NICE: Network-Integrated Cluster-Efficient Storage

Samer Al-Kiswany*, Suli Yang✦, Andrea C. Arpaci-Dusseau✦, Remzi H. Arpaci-Dusseau✦
* University of Waterloo, alkiswany@uwaterloo.ca
✦ University of Wisconsin-Madison, {suli, dusseau, remzi}@cs.wisc.edu

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 to the current key-value 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.

KEYWORDS

Key-value storage, software-defined networks, co-design, distributed storage

1 INTRODUCTION

The end-to-end design principle [38] pervades the design of virtually every modern distributed system [1, 3, 4, 11, 17]. 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 [15, 20, 22, 24, 39, 45] and scalable key-value stores [6, 9, 12, 18, 26]. 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 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 [40], 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, 30] 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 enables building a new fault tolerance mechanism: consistency-aware fault tolerance. This mechanism simplifies building consistency protocols by making failed nodes, or nodes with inconsistent data, inaccessible.

To demonstrate the efficacy of the NICE approach, we design and implement a key-value storage prototype, NICEKV. Our

HPDC '17, June 26-30, 2017, Washington , DC, USA

ACM ISBN 978-1-4503-4699-3/17/06…$15.00
http://dx.doi.org/10.1145/3078597.3078612

empirical evaluation with synthetic and real workload benchmarks shows that the NICEKV prototype realizes significant performance gains compared to a broad set of NOOB storage configurations. Membership maintenance in NICEKV is highly scalable and eliminates the maintenance operations overhead. NICEKV request routing achieves single-hop routing without requiring extra resources. NICEKV replication is network and storage optimal (discussed in detail in section 4.2), effectively halving the network-generated load and reducing storage load by 3× to 9×, depending on replication level. The NICEKV two-phase commit consistency protocol uses the consistency-aware fault tolerance mechanism to tolerate failures without increasing operation overhead. NICEKV load balancing effectively spreads client requests across servers without deploying dedicated load-balancing machines. The combination of these optimizations is powerful; the NICEKV prototype can achieve up to 7× put/get performance improvement as compared to the traditional network oblivious approach.

The rest of this paper is organized as follows. In Section 2, we present an overview of the current NOOB storage design, and the recent advances in software-defined networks. We then present the proposed NICE approach in Section 3, detail the system design 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 network-oblivious: the network is only used as a point-to-point communication medium without storage system control over its operations. NOOB storage systems typically implement storage logic and protocols within end hosts; this approach is fundamentally inefficient as many core storage system operations are, in principle, network-level operations, e.g., replication or request routing.

Many NOOB storage systems adopt a design based on consistent hashing [25]. 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 [37] and Chord [40] 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 [6, 9, 12, 18, 26], 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 [6], or through an epidemic protocol entailing *O(log n)* steps and over *O(N)* messages [41].

Access Mechanism. To access the system, current systems adopt one of the following three techniques to route client requests to the node maintaining the object. The first technique, which we refer to as the Replica-Oblivious Gateway (ROG), uses a generic off-the-shelf load balancer that forwards client requests to a storage node selected in a random or a round-robin fashion. This approach is common in production systems [6, 9, 18, 26] due to its ease of deployment and use of existing load balancers. This approach imposes two extra hops for routing a request.

The second approach we call the Replica-Aware Gateway (RAG). This approach uses a load balancer or a proxy access node [6] that is aware of replica placement, and imposes one extra hop for routing a request.

In the third approach, known as the Replica-Aware Client (RAC), the clients cache the metadata of previously accessed objects [33], and use it to route subsequent requests. This approach only works for deployments in which clients are collocated with the storage system and are allowed to know 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 [23] (e.g., shared cloud storage like Amazon S3), this approach is not an option. Finally, this approach hinders deploying load balancers.

2.2 Software-Defined Networks

The SDN architecture divides the network into two planes: data and control. The data plane is a traffic forwarding plane that uses the information in the switch forwarding tables to forward messages. The control plane is an external software-based logically-centralized component that controls one or more switches by altering the entries in switch forwarding tables. The communication API between the controller and the switches is based on the widely adopted OpenFlow standard [19].

