the expense of responsiveness. We can set the t_{up} according to the limitation of extra hops, e.g., a smaller t_{up} under a stricter limitation; (ii) Suppose we have already migrated some RSNodes to the upper tier. For an RSNode, if none of those "migrated RSNodes" could be migrated downward and merged with the RSNode, we should postpone its upward migration. For example, if none of RSNodes in the *Tier*-0 is of *Pod*-1, then we should postpone migrating *Pod*-1 RSNodes from *Tier*-1 to *Tier*-0 (an RSNode being of a *Pod* means that the RSNode selects replicas for traffic from clients in the *Pod*). This rule reduces the chance of splitting a merged RSNode for multiple traffic groups into two (only migrating a part of the merged RSNode to the upper tier).

The details of our algorithm are shown in Algorithm 1. When the traffic statistics of a group are updated, Algorithm 1 first checks whether the total amount of extra hops exceeds the threshold E (*Constraint 3*). It avoids violating *Constraint 3* by migrating the corresponding RSNode downward (lines 2-3). Then Algorithm 1 prevents the traffic flowing through one merged RSNode from being too high (*Constraint 2*) by horizontally or downward migrating the RSNode (lines 4-8). If there is no constraint violations, Algorithm 1 tries to migrate the RSNode upward and looks for opportunities of merging RSNodes (lines 9-15). If the upward migration is denied, we horizontally migrate the RSNode to merge RSNodes in the current tier (lines 11, 14).

Algorithm 1. Update Replica Selection Plan

```
   Params: $g$ (traffic group with newly reported traffic statistics), $t$ (traffic statistics for $g$), $r$ (RSNode for $g$), $c$
           (threshold of multi-confirm times), $o_d$ (destination NetRS operator for $r$)
 1 Function UpdateRSP($t, r, c$)
 2   if Violate Constraint 3 then
 3     $o_d$ = DownwardMigration($r$); return $o_d$
 4   if Violate Constraint 2 then
 5     $o_d$ = HorizontalMigration($r$);
 6     if No RSNode is located in $o_d$ then
 7       $o_d$ = DownwardMigration($r$)
 8     return $o_d$
 9   $o_d$ = UpwardMigration($r$);
10   if Upward migration cannot (potentially) reduce merged RSNodes then
11     $o_d$ = HorizontalMigration($r$); return $o_d$
12   if Upward migration of $r$ should be postponed then
13     if $c >$ confirmed times of migrating $r$ then
14       $o_d$ = HorizontalMigration($r$);
15   return $o_d$
16 end Function
```

In Algorithm 1, we consider that upward migrating an RSNode has the potential of reducing merged RSNodes when at least one of the following cases is true. Suppose the RSNode is currently assigned to the NetRS operator o. (i) We could migrate all RSNodes in NetRS operator o to the upper tier without introducing new merged RSNodes. In this case, we may be able to eliminate one merged RSNodes by migrating all its RSNodes to the upper tier in the future. (ii) We could migrate all RSNodes in NetRS operator o and RSNodes in other x NetRS operators to the upper tier while introducing no more than x new merged RSNodes. In this case, we allow upward migration even when there is no RSNode in the upper tier.

We design a mechanism called multi-confirm to postpone an upward migration. Suppose c is the threshold of multi-confirm times. For an RSNode whose upward migration should be postponed, we perform the migration only if Algorithm 1 determines to migrate the RSNode to the upper tier c times in a row. Before Algorithm 1 determines to upward migrate an RSNode for the c-th time, traffic statistics of other groups should have been reported so that Algorithm 1 can make a more informed decision. Under this premise, we should minimize c as Algorithm 1 could be less responsive with a larger c.

4 NetRS Controller

In this section, we first introduce an exception handling mechanism (Section 4.1) that ensures the high availability of the data store. Then we show how the controller determines the RSNodes placement (Section 4.2) and deals with failures of NetRS operators (Section 4.3).

