The Network-Integrated Storage System

Ibrahim Kettaneh, Ahmed Alquraan, Hatem Takruri, Suli Yang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Samer Al-Kiswany

Abstract—We present NICE, a key-value storage system design that leverages new software-defined network capabilities to build cluster-based network-efficient storage system. NICE presents novel techniques to co-design network routing and multicast with storage replication, consistency, and load balancing to achieve higher efficiency, performance, and scalability.

We implement the NICEKV prototype. NICEKV follows the NICE approach in designing four essential network-centric storage mechanisms: request routing, replication, consistency, and load balancing. Our evaluation shows that the proposed approach brings significant performance gains compared with the current systems design: up to 7× put/get performance improvement, up to 2× reduction in network load, 3× to 9× load reduction on the storage nodes, and the elimination of scalability bottlenecks present in current designs.

Index Terms—Key-value storage, software-defined networks, network-integrated design, network-system co-design, distributed storage

1 INTRODUCTION

The end-to-end design principle [1] pervades the design of virtually every distributed system [2, 3, 4, 5, 6]. In its extreme form, critical functionality is implemented solely in end hosts, with a relatively dumb and fast network to connect them.

One locale that closely adheres to the end-to-end principle is distributed storage, including distributed file systems [7, 8, 9, 10, 11, 12] and scalable key-value stores [13, 14, 15, 16, 17]. In these widely deployed and increasingly important systems, the network is used as a point-to-point communication medium, while storage logic and protocols are implemented entirely in client libraries and server code.

Unfortunately, such Network-Oblivious (NOOB) storage systems are fundamentally inefficient. Consider, for example, the simple task of replicating a data block. To do so, a node first sends the block to one server, and then another, and then another; as a result, the same data redundantly traverses some number of network links and switches, increasing load on the network significantly. Even the simple task of locating a data item presents a significant challenge; for example, in protocols such as Chord [18], a logarithmic number of nodes must be contacted simply to discover the location of a particular key.

In this paper, we propose an alternative approach in which we co-design storage logic and networking support to realize more efficient, scalable, and reliable distributed storage. Such Network-Integrated Cluster-Efficient (NICE) storage harnesses recent advances in Software-Defined Networks (SDNs) [19, 20] to optimize key aspects of modern distributed storage architectures. For example, NICE storage systems can replicate a block while generating the least possible network load, and it can forward a request to the proper node in a single hop.

Two recent developments provide a unique opportunity to address NOOB inefficiencies and indicate that a network-integrated design paradigm that co-designs network and end-point functionality has a much higher chance of being successful today. First, recent advances in software-defined networks (SDNs) provide a standard interface for implementing in-network application specific optimizations, and for building a control mechanism that can orchestrate network and storage operations. The second development is the wide adoption of data centers as the main cloud-computing platform. Having a single administration of the entire hardware/software stack and the ability to compartmentalize the infrastructure facilitates adopting custom solutions for different applications or subsystems.

NICE uses SDN technology to virtualize the storage system. The client accesses a virtual storage system deployed on a range of virtual IP addresses. The NICE network controller modifies client packets and forwards them to the proper storage node. Having a network controller that is informed of the storage system metadata and has full control of the network decisions enables optimizing packet paths to improve four essential storage mechanisms, including: request routing, which directs requests from clients to storage nodes; replication, for preventing data loss when nodes or storage devices fail; load balancing, which dispatches client requests across replicas to handle workload variation. Finally, NICE virtualization simplifies building consistency protocols by making failed nodes, or nodes with inconsistent data, inaccessible.

We implemented NICEKV, a key-value storage system following the NICE design. Our NICEKV prototype leverages the capabilities of the widely adopted OpenFlow standard. Our evaluation using synthetic and real workload benchmarks shows that NICEKV brings significant performance gains compared to a broad set of NOOB storage configurations and two production systems: Ceph and Swift. Our evaluation shows that NICEKV has a scalable membership maintenance mechanism, achieves single-hop request routing eliminating the need for deploying load balancer, and achieves network and storage optimal replication, effectively halving the network-generated load and reducing storage load by 3× to 9×, depending on replication level. NICEKV load balancing effectively spreads client requests across servers without deploying dedicated load-balancing boxes. The combination of these optimizations is powerful; the NICEKV prototype can achieve up

- Ibrahim Kettaneh, Hatem Takruri, Ahmed Alquraan, and Samer Al-Kiswany are with the Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada.
- Suli Yang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, are with the Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI, 53706, USA

to 7× put/get performance improvement as compared to the traditional network oblivious approach.

Furthermore, we explored the potential of using the recent programmable switches [21] to build a storage-aware load balancing techniques. We implemented NICEKV-P4 a key-value storage system using the P4 programming language. Our evaluation shows demonstrates the flexibility of this approach.

The rest of this paper is organized as follows. We present an overview of the NOOB systems design, and discuss the recent advances in software-defined networks (Section 2). In Section 3 we present the NICE architecture, detail the system design in Section 4, present the implementation of the NICEKV prototype in Section 5, and present our empirical evaluation in Section 6. We discuss related work in Section 7, and conclude in Section 8.

2 BACKGROUND AND RELATED WORK

In this section, we present an overview of a typical network-oblivious storage systems design, and summarize the recent advances in software-defined networks.

2.1 NOOB Storage System Design

Current distributed key-value storage systems are fundamentally inefficient as they are network-oblivious (NOOB): the network is used as a black-box point-to-point communication medium without any application-informed optimization of its operations, while storage logic and protocols are implemented by end hosts. This NOOB approach for designing distributed system is inefficient for storage systems as many core storage operations are, in principle, network-level operations, e.g., replication or request routing.

Many NOOB storage systems adopt a design based on consistent hashing [22]. In the original consistent-hashing design, the object hashing space represents a circular ring, all storage nodes are placed on the ring, and each node coordinates access to the objects in its part of the ring. Pastry [23] and Chord [18] were among the first to use consistent hashing to build a scalable peer-to-peer object storage system. They use, with high probability, *O(log n*) hops to route a request, while only storing *O(log n)* routing information on each node. While this approach scales well, it imposes additional latency.

To reduce the latency of request routing, prominent NOOB storage systems adopt a full-membership model [13, 14, 15, 16, 17], in which every node maintains complete knowledge about all the nodes in the system and their contents; hence, nodes can route any request directly to the responsible node. When a node joins or fails, all the nodes in the system need to be updated. This update happens through contacting every node and updating its information using *O(N)* connections and messages [13], or through an epidemic protocol entailing *O(log n)* steps and over *O(N)* messages [24].

Access Mechanism. Current key-value storage systems employ one of four techniques to route client requests to the node maintaining the object. First is the Replica-Oblivious Gateway (ROG), uses an off-the-shelf load balancer to forward requests to randomly selected storage node. If the selected node does not have the requested key, the node will forward the request to the node that maintains the object. This approach is common in current systems [14, 16, 25] due to its ease of deployment and use of existing load balancers. Unfortunately, as the load balancer is oblivious to the replicas' content, it will forward the majority of requests to a replica that does not maintain the object. Consequently, for the majority of requests this approach adds additional two hops for routing a request through the load balancer and the randomly-selected storage node.

The second approach, is the Replica-Aware Gateway (RAG). Similar to the previous approach, this approach uses a load balancer, but the load balancer is aware of the contents of each replica and can accurately forward the request to a replica that maintains the object. Consequently, this approach imposes one extra hop for routing a request.

Third is the Replica-Aware Client (RAC), in which clients cache the storage IP address of previously accessed objects [26], and use it to route subsequent requests. This approach achieves single-hop routing as requests are directly sent from a client to the replica storing the object. This approach only works in deployments in which it is permissible for clients to obtain detailed data placement and replication information. For deployments in which the clients do not have access to storage internal information or are located behind a NAT [27] (e.g., shared cloud storage like Amazon S3), this approach is not viable. Finally, this approach hinders deploying advanced load balancers as each client accesses the replicas directly.

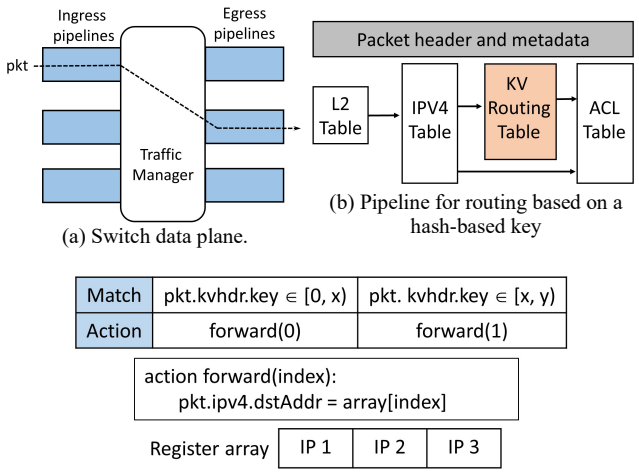
The fourth approach is the Replica-Aware Proxy (or proxy for short). This approach uses a proxy node that is aware of the replica placement to forward the request [13]. Unlike the previous three approaches, the proxy node is on the data path for get and put operations. The data for get operations is sent to the proxy, which then sends it to the client.

2.2 Software-Defined Networks

The software-defined networking paradigm re-architects the network into two planes: data and control. The data plane is a packet-forwarding plane that uses the information available in the switch forwarding tables to forward packets. The control plane is a software based control logic, typically deployed on an external server (i.e., not on the switch). Recent technology advances enable customizing the control and data planes.