The OpenFlow standard [30] facilitates external control of a single-switch forwarding table. It allows inserting or deleting forwarding rules. Each forwarding entry includes a matching rule and an action list. If a packet matches a rule, the actions in the actions list are performed in order on the packet. OpenFlow has a rich set of matching rules including wild cards for matching IP and MAC addresses, protocol or port numbers. The actions include packet forwarding to a specific switch port, dropping the packet, sending the packet to the controller, or modifying the packet. The possible modifications include changing the source/destination MAC/IP addresses. To avoid the need for switches to contract the controller on every packet, forwarding rules are stored on switches and have an expiry period that is set by the controller. Controllers can update, delete, or extend the validity of the existing rules at any time.

These capabilities enable fine-grained control of network operations and facilitate application-optimized traffic engineering.

3 NICE SYSTEM ARCHITECTURE

NICE leverages the programmability and fine-grained control of network operations provided by recent advances in software-defined networks [19, 30] 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.

This section presents the system architecture and details the two core techniques that NICE proposes: 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 1): 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.

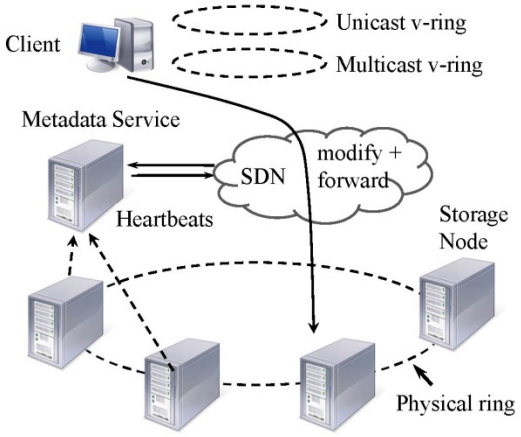

Figure 1. System Architecture. The client sends the requests using two virtual rings (vrings). The requests are rerouted in the network to the responsible storage node. The metadata service receives heartbeats from the nodes and maintains the mapping information in the forwarding tables.

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 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 prefix IP matching: we divide the virtual ring addresses into subgroups such that the number of vnodes per subgroup is a multiple 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 [40]. Compared to 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 or three-phase commit [41], Paxos [27], or Raft [32]. We illustrate in Figure 2 the put operation using the two-phase commit protocol (2PC), as a representative of these protocols to simplify our discussion. 2PC is among the early proposed protocols that are still widely used [5, 8, 13, 15].

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 and Raft use 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.

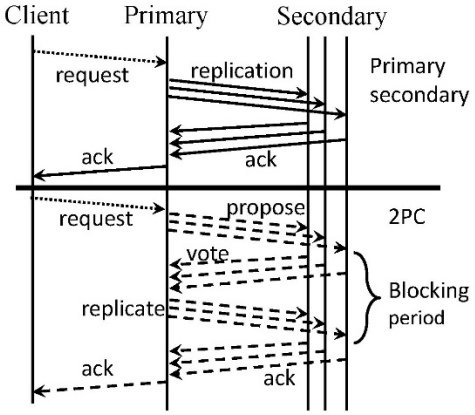

Figure 2. Put protocol alternatives. The figure shows the primary-secondary and 2PC put protocols. In the primary-secondary design (solid arrows) the primary replica serves all put and get request, hence no consistency protocol is necessary. In the two-phase commit (2PC) design (dashed arrows), two rounds of the 2PC protocol are needed to guarantee consistency.

We propose a consistency-aware fault tolerance mechanism. This mechanism solves the inefficiency problem found in current protocols by presenting inconsistent nodes as failed nodes to the client. The mechanism hides failed nodes, and newly-joining, but still inconsistent 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.

Unlike fail-stop failure model, which may allow returning nodes to receive client requests, the proposed mechanism *deterministically* only routes client requests to consistent nodes. Inconsistent nodes can communicate with the other consistent nodes to update their data set. This approach simplifies building fault-tolerant consistency protocols (as we will see next) by guaranteeing that clients can only access consistent nodes.

4 SYSTEM DESIGN

In this section we first detail the design of system metadata service (i.e., metadata for mapping objects to nodes), then we extend the core techniques of NICE to build an efficient replication mechanism, improve the consistency protocol fault tolerance, 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 replica. Similar to handling failures, the metadata service updates the switch and informs the affected replicas of the membership change.