4.1 Exception Handling Mechanism

NetRS uses a mechanism of Degraded Replica Selection (DRS) to handle exceptions. The DRS requires that clients 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 RSNodes placement problem in Section 3 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 until finding a feasible RSP. (ii) A NetRS operator does not work as expected, e.g the NetRS's utilization exceeds the threshold due to workload changes. The NetRS controller will enable the DRS for some traffic groups that use the NetRS operator as RSNode. (iii) The NetRS operator working as an RSNode fails.

4.2 Determining RSNodes Placement

The NetRS controller needs some inputs to determine the RSNodes placement, including (i) traffic statistics of each traffic group, specifically, each tier's traffic in a traffic group; (ii) The maximum traffic that is allowed for using each NetRS operator as an RSNode; and (iii) the maximum amount of extra hops allowed. These inputs may change due to many causes. The NetRS controller is able to detect input changes due to causes, like changes of workloads, failure of clients, by monitoring the traffic at each NetRS operator. The NetRS controller could also subscribe to events of the system to detect input changes. For example, in the virtualized environment, the migration of virtual machines may imply the remapping of clients and servers to end-hosts, which leads to traffic changes of each traffic group. The system should notify the NetRS controller for some input changes, like changes of resource allocation policies.

Given the required inputs, the NetRS controller can use two approaches to determine the RSP. On one hand, the controller can get the RSP by solving the ILP problem with an optimizer (e.g., Gurobi [22], CPLEX [23]). Using this approach, when inputs have changed, the controller first enables DRS for involved traffic groups to avoid constraint violations. Then it waits for the optimizer to solve the ILP again. On the other hand, the controller can use Algorithm 1 to determine the RSP. With this approach, the controller first initializes the RSP by placing RSNodes in NetRS operators of lower tiers (there should be no constraints violation). For example, the controller could specify the NetRS operator(s) co-located with the rack's ToR switch as the RSNode for requests from the rack. Then, the controller optimizes the RSP by migrating RSNodes according to Algorithm 1. Considering that Algorithm 1 optimizes the RSP by strategically migrating RSNodes to upper tiers, the NetRS controller has to go through this procedure again (first initializes and then optimizes the RSP) when inputs have changed.

4.3 Fault Tolerance

The NetRS framework is able to handle failures of NetRS operators. We assume the NetRS controller be always available because (i) it is not on the critical path of accessing the data store, and (ii) it could be highly available with standbys.

Algorithm 2. Update Replica Selection Plan When Failures Occur

```
Params: $R_f$ (set of RSNodes located in failed NetRS operators), $O_f$ (set of failed NetRS operators), $T_o^{max}$ (max traffic that can use NetRS operator $o$ as RSNodes), $o_r$ (destination NetRS operator for RSNode $r$)
Function FailureUpdateRSP($R_f, O_f$)
    for $r$ in $R_f$ do
        Enable DRS for traffic group of RSNode $r$
    for $o$ in $O_f$ do
        $T_o^{max} = 0$
    for $r$ in $R_f$ do
        $o_r$ = HorizontalMigration($r$);
        if There is no feasible $o_r$ then
            $o_r$ = DownwardMigration($r$)
        Update the RSP (placing $r$ in $o_r$)
end Function
```

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 unavailable NetRS operators from becoming RSNodes while determining a new RSP. As there are lots of mature mechanisms [20] that make the data center network 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 some traffic groups, the NetRS controller enables the DRS for corresponding traffic groups. Since network devices are highly reliable [24] and updating NetRS rules only takes about a few milliseconds [18], failures of NetRS operators

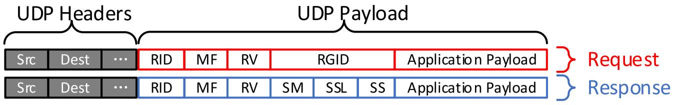


Fig. 2. Packet format of NetRS for the request and the response.

should not significantly affect the system availability. However, due to not taking full advantage of NetRS, the system may experience sudden performance degradation.