Flexible Control Plane. The control plane enables application to control multiple switches by modifying the rules in the switches forwarding tables. To update a switch forwarding table, the controller uses the OpenFlow standard API [20], a widely popular standard interface used to communicate with SDN capable switches. OpenFlow [19] allows modifying (i.e., inserting or deleting) the forwarding rules of a single switch. Each forwarding entry includes a matching rule and an action list. Matching uses wildcard matching rules on any field in the packet standard headers, including IP and MAC addresses, and protocol and port numbers. If a packet matches a rule, the switch performs the actions associated with that rule. The action list may contain multiple actions that are performed in order. The current OpenFlow standard defines a set of actions including packet forwarding to a specific switch port, packet drop, forwarding a packet to the controller, or modifying fields in a packet. The possible modifications include changing the source/destination MAC/IP addresses.

For packets that do not have a matching rule, the switch will forward the packet to the controller, which significantly increases packet latency. To avoid this inefficient path the switch caches the forwarding rules with a controller specified expiry period.



(a) Switch data plane.

(b) Pipeline for routing based on a hash-based key

(c) Simple match-action stage for routing based on a hash-based key for the KV routing table in subfigure (b)

Figure 1. Switch data plane.

Limitations. While OpenFlow significantly increases the flexibility of the network its capabilities are limited. Mainly, its data plane is rigid and cannot be extended. OpenFlow can only support current standard packet headers (i.e., does not facilitate defining custom application headers), and the actions supported are fixed and limited. Finally, current switch that support OpenFlow only implement a subset of OpenFlow features (detailed in Section 5).

Programmable Data Plane. To address the limitations of OpenFlow the research community developed a new generation of programmable switches that allow programming the data plane. Programmable switches allow the implementation of an application-specific packet-processing data plane that can process custom packet headers and is deployed on network devices and executed at line speed. While this technology did not yet garner wide adoption as OpenFlow, a number of vendors are already producing network-programmable ASICs, including Barefoot's Tofino [28] and Cavium's XPliant [29].

Figure 1(a) illustrates the basic data plane architecture of modern programmable switches. The data plane contains three main components: ingress pipelines, a traffic manager, and egress pipelines. A packet is first processed by an ingress pipeline before it is forwarded by the traffic manager to the egress pipeline that will finally emit the packet.

Each pipeline is composed of multiple stages. At each stage, one or more tables match fields in the packet header or metadata; if a packet feild matches, the corresponding action is executed. Programmers can define custom per-packet headers as well as custom actions. Each stage has its own dedicated resources, including tables and register arrays (a memory buffer). Figure 1(b) shows a simple example of a pipeline that routes a request to a key-value store based on the key, and Figure 1(c) shows the details of the KV routing stage in Figure 1(b). The stage forwards the request based on the key in the packet's custom L4 header. The programmer implements a forward() action that accesses the register array holding nodes' IP addresses. An external controller can modify the register array and the entries in the table.

Programmers use domain-specific languages like P4 [21] to define their own packet headers, define tables, implement custom actions, and configure the processing graphs.

These recent advances in networking technology enable fine-grained control of network operations and facilitate application-optimized traffic engineering. In this paper we explore techniques to leverage these new OpenFlow and P4 capabilities to accelerate storage systems.

3 NICE SYSTEM ARCHITECTURE

NICE leverages software-defined networking capabilities to optimize storage system operations. We focus our attention to leveraging the OpenFlow capabilities due to the wide adoption of this technology. In Section 4 we extend our design to leverage the capabilities of P4-programmable switches.

NICE exploits the OpenFlow flexibility and fine-grained control [19, 20] to co-design network and storage operations. The NICE design virtualizes the storage system. The client accesses a virtual storage system deployed on a range of virtual IP addresses. The metadata service (detailed next) maps the virtual storage system to the physical one. The NICE design optimizes this mapping to achieve low-latency routing, efficient multicasting, load-balancing, and improved fault tolerance.

In the rest of this section, we first present the NICE architecture, then detail the two core techniques we propose: storage virtualization, and consistency-aware fault tolerance. The following section details how we extend these techniques to optimize replication, consistency, and load-balancing mechanisms.

3.1 System Architecture

Similar to the NOOB storage, NICE uses consistent hashing to partition the object space among the storage nodes. Nodes are placed in a consistent hashing ring, such that each node serves part of the ring. We call this the *physical ring*. Every storage node is the primary replica for one or more partitions, and can serve as a secondary replica for other partitions.

The system is composed of three components (Figure 2): storage nodes, client nodes, and a metadata service, all connected with an OpenFlow-enabled switching fabric. The storage nodes serve put and get requests and implement the replication, consistency, and load-balancing protocols. The storage nodes send periodic heartbeats to the metadata service. The metadata service maintains storage system metadata. The metadata includes information about which storage nodes are participating in the system, and which range of the hash space (partition) each storage node is serving. The metadata service does not maintain per-object metadata.

3.2 NICE Storage Virtualization

The first goal of virtualizing the storage system is to enable storage-aware routing of client requests; that is, to have a routing technique that can route a client request to the proper storage node (i.e., routing based on the key hash value). Building a storage-aware routing mechanism is challenging. While OpenFlow provides control over switch forwarding decisions, it only supports matching packets using information found in the packet headers (e.g., Ethernet, IP, UDP or TCP), not the packet payload data. Consequently, routing packets based on the key hash carried in the payload is not possible. Alternatively, allowing the client to know

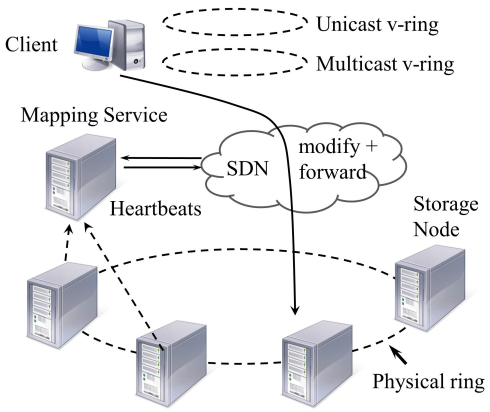


Figure 2. System Architecture. The client sends the requests using two virtual rings (vrings). The requests are rerouted in the network to the responsible storage node. The metadata service receives heartbeats from the nodes and maintains the mapping information in the forwarding tables.

the physical-ring mapping and replica-placement inherits the NOOB RAC limitations.

The NICE approach virtualizes the storage system; the client accesses a virtual storage system deployed on a set of *virtual nodes (vnodes)*. The virtual addresses are organized in a *virtual consistent hashing ring (vring)*. For instance, all the IP addresses in the range of 10.10.0.0 to 10.10.255.255 can be virtual nodes in a vring. The number of vnodes and their addresses are configurable and do not correspond to the physical ring configuration. To access the system, the client hashes the object name and finds the vnode responsible for serving the object. The client sends the put/get request to the vnode address using UDP.

The metadata service maps the virtual ring to the physical ring. It maps a subset of virtual addresses to a single physical node address. While different mapping techniques are possible, we use simple IP prefix matching: we divide the virtual ring addresses into subgroups such that the number of vnodes per subgroup is a power of 2 (e.g., all vnodes in 10.10.1.0/24 form a subgroup). The metadata service maps any packets sent to a particular subgroup to a particular physical node. To this end, the switch will *modify* the destination IP and MAC addresses in the packet headers to be the IP and MAC addresses of the primary replica, then *forward* the packet to the switch port of the primary replica.

This mapping technique achieves three benefits. First, it achieves low-latency single-hop routing, as the client requests are directly routed in the network to the responsible node at switching speed. Second, by decoupling the virtual ring from the physical ring this technique simplifies deployment, as clients never need to change their virtual ring configuration, even when the physical ring configuration changes. Finally, this approach allows for multiple vnodes to be mapped to a single physical node, improving performance and load balancing [18]. Compared with NOOB request routing, NICE routing achieves the optimal routing latency of the RAC approach without suffering from its limitations.

3.3 Consistency-Aware Fault Tolerance

To guarantee sequential consistency NOOB storage systems use complex consistency protocols like two-phase (2PC), three-phase commit [24], Paxos [30, 31], or Raft [32]. We illustrate in Figure 3 the put operation using the 2PC protocol, as a representative of these protocols to simplify our discussion. 2PC is among the early proposed protocols that are still widely used [7, 33, 34, 35].

Failure handling is a main differentiating factor between consistency protocols. The 2PC commit protocol is brittle in face of node failures during the put operation and may block if the primary node fails. To overcome the 2PC problems, Paxos uses a majority-based (i.e., quorum) design, in which at least the majority (but not all) of the nodes need to participate in the put operation. The drawback of this approach is that failed nodes (or disconnected nodes) may have stale data when they join back; consequently, it is necessary to access the majority of the nodes during the get operation as well to guaranty consistency. This approach creates unnecessary high overhead during get operations. An alternative approach is to send get requests only to the primary (a.k.a. leader) node (e.g., Raft). Unfortunately, this approach does not scale because all put and get requests are served by a single node.

We propose a consistency-aware fault tolerance mechanism. This mechanism solves the inefficiency problem found in current protocols by allowing any storage node with consistent data to serve get requests. The mechanism hides inconsistent nodes, including failed and newly joining nodes, until they have a consistent version of the data. To this end, when a node fails it is removed from the switch mapping, rendering the node inaccessible from the client's point of view. When a node restarts, it joins the system in two phases. First, it is made accessible to other storage nodes and to client put requests only. During this phase the rejoining node will receive new objects and will fetch consistent versions of the objects that have been changed while the node was offline. Second, when the node has consistent data, it is made accessible for clients' get requests. This approach simplifies building consistency protocols (as we will see next) by guaranteeing that clients can only access consistent nodes.