This membership maintenance design is scalable; regardless of the number of storage nodes, the membership service needs *O(S)* messages to update the switch's forwarding table, where S is the number of switches in the platform, and only *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, is independent of the total number of nodes and is typically small (3 or 5).

While our current metadata service is centralized, it can radially adopt well-known designs for building a highly reliable distributed metadata services. One approach we are currently investigating is having a hot standby replica of the metadata node. Two workload characteristics make this design feasible: the stored metadata is small and changes infrequently, and the load on our metadata service is low as it is mainly invoked on node or network failures. These two characteristics make maintaining a hot standby server feasible.

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 [6, 9, 12, 18, 26, 33] (with the other popular technique being erasure coding).

Challenge. On a put request, a single node (known as the primary replica or the coordinator node) replicates the new object on *R-1* different 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-1* copies of the data on every put.

To alleviate the load on the replicating node Renesse et. al. proposed chain replication [43]. 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.

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 at least *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 object replicas, while the messages using the unicast ring are sent to one of the replicas of the object (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 an equal amount of data.

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 or require getting the object from the majority of nodes to resolve data inconsistency.

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 3) follows the 2PC protocol design with two main differences. First, it leverages multicast-based replication to load balance and efficiently replicate an object. Second, it improves the 2PC fault tolerance without requiring quorum-like protocols.

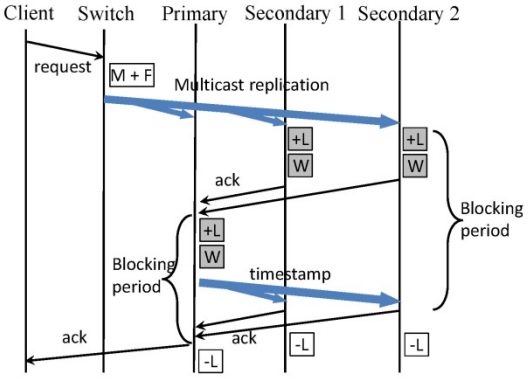

Figure 3. Consistency Mechanism. Timeline of the message sent in put operation in NICE storage. The switch performs modify and forward (M+F) for client packets to map the virtual address to the multicast group. (+L) is when a node logs the operation. (-L) is when the log entry is deleted. (W) is when the node writes the new object to the persistent storage. Gray boxes denote forced writes, and bold arrows denote multicasting. Object locks are maintained in memory only.

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, store the object to persistent storage, 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 release the lock on the object 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.

4.4 Fault Tolerance

Failure Model. NICE adopts a fault model assumed by current NOOB systems in which all node failures are assumed to be transient, with permanent failures being handled by administrator intervention [6, 9, 18] (The procedure for permanently adding or removing nodes is discussed at the end of this section). Consequently, when a node fails or is disconnected, the system does not automatically re-replicate the objects stored on that node, as these objects are still durably fully replicated.

Failure Detection. Node failure (include disconnected nodes) is detected through two techniques: the metadata service will declare the node failed if it misses three heart beats 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 two 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 [18]. 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. This makes the node receive and participate in the put operations but not serve get requests. Second, the recovering node contacts the handoff node to get all the objects stored during its failure. Finally, the node informs the metadata service that it has consistent data. The metadata service will add the node to the unicast vring mapping, remove the handoff node from all mappings, 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 3), 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 3, the secondary nodes will detect the failure by timing out on the replication message 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 and could have been served to subsequent get requests. 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 the administrator needs to reconfigure the system to add new nodes or remove nodes that permanently failed. To permanently remove a node, the administrator informs the metadata of the node removal. The metadata in its turn updates the forwarding rules related to the leaving node and informs all effected nodes of the membership change. Adding a new node to a replica set follows a procedure similar to rejoining a node after a temporary failure. The node is added first to the put vring to receive new updates and the primary node is informed of the new node. The node contacts the primary node to retrieve all keys stored in the hash range. Once the new node has consistent data it is added to the get vring and is made visible to get operations.

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 [14, 16]. In this case, storage systems use load balancing to distribute the get/put load on all the replicas of a given object.

Challenge. In current systems, a load-balancing node is deployed as a gateway to the system to forward client requests using the ROG or RAG approach (§2). This approach increases operation latency and requires provisioning load-balancers to avoid creating a system choke point. Alternatively, to avoid these drawbacks and to avoid the complexity of consistency protocols, latency-sensitive systems eschew load balancing and adopt the primary-secondary design [33, 42]. 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 [21, 44] 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, through heartbeats, periodic workload statistics, including the range of client IP addresses accessing each partition.