To minimize the impact of failures of NetRS operators, we propose Algorithm 2 to quickly find an alternative RSP when failures occur. Algorithm 2 first enables DRS for RSNodes in failed NetRS operators to minimize the impact of failures on the system availability (lines 2-3). Then Algorithm 2 sets the maximum traffic allowed using a failed operator as an RSNode to 0 so that no RSNode would be assigned to them (lines 4-5). For each RSNode in failed operators, we search an alternative operator for it by first trying to migrate it horizontally and then downward (lines 6-10). Algorithm 2 does not perform upward migration. If the NetRS controller determines the RSP by solving the ILP problem, then allowing upward migration makes the placement of RSNodes that are located in failed operators another ILP problem and solving it could be time-consuming. If the NetRS controller determines the RSP using Algorithm 1, then Algorithm 1 should be the one to handle upward migrations.

5 NetRS Operator

This section describes the NetRS operator. We first introduce the packet format of NetRS (Section 5.1). Then, we show the processing pipeline of a programmable switch according to NetRS rules (Section 5.2), and the working procedure of a NetRS selector running on the network accelerator (Section 5.3). Finally, we present how NetRS monitors collect traffic statistics (Section 5.4).

5.1 Packet Format

The packet format plays an important role in propagating information. Clients, servers, switches and network accelerators should agree to the common format. As stateful network protocols (e.g., TCP) introduce latency overheads, recent latency-critical data stores [17], [25] generally use stateless network protocols (e.g., UDP). Some data 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 data 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 states. 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. In order to reduce bandwidth overheads of NetRS protocol, we use separate packet formats for requests and responses to carry different information. Fig. 2 shows the packet format of request and response, respectively. The request and response packet have the 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 indicates 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 as 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 sent, 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 obtains all replicas that can handle the request by querying its local database of replica groups with the RGID. The size of the database is small because latency-critical distributed data stores typically use consistent hashing to place data. Using RGID make 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. In the case of server failures, clients can inform the NetRS selector of the availability of each corresponding replica through RGID. In the last x bits of RGID, if the ith bit is 0, the NetRS selector will exclude the ith replica for selection.

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.

5.2 NetRS Rules

The NetRS controller updates NetRS rules of each NetRS operator based on the RSP. Each NetRS operator relies on its NetRS rules to forward packets 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: non-NetRS packet, NetRS request, and NetRS response. The switch uses the segment of a magic field in a packet to determine the type of a 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 NetRS packets, including NetRS requests and responses.

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

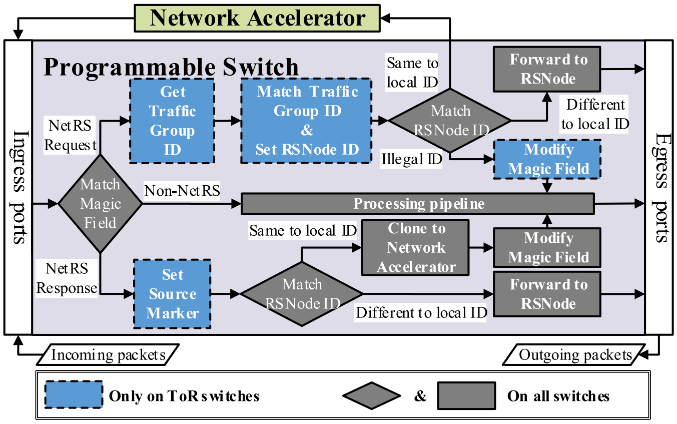


Fig. 3. NetRS rules within ingress pipelines of a programmable switch.

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 not in the switch's local ID(s), then the switch will forward the packet to the next hop towards the RSNode. Otherwise, if the RSNode ID is the same as one of the local ID(s), 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 5.3).

When a NetRS response arrives, 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 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 5.4). By cloning the packet of NetRS response, we could avoid the latency overhead of packet processing in accelerators.

As the RSP and traffic groups are agnostic to end-hosts, clients of the data store are unable to determine the RSNode ID for NetRS requests. With the network topology described in Section 2, 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 5.4). 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 response.

In order to enable the DRS for a traffic group (Section 4.1), 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 $f(M_{drs})$ to its magic field, where $f(\cdot)$ is an invertible function.