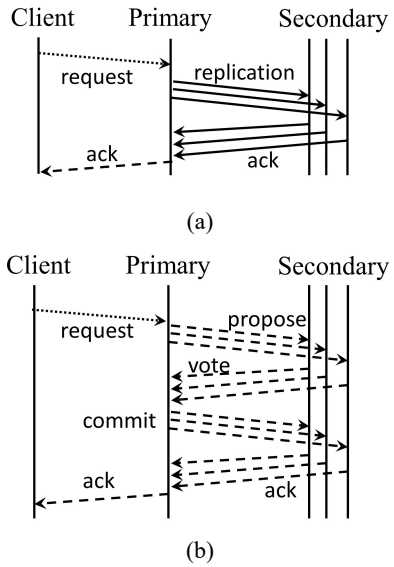


Figure 3. Put protocol alternatives. The figure shows (a) the primary-backup and (b) the 2PC put protocols. In the primary-backup design (solid arrows) the primary replica serves all put and get request. In the two-phase commit (2PC) design (dashed arrows), two rounds are needed to guarantee consistency.

4 SYSTEM DESIGN

In this section we first detail the design of system metadata service, then we extend the core techniques of NICE to build an efficient replication mechanism, improve the consistency protocol efficiency, and provide in-network load balancing.

4.1 Metadata Service Design

The metadata service is the only component that maintains the system membership and metadata, i.e., it has complete knowledge of all storage nodes in the system and the physical ring partitions they serve. The metadata service is composed of two modules: the membership module and the SDN controller. The membership module monitors storage nodes via heartbeats and detects membership changes (joins and failures), while the SDN controller controls the OpenFlow switches and updates the forwarding tables on membership changes. The SDN controller implements a layer 3 learning switch; it learns which storage node is connected to which switch port and uses this information to build unicast and multicasting forwarding rules.

Storage nodes maintain partial membership information related to the ring partitions of which they are part. Every node only knows the secondary replicas for the partition it is the primary replica for, and knows the primary replicas of every partition it is serving as a secondary replica; resulting in only *O(R)* information maintained at every node where *R* is the replication level.

When a node fails, the metadata service selects a handoff node to serve in lieu of the failing node (we detail the fault tolerance mechanism later). The metadata service updates the switch forwarding rules to correctly route requests destined to the failed node to the selected handoff node. The metadata service also informs the affected replicas of the membership change.

On a node join, the metadata service selects which ring partitions the new node will serve as a primary or secondary. Similar to handling failures, the metadata service updates the switch and informs the affected replicas of the change.

This membership maintenance design is scalable in terms of number of storage nodes. The membership service need to maintain switches forwarding tables which requires *O(S)*, where S is the number of switches in the platform, and *O(R)* messages to inform the affected replicas of the membership change. Note that each storage node only knows about the replicas it shares data with (which is *O(R)* of nodes). *R*, the replication level (typically 3 or 5), is independent of the total number of nodes.

The metadata server is logically centralized. While our system prototype uses a central metadata service, it can adopt known techniques for building a highly reliable distributed metadata services, including partitioning the key space among metadata service replicas, or having a hot standby replica. We are currently exploring the latter approach. Three workload characteristics make having a metadata hot standby: the stored metadata is small, changes to metadata are infrequent, and the load on the metadata service is low as it is mainly invoked on node or network failures.

4.2 Replication

Storage systems should not lose data when a node fails. The main data reliability approach adopted by the majority of NOOB storage systems is replication [13, 14, 15, 16, 17, 36] (with the other popular technique being erasure coding).

Challenge. On a put request, a single node (known as the primary replica or the coordinator) replicates the new object on *R-1* storage nodes through *R-1* unicast TCP connections, enabling the system to tolerate *R-1* replica failures without losing data.

This approach, in principle, is network non-optimal as the same data will traverse some links multiple times, especially those close to the node replicating the object. Further, this approach creates a high load on the node replicating the object as it needs to send/receive *R* copies of the data on every put.

To alleviate the load on the replicating node Renesse et. al. proposed chain replication [37]. In chain replication, nodes are organized in chains, and each node replicates the new object to the next node in the chain until the required number of replicas is created. While this approach may distribute the replication load across the nodes, it significantly increases the operation latency, and is equally network non-optimal, as it generates an equivalent amount of network traffic.

NICE Design. NICE builds network- and storage-optimal replication mechanism by leveraging network-level multicasting. The consistency mechanism discussed next requires to precisely identify and control which nodes are part of a given multicast group. While one may consider using traditional IP-multicasting, the fact that it requires every node to separately join/leave any multicast group makes it significantly harder (if not impossible) to build and maintain hundreds of multicast groups in face of node join and failure and to *precisely* identify when a particular multicast group has converged. OpenFlow helps solve these issues by allowing direct and centralized control of all groups.

NICE design divides storage nodes into overlapping replica sets; every physical node is, typically, a primary replica in one replica set and a secondary replica in *R-1* other sets.

To realize single-hop replication, NICE storage follows the virtual-storage approach discussed earlier. The client has two virtual rings: a unicast ring (discussed in the previous subsection) and a multicast ring. Each ring uses a separate IP address range (e.g., 10.10.0.0/16 for the unicast vring, and 10.11.0.0/16 for the multicast vring). As the name indicates, messages sent to an address in the multicast ring are multicasted to all replicas of an object, while the messages using the unicast ring are sent to one replica (the primary replica unless load balancing is used). The multicast ring is only used to send the put request and data.

Similar to the unicast vring, the multicast vring is divided into subgroups with each subgroup mapped to a replication set. For any packet targeting a virtual multicast address, the switch will *modify* the destination IP address to be the IP multicast address of the target replication set, and *forward* the packet to all the switch ports of the target replicas.

The proposed replication mechanism is optimal: first, it uses a single hop to route the put request; second, it uses optimal network paths for data replication (considering data center tree topology, the optimal path is equivalent to link-layer multicasting paths); third, it offloads the replication overhead from the primary replica to the network switch, achieving high performance and scalability. This approach is also optimal in terms of storage node load as each storage node only receives the data once. Finally, this replication approach is load balanced by design; the primary and secondary replicas send/receive almost an equal amount of data.

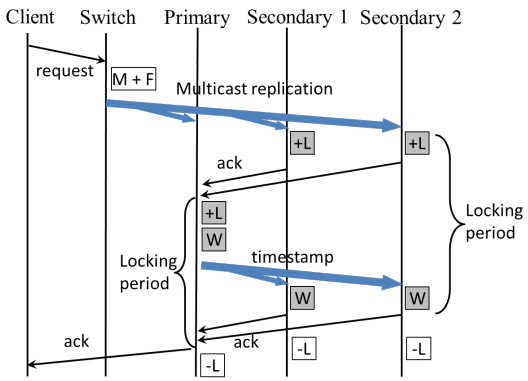


Figure 4. Consistency Mechanism. Timeline of the message sent in put operation in NICE storage. The switch performs modify and forward (M+F) for client packets to map the virtual address to the multicast group. (+L) is when a node logs the operation. (-L) is when the log entry is deleted. (W) is when the node writes the new object to the persistent storage. Gray boxes denote forced writes, and bold arrows denote multicasting. Object locks are only maintained in memory.

4.3 Consistency Mechanism

Sequentially consistent storage systems should guarantee data consistency across replicas, even when nodes fail or are disconnected and later join back with inconsistent data.

NOOB consistency protocols either face the possibility of blocking on node failure, require getting the object from the majority of nodes to resolve data inconsistency, or send all requests to a single node.

NICE proposes a consistency-aware fault tolerance mechanism. Here we demonstrate how NICE uses this mechanism to improve 2PC fault tolerance. The NICE-2PC mechanism (shown in Figure 4) follows the 2PC protocol design with two main differences. First, it leverages multicast-based replication to offload replication to the network, leading to load balanced and efficiently replication. Second, it improves the 2PC fault tolerance without requiring quorum-like protocols.

During the put operation, the client request is multicasted by the switch to all of the replicas. Upon receiving a complete object, the secondary replicas lock the object, log the operation, and acknowledge the operation to the primary replica. The primary replica, upon receiving an acknowledgment from all secondary replicas, generates a time stamp and multicasts the time stamp to all replicas. The timestamp contains the following quadruplet: primary address, primary timestamp, client address, and client timestamp. The timestamp creates an order between put operations to the same object, even between retrials of the put operation by the same client. The secondary replicas store the object to persistent storage following the timestamps order, release the lock, and acknowledge the end of the operation to the primary replica, which in turn acknowledges the operation to the client. We detail the fault tolerance mechanism next.

Get operations can be served by any replica. To avoid inconsistency, replicas lock operation during the put operation and only release the lock when the primary informs all nodes that the put operation completed successfully. Figure 4 show the blocking period in which read requests are queued until the concurrent put operation completes.

4.4 Fault Tolerance

Failure Model. NICE follows the failure model assumed by current NOOB systems: node failures are assumed to be transient, with permanent failures being handled by administrator intervention [13, 14, 16]. At the end of this section we discuss the procedure for adding and removing nodes from NICE. Consequently, when a node fails or is disconnected, the system does not automatically replicate the objects stored on that node, as these objects are still durably fully replicated.

Failure Detection. NICE adopts two techniques for detecting node failure: heartbeats and notifications from other nodes. The metadata service will declare the node failed if it misses three heartbeats from the node, or if a node reports to the metadata service that another node is irresponsive (e.g., if a node time-outs twice while waiting for a reply from a particular node in the 2PC protocol). Node failure causes two main problems: First, when a failing node recovers/rejoins, it often contains old (inconsistent) versions of the objects, if any of the stored objects have changed while the node was offline/disconnected. Second, newly stored objects will be under-replicated. Next we discuss how we handle these problems.