The metadata service divides the client address space into *R* divisions, such that each division size is a multiple 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 to NOOB load balancing, NICE builds an in-network load balancing without increasing the latency or deploying extra resources, as is the case in NOOB systems.

This approach increases the number of forwarding entries per partition of the unicast vring from 1 to *R* entries, each forwarding a subset of the clients to one of the replicas. Our future work will investigate more intelligent load-balancing techniques.

4.6 Switch Scalability

The proposed approach requires, for each physical partition, one entry in the switch forwarding table for the unicast vring mapping and one entry for the multicast vring mapping, if no load balancing is used. 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. Given this requirement, current switches can support large-scale storage systems with thousands of nodes. Current switches support tables with 128K or more entries; they can easily support storage systems with up to 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.

5 IMPLEMENTATION DETAILS

We implemented the NICEKV prototype following the NICE design. The NICEKV prototype is implemented in 14K lines of C++ code. The controller is implemented using 1K lines of python using the Ryu [10] framework.

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

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. While NICEKV implements a single node mapping service, the service can be easily partitioned on multiple nodes.

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 receive the data. After returning, the protocol keeps supporting straggling nodes until they finish or timeout.

5.1 Deployment Experience

NICE exploits the latest capabilities of OpenFlow-enabled switches. Unfortunately, through examining three platforms with OpenFlow-enabled switches, we found that the current switches

lag in terms of the supported OpenFlow features. 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.

The CloudLab [2] Utah cluster, which we use, provides partial support for 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 [7] 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 of the switching speed.

6 EVALUATION

Our evaluation demonstrates the performance benefits brought by NICE. This section first compares the performance of NICE and NOOB storage, then evaluates the two systems using the Yahoo benchmark [16]. In addition to the NICEKV prototype, we have implemented a NOOB storage prototype with rich configuration options. 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. NOOB prototype implements two consistency mechanisms: 2PC and Primary-backup designs. The NOOB prototype allows us to compare NICEKV to range of NOOB designs and configurations. Finally, to verify the NOOB performance, we ran a synthetic single client put and get workloads to compare the NOOB-RAG performance to the OpenStack Swift key-value store [6]. In both workloads NOOB-RAG performance was equivalent or slightly better than Swift storage.

Platform. We use a cluster of 30 nodes on the Cloud-Lab [2] Utah site. Each node has an 8-core ARMv8 2.4 GHz processor, 64GB memory, 120GB 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 radially 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 and sequential consistency.

6.1 Request Routing Evaluation

We compare the request routing performance of the NICEKV prototype, and three NOOB storage configurations: ROG, RAG, and RAC. We measure the performance of get requests issued from a single client. The evaluation shows the average of 1000 get operations while varying the object's size from 4 bytes to 1 MB.

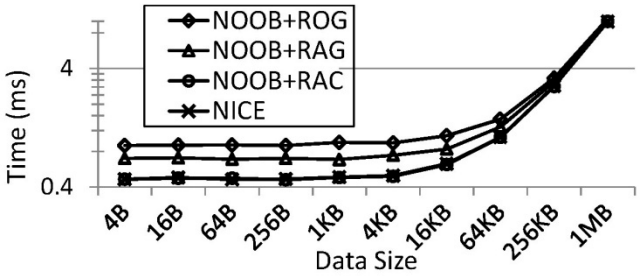

Figure 4. Request Routing Performance. The average time of the get operation. Note the log scaled y-axis. NICE and NOOB-RAC completely overlap.

Figure 4 shows the performance of the get operation on the four systems. NICE and NOOB+RAC systems achieve comparable performance as both achieve single-hop request routing. For small data sizes (less than 64KBs) NICE and NOOB+RAC systems achieve 2× and 1.5× performance improvement compared to NOOB+ROG and NOOB+RAG systems, respectively. This improvement is due to the delay added by the request routing mechanism. The benefits are not as pronounced with large data sizes, as transfer time dominates.