NetRS assumes requests of single packet. As the main content of a reading request is the data's name, which is typically very small [7], a reading request can usually fit a single packet well. NetRS is able to handle responses of multiple packets. With NetRS, servers only need to label the last packet in a response as a NetRS response and label others as non-NetRS packets.

5.3 NetRS Selector

The NetRS selector is responsible for performing replica selection and maintaining corresponding local information.

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 last packet of a response to $f^{-1}(m)$, where m is the magic field value of the corresponding request. This mechanism guarantees that (i) the server marks the last packet in a response as a NetRS response, only if its corresponding request had flowed through a NetRS selector; (ii) the NetRS monitor could recognize the response of a request using DRS (for a packet with magic field M_{drs}, the ToR switch will set the source marker and modify the magic field to M_{mon}). 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.

5.4 NetRS Monitor

The NetRS monitor collects traffic statistics 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 introduces lots of 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 the data 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.

6 Evaluation

We conduct simulation-based experiments to extensively evaluate the NetRS framework. We do not evaluate with NetRS with a small-scale prototype because NetRS concentrates on the difficulties in performing in-network replica selection in the context of datacenter-scale systems. In order to conduct large-scale experiments, the common practice is using simulations.

6.1 Simulation Setup

In our experiments, we use the simulator from C3 [9], which simulates clients and servers of a data store. 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) [21] containing 1024 end-hosts. Each switch has one network accelerator attached to it.

We set major parameters in our evaluation based on the experimental parameters in C3 [9]. The service time of a server follows exponential distribution while the mean value (t^{kv}) is 4 ms. Each server could process $N_p (N_p = 4)$ requests in parallel. The performance of each server fluctuates independently with an interval of 50 ms. The fluctuation follows the bimodal distribution [26] with the range parameter $d = 4$ (in each fluctuation interval, the mean service time could be either t^{kv} or t^{kv}/d with equal possibility). Data objects 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 data object out of 100 million data objects according to the Zipfian distribution (the Zipf parameter is 0.99). For each experiment, the data 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 [17]. Specifically, the RTT 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 by default, unless otherwise noted. 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 [9], which is state-of-the-art). C3 aims to minimize the product of queue length and service time across each server. Each RSNode estimates the status of a server according to the queue length and the service time piggybacked in responses, the number of the server's pending requests from the RSNode (pr_s), and the number of independent RSNodes in the system. C3 tries to avoid the "herd behavior" of RSNodes by ranking replicas accounting pr_s. C3 penalizes long queues by amplifying the estimated queue length of each server with a cubic function.

- *CliRS*: A commonly used replica selection scheme in latency-critical data stores [5], [6], [30], [31]. 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 [8].
- *NetRS-ToR*: Using the NetRS framework for replica selection with a straightforward ToR-based RSP, which 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.
- *NetRS-ADP*: Using the NetRS framework for replica selection with a ToR-based RSP initially, and then the NetRS controller adaptively optimizes the RSP using Algorithm 1.
- *Oracle*: Each client selects replica with the minimal $t_s \cdot q_s$, where t_s and q_s is the service time and queue length of a server, respectively. Oracle is only a theoretical scheme. Clients using Oracle are able to obtain the instantaneous t_s and q_s of each server.

6.2 Results and Analysis

This section provides experimental results in a variety of scenarios. Under open-loop workloads, considering that the system throughput is fixed, we concentrate on the response latency of system. By default, the aggregate arriving rate of requests (A) corresponds the 90 percent system utilization ($\frac{t^{kv}A}{N_s N_p}$), which is low considering the perfermance fluctuation ($\frac{2}{1+d}\frac{t^{kv}A}{N_s N_p} = 36\%$). Under closed-loop workloads, we evaluate the throughput of system with different concurrency levels. In our deployment, $U = 25$ and $E = 10$ percent A, where U is the maximum NetRS's utilization allowed for an accelerator, and E is the maximum amount of extra hops.