Failure Hiding. To handle the inconsistency problem of the failing nodes, on failure detection, the metadata service removes the failing node from the switch unicast and multicast vring mappings and informs the affected replicas. This effectively renders the node non-existent from the client point of view. When the node recovers, the switch mappings are updated only after the node is deemed consistent, as we will see next.

Maintaining Replication Level during Temporary Failures. When a node failure is detected the metadata service selects a handoff node to serve as a secondary replica in the hash region of the failing node [16]. Any storage node in the system that is not already part of the effected replication set can serve as a handoff node. The handoff node temporarily serves the object range until the failing node comes back. To simplify recovery, the handoff node stores the newly stored objects in a separate directory. If the handoff node receives a get request for an old object that it does not have, the handoff node will forward the request to the primary replicas. After selecting the handoff node, the metadata service updates the switch forwarding tables for both virtual rings and informs the affected replicas. When the original node comes back, it will discover the handoff node through contacting the metadata service and retrieve all the new objects. Primary node failure is discussed below. The system can handle multiple failures as long as at least one node in every region is an original node (not a handoff node) in the region.

Node Recovery. When a node recovers from failure, it contacts the metadata service to rejoin the system. Rejoining the system takes three steps: First, the metadata service adds the rejoining node to the multicast vring mapping, and the node will start receiving put requests. Second, the recovering node contacts the primary node to get all updates received during its downtime. Finally, the node informs the metadata service that it has consistent data. The metadata service will add the node to the unicast vring mapping, making the node available to get requests, and inform the affected replicas.

Failures during Put Operation. If a node fails during a put operation the operation will fail and the client will retry.

If a secondary node fails during a put operation (i.e., before sending the last ack to the primary replica in Figure 4), the primary node will detect the failure through missing either of the two ack messages from the node. The primary node will abort the operation and inform the client. The primary node will also inform the metadata service of the failure, starting the process for hiding the failure as detailed above.

If the primary node fails before sending the final acknowledgment to the client, the client will time-out and retry the operation. If the primary node fails before sending the "timestamp" message in the 2PC protocol in Figure 4, the secondary nodes will detect the failure by timing out and will inform the metadata service starting the failure-handling process detailed above. When a primary node fails, the metadata service selects one of the secondary nodes to act as a primary node. The new primary will contact the secondary nodes to identify all the objects that are locked on any secondary node. If an object is locked on any node, this means that node did not receive the timestamp message from the old primary. For locked objects, the primary does the following: if the object is committed on any secondary node, then this means the object was committed by the old primary. The primary will commit and unlock the object. If an object is locked on all secondary nodes, then the new primary will abort the operation. In case of a complete cluster failure, in which all in-memory locks are lost, the persistent logs on the nodes will identify the latest put operations. The new primary will check them all using the rules above.

Ring Re-Configuration. Occasionally, administrators may need to reconfigure the system to add or remove nodes. To permanently remove a node, the administrator updates the system configuration and informs the metadata of the node removal. The metadata removes all the forwarding rules related to the removed node, and updates all the effected replicas of the membership change. The metadata service handles adding a new node in a similar way to a re-joining a node after a temporary failure. The metadata updates the forwarding rules to add the new node to put vring and informs the other replicas of the change in the membership. Then, the new node contacts the primary replica to retrieve all keys stored in the hash range. Once the new node has consistent data the metadata service adds the node to the get vring making it available for serving get requests.

4.5 Load Balancing

While consistent hashing distributes the objects evenly across storage nodes, objects' popularity rarely follows a uniform distribution, leading to a skewed distribution in which a subset of objects is highly popular [38, 39]. In this case, storage systems use load balancing to distribute the get/put load on all the replicas of a given object.

Challenge. Current systems deploy a load-balancing node as a gateway to forward client requests using the ROG or RAG approach (§2). This deployment adds additional latency and requires provisioning load-balancers to avoid creating a system bottleneck. Latency-sensitive systems eschew load balancing and adopt the primary-backup design [26, 40]. Alternatively, if a weaker consistency is an option, a client-side load balancing can be adopted (e.g., the client can randomly pick one of the replicas).

NICE Design. The NICE metadata service implements a workload-informed consistency- and replica-aware load balancer. Unlike the NOOB storage design, our multicast-based put operations are load balanced by design; consequently, our load-balancing technique focuses only on get requests. While previous effort explored SDN-based load balancing [41, 42] our approach advances the previous approaches by using the storage metadata to build consistency- and replica-aware load balancer.

To perform workload-informed load-balancing, the metadata service collects periodic workload statistics. The statistics include the range of client IP addresses accessing each partition, and the size of the request queue. The statistics are piggybacked on the periodic heartbeats nodes send to the metadata service.

The metadata service divides the client address space into R divisions, such that each division size is a power of 2. Requests coming from each division will be forwarded to a different replica. The metadata service alters the switch forwarding rules to match both the packet source and destination IP addresses. The destination IP determines which physical ring partition the request is targeting, while the source IP determines which replica to forward the request to. For requests coming from IP addresses that are not covered by these divisions, the metadata service forwards them to the primary replica. When an administrator adds a new node to a replica set the metadata server reparations the client address space to utilize the new replica for get requests.

Compared with NOOB load balancing, NICE builds an in-network load balancing without increasing the latency or deploying extra resources.

This approach increases the number of forwarding entries per partition of the unicast vring from 1 to R entries, each forwarding a subset of client requests to one of the replicas.

This load balancing approach is coarse grained as it assigns clients to replicas without accounting for differences in client request rate. A few clients with high request rate can imbalance the load across replicas. In Section 4.5 we present a solution to this shortcoming.

4.6 Switch Scalability

The proposed approach can scale to support large storage systems. Without load balancing, each physical partition requires one entry in the switch forwarding table for the unicast vring mapping and one entry for the multicast vring mapping. This leads to a total of $2N$ entries in the forwarding table. Where N is the number of storage nodes. If load balancing is enabled, it uses R entries per partition (Where R is the replication level), leading to a total of $(R + 1)N$ entries. Current switches include forwarding tables that can support 128K forwarding rules or more which makes them capable of supporting storage systems with 64K storage nodes without load balancing. With load balancing enabled and with a replication level of 3 they can support up to 32K storage nodes.

4.6 Leveraging Programmable Switches

To make NICE design deployable on current commodity switches, we constrained the design to only use the capabilities of current OpenFlow standard. Unfortunately, OpenFlow limitations restricted some design decisions in NICE. In particular, NICE load balancing design is brittle in face of skewed workloads or heterogeneous infrastructure. The NICE load balancing technique (Section 4.5) partitions the clients into R groups and assigns a subset of clients to

each replica. This approach is not efficient if clients vary in their level of activity. If a small set of clients generate most of the load, a typical scenario in many applications (e.g., active tweeter users), the load will be skewed across replicas. Similarly, if the replicas are deployed on heterogeneous hardware, assigning equal load on all replicas is inefficient.

To address this challenge we implemented a flexible load balancer using the P4 programming language. The load balancer implements weighted load balancing techniques in which the controller assigns weights to replicas and configures the switch to distributed the load following these weights.

Following this approach, we changed the NICE design to use the P4 capabilities. The new design is identical to the OpenFlow design with the exception of leveraging the P4 capabilities to implement the weighted load balancing technique.

P4-based Load Balancing Design. Figure 5 shows the design of the load balancing stage. Every packet will be processed using the action in the load balancing table. The table has a single entry and a single action. The action randomly selects a number. The selected number is then used as an index to the IP addresses table.

To implemented the weighted load balancing technique, replicas report the length of the request queue in a periodic heartbeat. Every second, the controller calculates the average queue length for each replica and assigns proportional weights to each replica. The leader updates the list of random number in the load balancing table to reflect these weights. For instance, if replica 1 should receive double the load of any other replica, the action in figure 5 will be rand(1, 1, 2, 3), doubling the chance replica 1 is selected.

5 IMPLEMENTATION DETAILS

We implemented the NICEKV prototype following the NICE design. NICEKV is implemented in 14K lines of C++ code. The controller is implemented using 1K lines of python using the Ryu framework [43].

The rest of the section discusses implementation details of the network centric operations, and summarizes our experience with the state-of-the-art switches.

Clients. NICEKV is accessed through a client library with a simple interface for read, write, and delete operations. Read (get) and write (put) operations read or write entire objects. The library computes the hash of the requested key and maps it to the virtual IP address, then it will send the request to that virtual IP address.

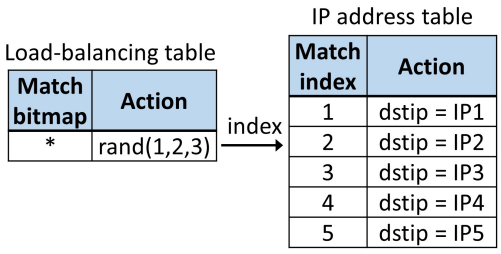


Figure 5. Logical view of the load balancing logic. The load balancing entry generates an index of the selected destination's IP address. Using the index, the IP address table sets the destination's IP address.

Mapping Service. The SDN controller implements a layer 3 learning switch. If the controller receives a packet destined to a not-yet-seen IP address, the controller will check if the address is a vnode address and update the switch to map the address to its physical counterpart, else the controller will buffer the packet and broadcast an ARP request for the unknown address. On receiving an ARP reply, the controller will update the forwarding tables and forward the buffered packets. The controller keeps a list of recently ARPed addressed to avoid flooding the network with ARP requests.

Request Routing. We use UDP to send client requests and TCP for all other communications, i.e., the client sends the put/get request to the vnode IP address using UDP and waits for the reply on a client-side TCP socket. This design decision allows mapping multiple vnode addresses to a single physical address without worrying about handling the reverse mapping required for TCP, i.e., mapping the physical node address to multiple vnodes. Further, UDP is required for IP multicasting.

Replication. For large objects, replication requires a *reliable* transport for data dissemination. NICEKV builds a simple reliable UDP-based multicast transport layer that uses primitive flow and congestion control techniques. Data is divided into multiple chunks, each less than a single network MTU (1400 bytes). The protocol uses NACKs to inform the client of missing packets, and the client sends the missing packets using a unicast connection. ACKs are used for flow control.

We implemented a version of the reliable multicast protocol for quorum protocols. We optimized the quorum implementation by pushing the quorum design down to the multicast transport layer. To this end, we designed a reliable any-k multicasting protocol. For flow control, the protocol tracks a window of transmitted packets and advances the window when any k of the recipients acknowledges receiving the packets. The protocol returns when any k of the nodes fully receives the data. After returning, the protocol keeps supporting straggling nodes until they finish or timeout.

P4 implementation. We implemented NICEKV-P4, a version of NICEKV with a P4 network protocol implementation. The switch data plane is written in P4 v14 [21, 44] and is compiled for Barefoot's Tofino ASIC [28], with Barefoot's P4Studio software suite [45]. Our P4 code uses less than 5% of the on-chip memory available in the Tofino ASIC, leaving ample resources to support other switch functionalities.

5.1 Deployment Experience

We experimented with three testbeds that are provisioned with three models of switches. Unfortunately, we found that the current switches lag in terms of the supported OpenFlow features. All switches supported only a subset of the OpenFlow standard. Efficiently modifying packet headers, in particular, was rarely supported. Only one switch supported this feature, but in software, resulting in three orders of magnitude slower switching speed if the switch is tasked with modifying a field in the header.

The CloudLab [46] Utah cluster, which we use, uses Comware switches which supports a subset of OpenFlow features; in particular, it supports forwarding the packets to multicast addresses but does not support modifying the packet IP destination address.

Modifying the packet IP destination addresses is necessary for mapping virtual addresses to physical addresses.

To address this challenge, we deployed Open vSwitch [47] on every client machine. Open vSwitch is a software-based OpenFlow-enabled virtual switch. Further, we extended the NICEKV SDN controller to control multiple switches (i.e., multiple Open vSwitches and a single hardware switch). The controller installs the rules to modify packet headers (mapping virtual to physical addresses) on the client side Open vSwitches, and installs forwarding and multicasting rules on the hardware switch. Our evaluation shows that our new deployment leads to less than 4% performance loss compared to the same deployment without using Open vSwitch.

6 EVALUATION

We evaluated NICE using synthetic as well as real world benchmarks using the Yahoo YCSB benchmark [39]. We empirically compare NICE with three object storage systems: Ceph [12], OpenStack Swift [13], and NOOB, our in-house key-value systems. We choose Ceph and Swift as these are production-quality widely-used NOOB storage systems, the NOOB prototype allows us to compare NICE with range of NOOB designs and configurations. We compared the performance of quorum based design. The results of those experiments are available here [48].

Ceph. Ceph adopts a primary-backup approach in which all client put and get operations are received and processed by the primary replica. Clients send their put requests directly to the primary, which replicates the data, then replies to the clients. The primary does not serve concurrent put or get requests until the current put operation completes.

OpenStack Swift. Swift adopts a proxy-based design; client put and get requests are sent to a proxy node, that sends the request to the responsible storage node, then the proxy replies to the client. The proxy node is on the data path for both put and get operations.

NOOB prototype is a highly configurable storage system that implements three common access mechanisms, and two replication techniques. NOOB facilitates comparing NICE with wider design choices. The NOOB system implements the three common access mechanisms: RAC with client side caching, RAG with a replica-aware load balancer, and ROG with a randomized load balancer. Furthermore, NOOB prototype implements two replication/consistency mechanisms: two-phase commit (2PC) and primary-backup. The NOOB prototype allows us to compare NICE with range of NOOB designs and configurations.

Platform. We use a cluster of 30 nodes on the CloudLab [46] Utah site. Each node has an 8-core ARMv8 2.4 GHz processor, 64GB memory, 120GB SATA3 Micron SSD disk and 1 Gbps NIC. The nodes are connected to an OpenFlow enabled switch that supports OpenFlow 1.3.1. While the evaluation uses a single hardware switch the controlled switching topology (including Open vSwitches software switches) is much more complex. Further, NICE can readily support multi-switch platforms, as the controller will install the same rules on all participating switches.

Deployment Configuration. Unless otherwise specified, we deploy the systems on 16 nodes (one mapping node and 15 storage nodes), 14 nodes for clients and load balancers, and configure the system with replication level of 3.

6.1 Request Routing Evaluation

We evaluate the performance of request routing in four systems: NICE, Ceph, Swift, and NOOB storage prototype. For NOOB storage we evaluate three configurations: ROG, RAG, and RAC. The workload used consists of get-only requests issued by a single client. The object size varies from 4 bytes to 1MB. Figure 6 shows the average of 1000 get operations.

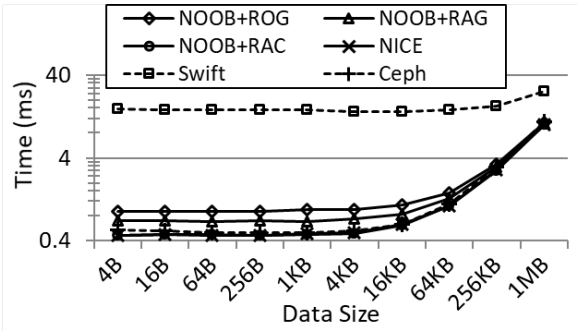


Figure 6. Request Routing Performance. The average time of the get operation. Note the log scaled y-axis. The lines for NICE, Ceph, and NOOB-RAC overlap.

Figure 6 compares the performance of the four systems. Systems that use a single-hop request routing (i.e., direct access from client to the primary replica of an object), including NICE, Ceph, and NOOB+RAC, achieve the lowest latency and have comparable performance. For object sizes less than 64KBs NICE, Ceph, and NOOB+RAC systems achieve 1.5× performance improvement compared to NOOB+RAG, and 2× improvement compared to NOOB+ROG. This improvement is due to eliminating the request routing delay imposed by RAC and RAG designs. Swift achieves the lowest performance: NICE achieves over 30× improvement compared with Swift. The benefits are not as pronounced with large data sizes, as transfer time dominates. Nevertheless, Swift performance lags (by over 2.5×) even with large data sizes. The main reason for Swift low performance is that Swift completely hides storage nodes, and clients only interact with the proxy nodes. Consequently, to serve a get request to a client, instead of sending the get response directly from a storage node to the client, Swift transfers data from storage nodes, to proxy nodes, then to clients, which introduced additional latency, increases system load, and reduced throughput.

6.2 Replication Evaluation

We evaluate the performance of the replication mechanism and compare it across the four systems: NICE, Ceph, Swift, and NOOB with primary-backup design. We evaluate the three request routing mechanisms in NOOB: ROG, RAG, and RAC. We compare these systems in terms of replication time, network load, and ratio of load of the primary replica to the secondary replicas. The workload used consists of put-only requests issued by a single client. The object size varies from 4 bytes to 1MB. Figure 6 shows the average of 1000 get operations.

Replication time. Figure 7 compares the replication time of all evaluated systems. The results show that NICE achieves the best performance across object sizes. NICE achieves: up to 4.3× compared to NOOB+ROG, up to 3.4× compared to NOOB+RAG, up to 2.6× compared to NOOB+RAC, up to 2.6× compared to Ceph, and over 40× compared to Swift which uses the Replica-

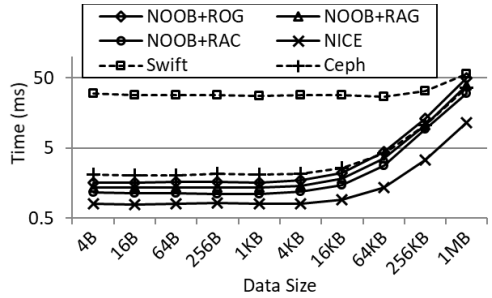


Figure 7. Replication Performance. The average time of the put operation. Note the log scaled y-axis

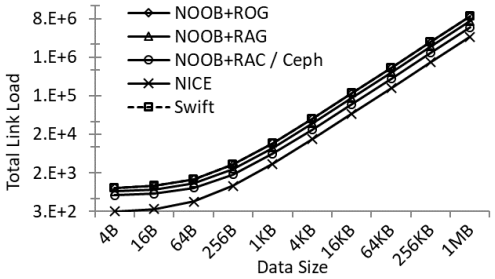


Figure 8. Network Link Load. The total network link load of the put operation.

aware proxy design. NICE achieves this significant performance improvement by using optimal multicast-based replication and through eliminating request routing overhead using single-hop routing.

Network load. We evaluated the network load generated by the put operation on all tested systems (Figure 8). We measure the network load as the total amount of data transferred on every link in the network (i.e., a 1KB bject traversing two links count as 2KB of network load). NICE significantly reduces the network load compared to the other systems. This improvement holds regardless of object size. NICE generates between 1.7× to 3.5× less network load compared to the other systems.

Storage Load Ratio. We evaluate the load imbalance between the storage nodes as the ratio between the amount of data processed (sent or received) by the primary replica to the amount of data processed by the secondary replicas (Figure 9). Figure 9 shows that NICE load balances the workload between the primary and secondary with both achieving the optimal load of 1 (i.e., receiving the object once only). Ceph and all NOOB configurations impose 3× more work on the primary than on the secondary (this load imbalance is proportional to the replication level). In Swift, the proxy node has 4× more load than any other replica.