6.2.1 Response Latency Under Open-Loop Workloads

Oracle outperforms other schemes in reducing response latency. It is because, compared with other schemes, Oracle uses the most recent status of servers. Clients are able to avoid selecting poorly-performing servers and reduce the occurrence of "herd behavior".

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 and 5 do not show bars exceeding the respective latency thresholds.

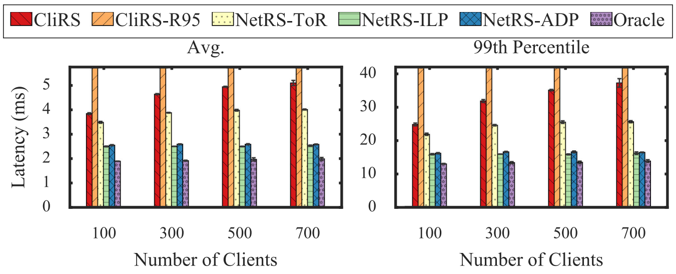


Fig. 4. The comparison of latency with varying number of clients.

When no unexpected event occurs, the latency with NetRS-ADP is comparable to the latency with NetRS-ILP.

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) NetRS-based schemes outperform CliRS, and NetRS-ILP shows the best performance. Compared with CliRS, NetRS-ILP reduces the mean latency by 34.9-50.3 percent and the 99th latency by 35.7-56.3 percent. Compared with NetRS-ToR, NetRS-ILP reduces the mean latency by 34.4 percent and the 99th latency by 34.1 percent on average. (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-based schemes regardless of changes in the number of clients. The underlying reason is that, with NetRS-based schemes, 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 lead to worse replica selection and performance penalties.

Impact of the System Utilization. Fig. 5 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 percent. Compared with CliRS, NetRS-ILP reduces the mean latency by 15.5-49.4 percent and reduces the 99th latency by 9.8-54.7 percent. Compared with NetRS-ToR, NetRS-ILP reduces the mean and 99th latency by 15.4-37.2 percent and 9.6-37.7 percent, respectively. We make 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, NetRS-based schemes introduce more reductions 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.

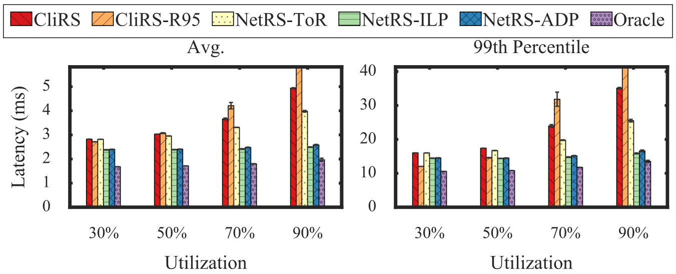


Fig. 5. The comparison of latency with varying system utilization.

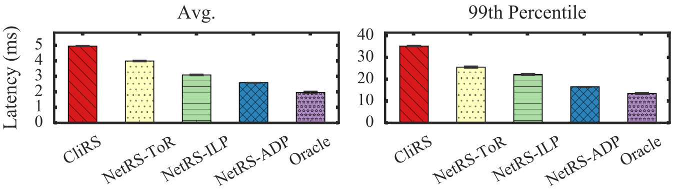


Fig. 6. The comparison of latency with workload changes.

Impact of Unexpected Workload Changes. Fig. 6 shows the response latency comparison of different schemes when unexpected workload changes occur (the system utilization increases from 50 to 90 percent). While using NetRS-ILP, the NetRS controller enables degraded replica selection for some traffic groups so that the unexpected workload changes will not cause any constraints violations. While using NetRS-ADP, the NetRS controller initializes the RSP with a ToR-based RSP and optimizes the RSP according to Algorithm 1. Compared with CliRS, NetRS-ADP reduces the mean latency by 47.5 percent and the 99th latency by 52.8 percent. Compared with NetRS-ILP, NetRS-ADP reduces the mean latency by 16.2 percent and the 99th latency by 25.1 percent. NetRS-ADP outperforms NetRS-ILP because it avoids constraints violations without enabling DRS.