6.3 Consistency Mechanism Evaluation

Storage systems may replicate an object to meet high demand. We evaluate the put operation efficacy while varying the replication level. We evaluate NICE, Ceph, Swift, and the best configuration for NOOB system, namely NOOB+RAC. We evaluate two configurations for NOOB+RAC: primary-backup and 2PC. The evaluation uses small 4-byte objects and large 1MB objects.

With 4-byte objects (Figure 10) NICE achieves the highest performance: up to 1.3× better performance than NOOB-2PC. NICE achieves comparable performance to NOOB primary-backup replication. Although NICE has an extra phase of communication

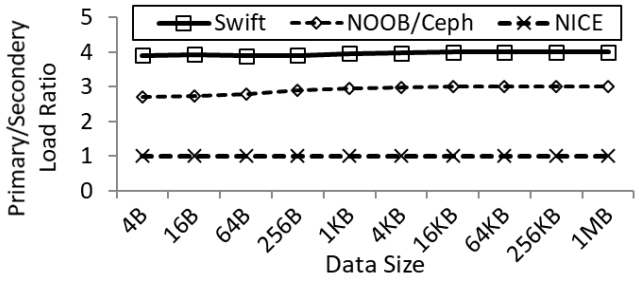


Figure 9. Storage Load Ratio. The ratio of the primary replica (or proxy node in case of Swift) to secondary replica load in terms of amount of data sent/received during the put operation.

compared to the primary only design, its use multicast-based replication reduces the data transfer time and eliminates the overhead of creating 8 TCP connections. We note that the performance of all systems degrades with higher replication levels, due to the increased overhead of the consistency protocol that dominates small object performance. The primary-backup design achieves better performance than NOOB-2PC due to 2PC protocol overheads.

Figure 11 shows the put operation time with 1MB objects. NICE achieves up to 5.5× better performance than NOOB systems. The primary-backup and 2PC achieve comparable performance since, with large objects; performance is dominated by replication cost. While NOOB performance degrades considerably: by 7× when

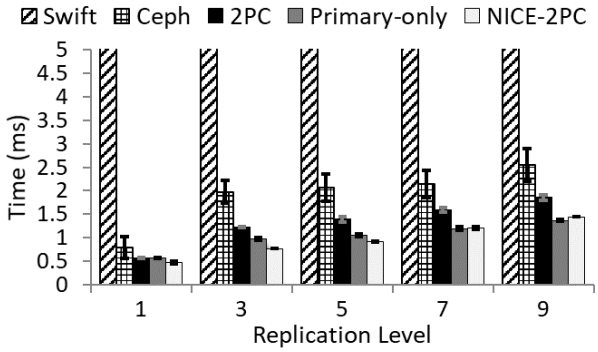


Figure 10. Consistency Mechanism Performance with 4-byte objects while varying the replication level. Error bars represent standard deviation. For clarity we truncate the figure to 5ms. Switft completes the workload in 24ms with 1 replica and in 47ms with 9 replicas.

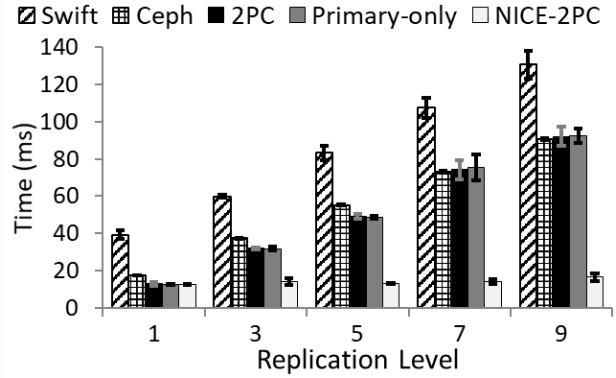


Figure 11. Consistency Mechanism Performance with 1MB objects while varying the replication level. Error bars represent standard deviation.

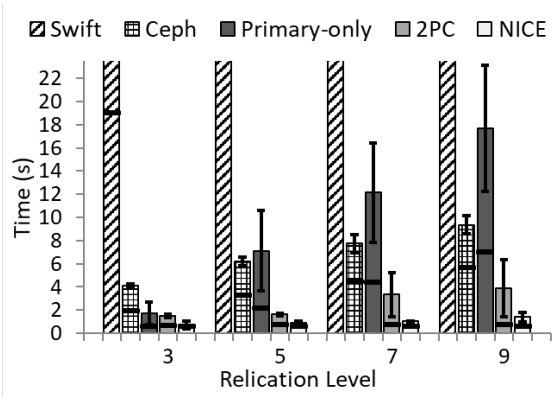


Figure 12. Load balancing evaluation with 4-byte objects. The systems performance under the load balancing workload while varying the replication level and number of clients. Bold markers show the performance of the get-only workload. Error bars represent standard deviation. For clarity, we truncate Swift bars, Swift finishes in 35ms with 3 replicas, and in 70ms with 9 replicas.

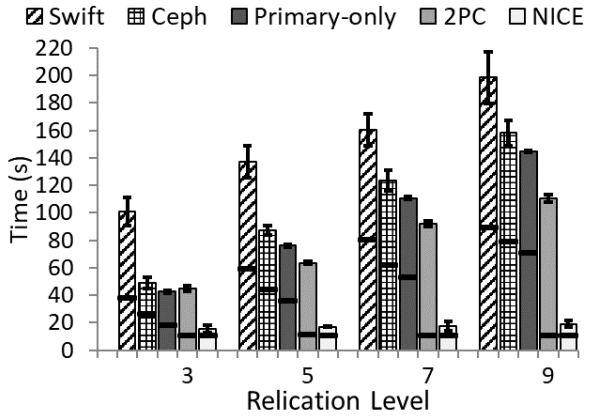


Figure 13. Load balancing evaluation with 1MB objects. The systems performance under the load balancing workload while varying the replication level and number of clients. Bold markers show the performance of the get-only workload. Error bars represent standard deviation.

increasing the replication from 1 to 9, NICE performance degrades slightly when increasing the replication level (by 17% when increasing the replication from 1 to 9).

NICE achieves up to 5× and 23× performance gain with 4-byte objects and up 5× and 6.5× with 1MB objects compared to Ceph and Swift, respectively. This is mainly due to the use of inefficient replication and due to the added overhead of the proxy nodes on the data path in Swift.

6.4 Load Balancing

To evaluate systems ability to load balance requests across replicas of the same object we designed a weak scaling experiment: we test the systems while increasing the number of replicas and proportionally increase the load (i.e., the number of clients). The experiment measures systems ability to efficiency utilize the added resources to serve proportionally equivalent load. We evaluate the performance of NICE, Ceph, Swift, and two NOOB storage configurations: primary-backup and 2PC. The experiment measures the systems performance when serving highly-popular frequently-updated objects. In each configuration 1 client puts a shared object 1000 times, while *R-1* clients each gets the shared object 1000 times.

We ran the experiment with 4-byte (Figure 13) and 1MB (Figure 14) objects. The results show that NICE achieves higher performance than NOOB, Ceph and Swift under all replication levels. NICE achieves up to 7.5× better than the primary-backup configuration, and up to 5.5× better than the 2PC configuration in both object sizes.

To understand the impact of contention between put and get requests we compare the results to get-only workload. The dark line markers on the bars in Figure 12 and Figure 13 show the performance of *R-1* clients issuing get operations without any put operation. The figures show that NICE and 2PC are able to load balance the get requests across replicas, while the primary-backup design performance degrades with the increased workload as no load balancing is used. Comparing the black marker to the top of the bar shows the added overhead due to the contention of the put and get requests. The figure shows that data consistency mechanism adds significant overhead to NOOB systems.

Figure 12 and Figure 13 show that NOOB storage system performance degrades considerably when increasing the replication level and the number of clients. NOOB primary-backup performance degrades by 10× with small objects and 3.5× with 1MB object, and the 2PC configuration degrades by 2.6× with both sizes. This indicates that NOOB design is not weakly scalable and generates high overhead under heavy demand. NOOB is unable to meet the increasing demand despite the proportional increase in the allocated resources. Significant replication costs (dominant in large objects) and consistency-protocol overhead (dominant in small object) are the reason why. NICE storage performance degrades slightly when increasing the replication level and the number of clients (only by 20% with 1MB objects and by 80% with 4-byte objects).

6.5 Fault Tolerance Evaluation

This experiment demonstrates the system fault tolerance mechanism. The experiment fails and recovers a replica while the system is consistently being accessed by three clients. The clients generate a continues stream of put and get requests with a ratio of 20/80 of put/get requests and with a key size of 1KB. All objects are in the same partition.

Figure 14 shows the number of put and get requests served per second. At the 30s mark, a secondary replica (node 2) fails. The following put operation will fail and the primary node will detect the replica failure. The primary node will inform the metadata service. The metadata service executes the fault tolerance steps: it removes the failed node from the switch mappings and adds the handoff node to the replica set. This process takes less than 2 seconds during this process the partition is unavailable for put operations (Figure 14 second 31). Client put requests during this period will fail and the client will retry after waiting for 2 seconds, in which case the operations will succeed. We are working on shortening this down time through allowing put operations to succeed even if one node fails (i.e., having R-1 replicas) and by creating, in the background, one more replica on the handoff node when it joins the replica set.