Impact of NetRS Operator Failures. Fig. 7 depicts the response latency comparison of two approaches to handle failures of NetRS operators that work as RSNodes. (i) *ENABLE-DRS* enables DRS for traffic groups that using the failed operator as RSNode. (ii) *ALTER-RSP* find an alternative RSP using Algorithm 2. For experiments using the NetRS-based scheme, they start with an RSP determined by solving the ILP problem. Then, after clients have issued about 1/5 of all requests, a NetRS operator that works as an RSNodes fails. Compared with CliRS, the NetRS-based scheme reduces the mean and 99th response latency whatever the approach to dealing with failures. Compared with ENABLE-DRS, ALTER-RSP reduces the mean and 99th latency by 18.8 and 20.3 percent, respectively. The performance gain of using ALTER-RSP is due to not relying on DRS for traffic groups that use the failed operator as RSNodes.

Impact of Network Congestion. Fig. 8 shows the response latency comparison of different schemes under network congestion. In the experiments, the latency of a switch can be 3 ms due to congestion. The congestion status of a switch changes dynamically. We compare an extra scheme, ProxyRS, which uses a centralized end-host as the RSNode. In this scenario, NetRS can also significantly cut the latency both on average and in the tail. Compared with ProxyRS, using NetRS-ILP reduces the 99th latency by 14.4 percent, the average latency

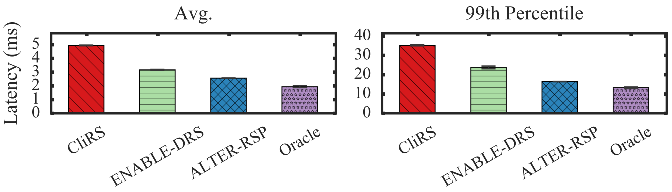


Fig. 7. The comparison of latency with NetRS operator failures.

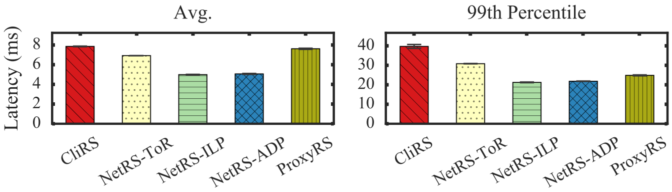


Fig. 8. The comparison of latency with network congestion.

by 34.4 percent. ProxyRS is able to provide comparable tail latency with NetRS-ILP due to selecting better replicas. However, since requests should go through long network paths, using ProxyRS makes the mean latency higher.

Impact of Accelerator Utilization. Fig. 9 shows the impact of the accelerator utilization on response latency. In the experiments, we use the NetRS-ADP scheme of replica selection with accelerator utilization ranging from 0.0 to 25 percent. The NetRS-ADP turns into CliRS when the utilization is 0.0 percent. We can observe that the latency is lower with higher accelerator utilization at first, however, further improving the utilization doesn't help reduce the latency. It is because, an accelerator could accommodate more RSNodes with a higher utilization threshold, which helps to reduce the number of RSNodes in the system. However, when there are only a few RSNodes, further reducing RSNodes has negligible impacts on the response latency.

6.2.2 Throughput Under Closed-Loop Workloads

Fig. 10 shows the throughput comparison of different replica selection schemes when the concurrency level of workload varies from 200 to 800. Compared with CliRS, NetRS-ADP improves the throughput by 37.4-56.2 percent. We observe that better replica selection has a more significant impact on throughput at lower concurrency levels. In order to improve the throughput, the system has to make full use of available replicas. A higher concurrency level implies that there are more requests in the system, which makes it easier to keep all corresponding replicas busy. Hence, at higher concurrency levels, NetRS-ADP and Oracle have fewer advantages over CliRS.

In summary, (i) NetRS could effectively improve the performance of data stores compared with selecting replica by clients; (ii) a proper RSNodes placement plays an important role in the performance improvement of NetRS; (iii) redundant requests are only suitable for scenarios of low utilization; (iv) NetRS is robust against unexpected events and network congestion.