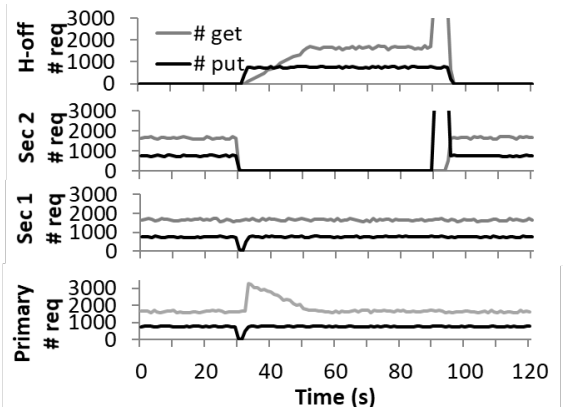


Figure 14. Fault Tolerance Evaluation. Secondary node 2 fails at 30s mark, triggering the fault tolerance mechanism, and 90s the node recovers, retrieves the missed objects from the handoff node, and starts serving client requests.

For get operations, the client selects, in a uniform random fashion, one of the recently put objects to get. When the handoff node starts serving client requests (second 31), it does not have any of the requested objects. In this case, it forwards all get requests to the primary replica. As more objects are stored at the handoff node less get requests are forwarded to the primary node.

At 90s mark, the failed node joins back, and starts retrieving the objects it missed. This is represented by the spike in put requests (and get requests at the handoff node). At the 95 second mark the returning node completes its recovery and has a consistent set of objects, the metadata service adds the node to the unicast switch mapping and removes the handoff node.

6.6 Real Workload Evaluation

We evaluate NICE and NOOB systems under real application workload generated using the Yahoo cloud serving benchmark (YCSB) [39]. We use two workloads from the YCSB benchmark suit: a read-only workload (YCSB-C) and the read-modify-write workload (YCSB-F) with 50% put requests. As in the majority of the Yahoo workloads, these two have a zipf popularity distribution.

We evaluate NICE and NOOB with primary-backup and 2PC configuration. The workload used consists of 10 clients each generating 20K requests generated using the YCSB workload discussed earlier. We use the default YCSB object size of 1KB.

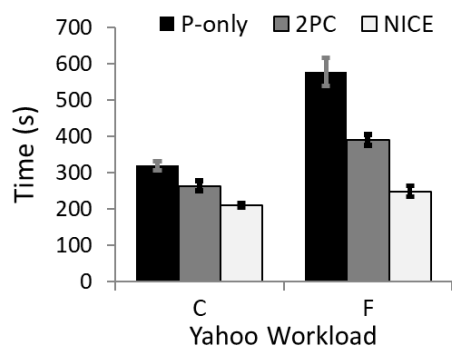


Figure 15. Yahoo Benchmark Evaluation. The three systems performance under two Yahoo benchmarks: read-only (C), and read-modify-write (F). Error bars represent standard deviation.

The experiment results (Figure 15) shows that NICE achieves the best performance under the two workloads. NICE achieves 1.6× and 2.3× better than primary-backup configuration under workload C and F, respectively. This improvement is due to the lack of load balancing in the primary-backup configuration. Compared with 2PC configuration, NICE achieves 1.25× and 1.5× better performance under workload C and F, respectively. 2PC configurations lags NICE due to the added latency by the load balancer (using the RAG request routing) and consistency-protocol overhead.

6.7 Evaluation with the Programmable Switch

We empirically compare two load balancing approaches: client partitioning (CP) which partitions the clients among replicas based on their IP address. This is the approach implemented in the OpenFlow-based NICEKV), and weighted replica (WR) in which the controller sets a weight for each replica and the switch assignments load to replicas proportional to it weight. This is the technique implemented in the P4-based implementation.

For these experiments we used a different cluster with a P4-programmable switch and 13 nodes. Each node has an Intel Xeon Silver 10-core CPU, 48GB of RAM, and 100Gbps Mellanox NIC. The nodes are connected to an Edgecore Wedge 100 ×32BF switch with 32 100Gbps ports. The switch has Barefoot's Tofino ASIC, which is P4 programmable. In all of our experiments, three machines ran the server code, while the other 10 machines generated the workload. Each client node is running 100 client threads. Each thread is generating read requests following the read-only YCSB workload C benchmark. The key size is 24 bytes and the value is 1KB.

To demonstrate the flexibility of the P4 based load balancing we compared the operation latency under two scenarios.

Scenario I: skewed client workload. A workload in which clients vary in the amount of requests they generate.

In this experiment the first 3 out of the 10 client nodes generate 50% of the requests. Figure 16 shows the latency of requests of the two approaches under this workload. NICEKV-WR achieves up to 50% lower latency than NICEKV-CP. This is mainly because NICEKV-CP partitions the clients across replicas based on their IP address, leading to all highly active clients being assigned to one replica. The selected replica received over 50% of the total load in the system while the other two replicas received less than 25% of the load each. The NICEKV-WR assigns equal weights to all replicas hence it uniformly distributes the client load across replicas leading to better load balancing and lower overall latency. Figure 16 shows that 50% of the requests experience significantly higher latency with CP compared to WR.

Scenario II: heterogeneous replicas. In this experiment we artificially slow the CPU of one of the replicas by 40% to emulate a platform with heterogeneous nodes.

In this experiment all clients generate the same amount of requests. Figure 17 shows the latency of requests of the two approaches under this workload. We notice for NICEKV-CP that 40% of requests experience significantly higher latency. NICEKV-WR balances the load among replicas proportional to their capabilities: it increases the load on the two capable nodes (hence a bit higher

latency for the bottom 60% of requests) and reduces the latency on the slow node (hence up to 50% lower latency for the top 40% of requests).

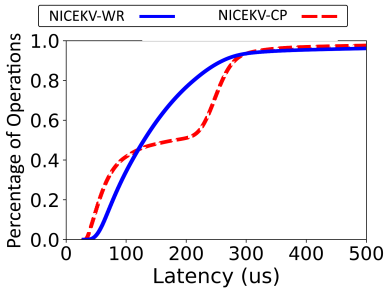


Figure 16. The latency CDF of NICEKV with client partitioning (CP) load and Weighted replicas (WR) load balancing under a skewed client workload.

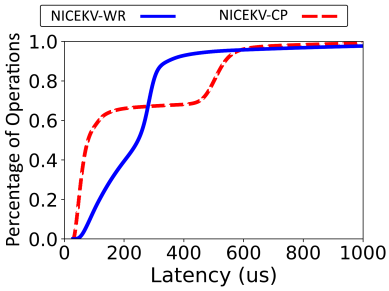


Figure 17. The latency CDF of NICEKV with client partitioning (CP) load and Weighted replicas (WR) load balancing with heterogeneous hardware.

7 OTHER RELATED WORK

Request Routing. Beehive [49] proposes a different approach for achieving, on average, single-hop request routing for special workloads: workloads with highly skewed power-law popularity distribution. Beehive replicates each object based on its popularity, with the extremely popular objects replicated on every node, hence accessible in a single-hop. Due to the network and storage overheads, this approach is only feasible for highly skewed workloads of infrequently updated objects.

SDN Optimized Systems. Recent research projects utilize SDN capabilities to provide load balancing [41, 42], access control [50], seamless VM migration [51], and to improve system security, virtualization and network efficiency [52]. These systems still use the network as a separate entity and use SDN to optimize its operations. Unlike current efforts, we co-design network and system operations and protocols to achieve significant benefits.

Recently, a number of projects explored techniques to leverage the new SDN capabilities. MOM [53], NOPaxos [54], Eris [55] build consistency protocols by relying on the network to order operations. SwitchKV [56] builds a key-value storage with a tier of caching nodes. SwitchKV uses the SDN-capability to optimize request routing for get requests from the cache. MBalancer [57] and Trajano et al. [58] leverage the SDN capabilities to build application aware load balancers. sRoute [59] uses SDN to optimize gather and scatter communication patterns in storage systems. Unlike these projects, we propose a new complete system architecture that co-designs network and storage support and optimizes a range of mechanisms including load balancing, replication, and consistency.

Leveraging Programmable Switches. Recently a number of projects started exploring techniques to leverage the capabilities of programmable switches to improve distributed systems. NetCache [60] implements a caching service in a single switch. The controller keeps track of the most popular objects and controls the cached objects in the switch. NetChain [61] optimizes vertical Paxos [62] by implementing chain replication on a chain of programmable switches. NetPaxos [63] considers moving the Paxos protocol to the network switches, such that one switch serves as a coordinator and other switches serve as replicas. The proposed approach requires implementing a substantial part of the protocol in switches and storing a potentially large protocol state. NetChain and NetPaxos are suitable for systems that store only a few megabytes of data (e.g., 8MB in the current NetChain prototype).

8 CONCLUSION AND FUTURE WORK

We present network-integrated cluster-efficient (NICE) storage, which co-designs storage logic and networking support to realize a more efficient, scalable, and reliable distributed storage. Our prototype evaluation shows that this approach can realize significant benefits: up to 7× performance improvement, substantial network-load reduction (up to 50%), and improved load balancing and scalability. While we focus the discussion on key-value storage systems, the proposed techniques for virtualization and consistency-aware fault tolerance are widely applicable.

Our future work will focus on two directions. First we plan to investigate building SDN-optimized storage systems that can support more complex key-value queries. Second, NICE design focused on improving replication-based storage system. The second popular reliability technique is the use of erasure coding. We plan to investigate techniques to accelerate storage systems using erasure coding.

ACKNOWLEDGMENT

We thank Ajay Bakre, Alvin Lam, and Emalayan Vairavanathan from NetApp Vancouver technical center (VTC) for their support and early feedback, Aaron Gember-Jacobson for his help with Openflow deployment issues, Thanumalayan S. Pillai for his help with the Yahoo benchmark experiment, and Robert Ricci and the CloudLab team for their support at CloudLab. This material was supported by funding from an NSERC Discovery grant, NSERC Engage grant, Canada Foundation for Innovation (CFI) grant, NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, as well as in-kind support from NetApp VTC, Canada. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSERC, NSF, or other institutions.

REFERENCES

[1] J.H. Saltzer, D.P. Reed, and D.D. Clark, *End-to-end arguments in system design.* ACM Transactions in Computer Systems, 1984. **2**(4): p. 277-288.
[2] *Amazon Elastic Compute Cloud (EC2).* [cited 2010; Available from: http://aws.amazon.com/ec2/.
[3] *Google app engine.* [cited 2015; Available from: https://appengine.google.com.
[4] *Microsoft Azure: Cloud computing platform and services.* [cited 2016; Available from: https://azure.microsoft.com/.

[5] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[6] *Spark lighting fast cluster computing*. [cited 2019; Available from: http://spark.apache.org/.

[7] B. Calder, J. Wang, A. Ogus, et al. *Windows azure storage: A highly available cloud storage service with strong consistency*. in *ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. *The Google File System*. in *SOSP'03*. 2003. Lake George, NY.

[9] J.H. Howard, M.L. Kazar, S.G. Menees, et al., *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems, 1988. **6**(1).

[10] K. Gupta, R. Jain, I. Koltsidas, et al., *GPFS-SNC: An enterprise storage framework for virtual-machine clouds* IBM Journal of Research and Development 2011.

[11] R. Sandberg, D. Goldberg, S. Kleiman, et al., *Design and Implementation of the Sun Network Filesystem*, in *Proc. Summer USENIX*. June 1985. p. 119--130.

[12] S. Weil, S.A. Brandt, E.L. Miller, et al. *Ceph: A Scalable, High-Performance Distributed File System*. in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*. 2006.

[13] *OpenStack Cloud Platform: OpenStack Swift*. [cited 2015; Available from: http://docs.openstack.org/developer/swift/overview_architecture.html.

[14] Basho. *Riak cloud storage*. [cited 2015; Available from: http://basho.com/riak-cloud-storage/.

[15] *Voldemort project*. [cited 2015; Available from: http://www.project-voldemort.com/voldemort/design.html.

[16] G. DeCandia, D. Hastorun, M. Jampani, et al. *Dynamo: Amazon's Highly Available Key-value Store*. in *SOSP07*. 2007.

[17] A. Lakshman and P. Malik, *Cassandra: A decentralized structured storage system.* SIGOPS Operating Systems Review, 2010. **44**(2): p. 35-40.

[18] I. Stoica, R. Morris, D. Karger, et al. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. in *SIGCOMM Conference*. 2001. ACM.

[19] *The Open Networking Foundation: Openflow switch specification. Version 1.5.0.* 2014.

[20] N. McKeown, T. Anderson, H. Balakrishnan, et al., *Openflow: Enabling innovation in campus networks.* SIGCOMM Computer Communication Review, 2008. **32**(2): p. 69-74.

[21] P. Bosshart, D. Daly, G. Gibb, et al., *P4: programming protocol-independent packet processors.* SIGCOMM Comput. Commun. Rev., 2014. **44**(3): p. 87-95.

[22] D.R. Karger, E. Lehman, F.T. Leighton, et al. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. in *Symposium on Theory of Computing*. 1997. ACM.

[23] A. Rowstron and P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. 2001. Heidelberg, Germany.

[24] A.S. Tanenbaum and M.V. Steen, *Distributed Systems: Principles and Paradigms*. 2 ed. 2006: Prentice Hall.

[25] *The Apache Cassandra Project*. 2012; Available from: http://cassandra.apache.org/.

[26] D. Ongaro, S.M. Rumble, R. Stutsman, et al., *Fast crash recovery in RAMCloud*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, ACM: Cascais, Portugal. p. 29-41.

[27] J. Technologies, *Network Protocols Handbook*. 2005: Javvin Technologies Inc.

[28] *Barefoot Tofino*. 2019; Available from: https://www.barefootnetworks.com/products/brief-tofino/.

[29] *Cavium / XPliant*. 2019; Available from: https://origin-www.marvell.com/documents/netpxrx94dcdhk8sksbp/.

[30] L. Lamport, *The part-time parliament.* ACM Transactions on Computer Systems (TOCS), 1998. **16** (2): p. 133-169.

[31] L. Lamport, *Paxos Made Simple.* ACM SIGACT News, 2001. **32**(4).

[32] D. Ongaro and J. Ousterhout. *In search of an understandable consensus algorithm*. in *USENIX Annual Technical Conference (USENIX ATC)*. 2014.

[33] *Mongodb*. Available from: https://www.mongodb.org/.

[34] *Postgresql*. 2019; Available from: http://www.postgresql.org/.

[35] M.K. Aguilera, A. Merchant, M. Shah, et al., *Sinfonia: a new paradigm for building scalable distributed systems*, in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, ACM: Stevenson, Washington, USA. p. 159-174.

[36] J. Ousterhout, A. Gopalan, A. Gupta, et al., *The RAMCloud Storage System.* ACM Transactions on Computer Systems, 2015. **33**(3): p. 7:1-7:55.

[37] R.v. Renesse and F.B. Schneider. *Chain replication for supporting high throughput and availability*. in *Symposium on Opearting Systems Design & Implementation (OSDI)*. 2004. San Francisco, CA.

[38] B. Atikoglu, Y. Xu, E. Frachtenberg, et al., *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*. 2012, ACM: London, England, UK. p. 53-64.

[39] B.F. Cooper, A. Silberstein, E. Tam, et al., *Benchmarking cloud serving systems with YCSB*, in *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, ACM: Indianapolis, Indiana, USA. p. 143-154.

[40] D. Terry, V. Prabhakaran, R. Kotla, et al., *Consistency-based service level agreements for cloud storage*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, ACM: Farminton, Pennsylvania. p. 309-324.

[41] N. Handigol, M. Flajslik, S. Seetharaman, et al. *Aster*x: Loadbalancing as a network primitive*. in *GENI Engineering Conference (Plenary)*. 2010.

[42] R. Wang, D. Butnariu, and J. Rexford. *Openflow-based server load balancing gone wild*. in *USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. 2011.

[43] *Ryu sdn framework*. [cited 2019; Available from: http://osrg.github.io/ryu/.

[44] *Programming protocol-independent packet processors (P4)*. [cited 2019; Available from: https://p4.org.

[45] *Barefoot P4 Studio*. [cited 2019; Available from: https://www.barefootnetworks.com/products/brief-p4-studio/.

[46] *Cloudlab*. [cited 2019; Available from: http://www.cloudlab.us/.

[47] *OpenvSwitch: Production quality, multilayer open virtual switch*. [cited 2019; Available from: http://openvswitch.org/.

[48] S. Al-Kiswany, S. Yang, A.C. Arpaci-Dusseau, et al. *NICE: Network-Integrated Cluster-Efficient Storage*. in *ACM International Symposium on High Performance Parallel and Distributed Computing (HPDC)*. 2017.

[49] V. Ramasubramanian and E.G. Sirer. *Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays*. in *NSDI*. 2004. San Francisco, CA.

[50] A.K. Nayak, A. Reimers, N. Feamster, et al. *Resonance: Dynamic access control for enterprise networks*. in *ACM Workshop on Research on Enterprise Networking (WREN)*. 2009.

[51] A.J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, et al. *Xvmotion: Unified virtual machine migration over long distance*. in *USENIX Annual Technical Conference (USENIX ATC)*. 2014.

[52] A. Lara, A. Kolasani, and B. Ramamurthy, *Network innovation using openflow: A survey.* IEEE Communications Society, 2014. **16**(1): p. 493 - 512.

[53] D.R.K. Ports, J. Li, V. Liu, et al. *Designing distributed systems using approximate synchrony in data center networks*. in *Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.

[54] J. Li, E. Michael, N.K. Sharma, et al. *Just say no to paxos overhead: replacing consensus with network ordering*. in *USENIX conference on Operating Systems Design and Implementation (OSDI)* 2016.

[55] J. Li, E. Michael, and D.R.K. Ports. *Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control*. in *Symposium on Operating Systems Principles (SOSP)*. 2017.

[56] X. Li, R. Sethi, M. Kaminsky, et al. *Be Fast, Cheap and in Control with SwitchKV*. in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* 2016.

[57] A. Bremler-Barr, D. Hay, I. Moyal, et al. *Load balancing memcached traffic using software defined networking*. in *IFIP Networking Conference*. 2017.

[58] A. Trajano and M. Fernandez. *Two-phase load balancing of In-Memory Key-Value Storages through NFV and SDN*. in *IEEE Symposium on Computers and Communication (ISCC)*. 2016.

[59] I. Stefanovici, B. Schroeder, G. O'Shea, et al. *sRoute: Treating the Storage Stack Like a Network*. in *USENIX File and Storage Technologies (FAST)*. 2016.

[60] X. Jin, X. Li, H. Zhang, et al. *NetCache: Balancing Key-Value Stores with Fast In-Network Caching*. in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 2017. Shanghai, China: ACM.

[61] X. Jin, X. Li, H. Zhang, et al. *Netchain: scale-free sub-RTT coordination*. in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2018. Renton, WA, USA: USENIX Association.

[62] L. Lamport, D. Malkhi, and L. Zhou. *Vertical paxos and primary-backup replication*. in *Proceedings of the ACM symposium on Principles of distributed computing*. 2009. Calgary, AB, Canada: ACM.

[63] H.T. Dang, D. Sciascia, M. Canini, et al. *NetPaxos: consensus at network speed*. in *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015. Santa Clara, California: ACM.