7 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 [8]. Vulimiri *et al.* [32] suggest that the use of redundant requests is a trade-off between response latency and system utilization. Shah *et al.* [33] and Gardner *et al.* [34] 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 [35] 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 [9] 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 replica selection via performing replica selection in data center network.

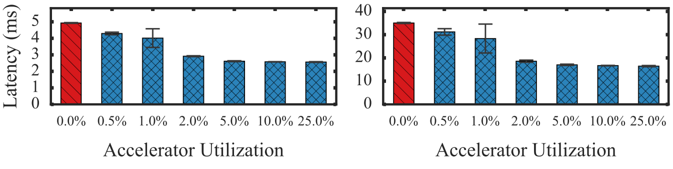


Fig. 9. The comparison of latency with varying accelerator utilization.

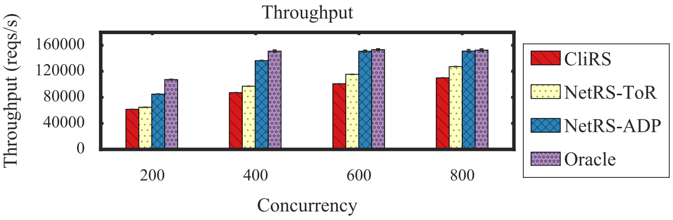


Fig. 10. The comparison of throughtput with varying concurrency levels.

In-Network Computing. In-network computing is widely used to enhance the performance of data stores. NetCache [36] caches hot data within programmable switches to balance loads of backend servers. IncBricks [17] builds an in-network cache system that works as a cache layer for key-value stores. Different from NetRS, which deals with the performance fluctuation of servers, these works leverage in-network computing to address the problem of workload skewness in data stores.

8 Conclusion

This paper presents NetRS, a framework that enables in-network replica selection for latency-critical data stores. NetRS exploits programmable switches and network accelerators to aggregate tasks of replica selection. Compared with client-based replica selection, 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 data center networks. We also propose algorithms to quickly find an approximately optimal solution to this NP-complete placement problem. Moreover, NetRS is able to handle exceptions, e.g., failures of network devices.

Acknowledgments

This work was supported by NSFC Grant No. U1705261, 61772222, and 61772212, Shenzhen Research Funding of Science and Technology - Fundamental Research (Free exploration) JCYJ20170307172447622, NSFC Grant No. 61821003, and 61832007, National Key R&D Program of China No. 2018YFB10033005, Hubei Province Technical Innovation Special Project (2017AAA129), Wuhan Application Basic Research Project (2017010201010103), Project of Shenzhen Technology Scheme JCYJ20170307172248636, Fundamental Research Funds for the Central Universities. This article was presented in part at the IEEE 38th International Conference on Distributed Computing Systems, Vienna, Austria, Jul. 2–6, 2018.

References

[1] H. Xiao *et al.*, "Towards web-based delta synchronization for cloud storage services," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 155–168.

[2] Z. Li *et al.*, "Efficient batched synchronization in dropbox-like cloud storage services," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2013, pp. 307–327.

[3] Z. Li *et al.*, "Towards network-level efficiency for cloud storage services," in *Proc. ACM Internet Meas. Conf.*, 2014, pp 118–128.

[4] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. USENIX NSDI*, 2013, pp. 385–398.

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

[6] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ.*, 2007, pp. 205–220.

[7] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS Perf. Eval. Rev.*, 2012, pp. 53–64.

[8] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[9] L. Suresh *et al.*, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 513–527.

[10] Y. Hua, "Cheetah: An efficient flat addressing scheme for fast query services in cloud computing," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.

[11] C. Li, D. Feng, Y. Hua, and F. Wang, "A high-performance and endurable SSD cache for parity-based RAID," *Frontier Comput. Sci.*, vol. 13, no. 1, pp. 16–34, 2019.

[12] Y. Su, D. Feng, Y. Hua, Z. Shi, and T. Zhu, "NetRS: Cutting response latency in distributed key-value stores with in-network replica selection," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 143–153.

[13] "Barefoot tofino: P4-programmable ethernet switch ASICs," 2017. https://barefootnetworks.com/products/brief-tofino/

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

[15] "Netronome NFE-3240 Family appliance adapters," 2017. https://www.netronome.com/products/nfe/

[16] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[17] M. Liu *et al.*, "IncBricks: Toward in-network computation with an in-network cache," in *Proc. 22nd Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2017, pp. 795–809.

[18] J. Li *et al.*, "Just say no to paxos overhead: Replacing consensus with network ordering," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 467–483.

[19] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 104–120.

[20] A. Singh *et al.*, "Jupiter rising: A decade of CLOS topologies and centralized control in Google's datacenter network," *Commun. ACM*, vol. 59, no. 9, pp. 88–97, 2016.

[21] M. Al-Fares , A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 63–74, 2008.

[22] "Gurobi optimizer: State-of-the-art mathematical programming solver," 2017. http://www.gurobi.com/products/gurobi-optimizer

[23] "CPLEX Optimizer: High-performance mathematical programming solver," 2017. https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer

[24] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.

[25] X. Li *et al.*, "Be fast, cheap and in control with SwitchKV," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 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 Endowment*, vol. 3, no. 1–2, pp. 460–471, 2010.

[27] D. Meisner *et al.*, "Power management of online data-intensive services," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 319–330.

[28] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp 267–280.

[29] A. Roy *et al.*, "Inside the social network's (datacenter) network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 123–137, 2015.

[30] "Project voldemort: A distributed database," 2017. http://www.project-voldemort.com/voldemort

[31] "Couchbase data platform: Couchbase server," 2017. https://www.couchbase.com/products/server

[32] A. Vulimiri *et al.*, "Low latency via redundancy," in *Proc. 9th ACM Conf. Emerg. Netw. Experiments Technol.*, 2013, pp. 283–294.

[33] N. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency ?" in *Proc. 51st Annu. Allerton Conf. Commun. Control Comput.*, 2013, pp. 731–738.

[34] K. Gardner *et al.*, "Reducing latency via redundant requests: Exact analysis," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2015, pp. 347–360.

[35] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.

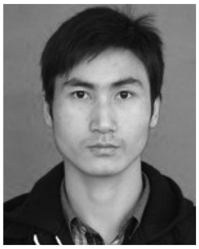
[36] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 121–136.

Yi Su received the BS degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2012. He is currently wirking toward the PhD degree at the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. He has several publications in major journals and international conferences, including *Journal of Parallel and Distributed Computing*, ICDCS, and ICPP. His research interests include cloud storage systems, big data processing systems.

Dan Feng (Member, IEEE)received the BE, ME, and PhD degrees in computer science and technology, from the Huazhong University of Science and Technology (HUST), China in 1991, 1994, and 1997, respectively. She is currently a professor and the dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel & Distributed Systems*, *ACM Transactions on Storage*, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She has served as the program committees of multiple international conferences, including SC 2011, 2013, MSST 2012, 2015. She is a member of ACM.

Yu Hua (Senior Member, IEEE) received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is currently a full professor with the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 100 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers* (*TC*), *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), USENIX ATC, USENIX FAST, INFOCOM, SC and ICDCS. He has been on the program committees of multiple international conferences, including USENIX ATC, RTSS, INFOCOM, ICDCS, MSST, ICNP and IPDPS. He is a senior member of the ACM and CCF, and a member of USENIX.

Zhan Shi received the BS and master's degree in computer science, and the PhD degree in computer engineering from the Huazhong University of Science and TechnologyHUST, China. He is working with the Huazhong University of Science and Technology (HUST) in China, and is an associate researcher in Wuhan National Laboratory for Optoelectronics. His research interests include storage management, distributed storage system, and cloud storage.



Tingwei Zhu received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012. He is currently working toward the PhD degree in computer architecture in HUST. His interests include software-defined networking and distributed storage systems. He has several publications in major journals and international conferences, including *Transactions on Networking*, IWQoS, ICPP, and *Journal of Network and Computer Applications*.