6.2 Replication Evaluation

We compare the replication performance of the NICE design and three configurations of the NOOB storage primary-only design: ROG, RAG, and RAC. The experiment measures the put performance of one client. The evaluation shows the average of 1000 put operations with objects sizes ranging from 4 bytes to 1 MB. The experiment measures replication performance in terms of operation time, generated network load, and load ratio between the primary and secondary replicas.

Replication time. Figure 5 shows the put operation time on the four systems. NICE storage achieves significant and consistent performance improvement across object sizes: up to 4.3× compared to NOOB+ROG, up to 3.4× compared to NOOB+RAG, and up to 2.6× compared to NOOB+RAC. The other systems lag NICE storage due to the extra effort needed for request routing and replication, while NICE storage uses optimal multicast-based replication.

Network load. Figure 6 shows the total link load generated by the put operation. NICE storage achieves, regardless of the object size, significant reduction in network load. NICE storage generates between 1.7× to 3.5× less network load compared to the other systems.

Storage Load Ratio. Figure 7 shows the ratio of the primary replica load to the secondary replica load. While all NOOB storage system configurations impose 3× more work on the primary compared to the secondary (this load imbalance is

proportional to the replication level), NICE load balances the load evenly across the primary and secondary replicas.

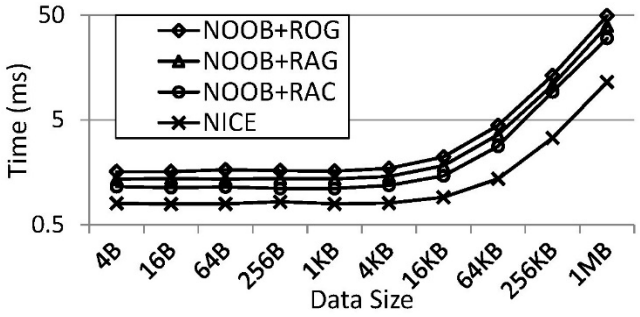

Figure 5. Replication Performance. The average time of the put operation. Note the log scaled y-axis

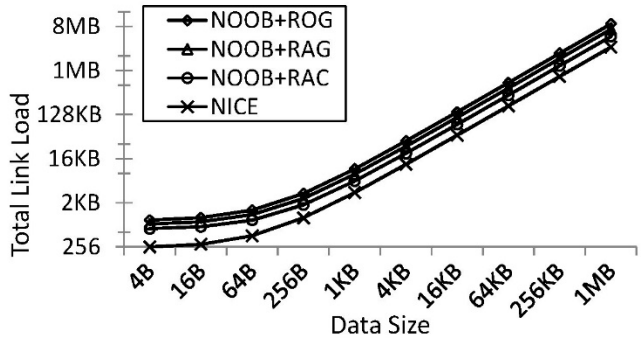

Figure 6. Network Link Load. The total network link load of the put operation.

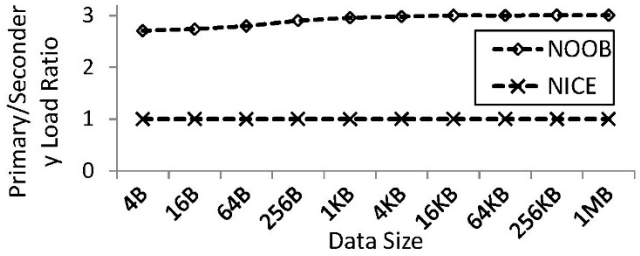

Figure 7. Storage Load Ratio. The ratio of the primary replica to secondary replica load in terms of amount of data sent/received during the put operation.

6.3 Quorum-based Replication Evaluation

This experiment compares NICE and NOOB storage quorum-based replication. The quorum design is appealing due to its ability to avoid slow or failed nodes. The experiment puts 1000 1MB objects using a replication level of 7, while varying the quorum write-set size (quorum size for short). To emulate slow nodes we configured the network connection of 3 replicas to be 50Mbps, while the rest of the nodes enjoy a 1Gbps connection.

Figure 8 shows (a) the put operation time and (b) achieved bandwidth when varying the quorum size. While the performance of both systems suffer with quorum sizes of 5 and 7 (as it is not possible to avoid slow nodes), we note that NICE storage achieves up to 5.6× better performance with quorum sizes of 1 and 3. While the primary replica in NOOB storage is waiting for the first quorum-size of nodes to finish, it is concurrently replicating the object to all replicas, including the slow ones, creating high contention on the primary link.

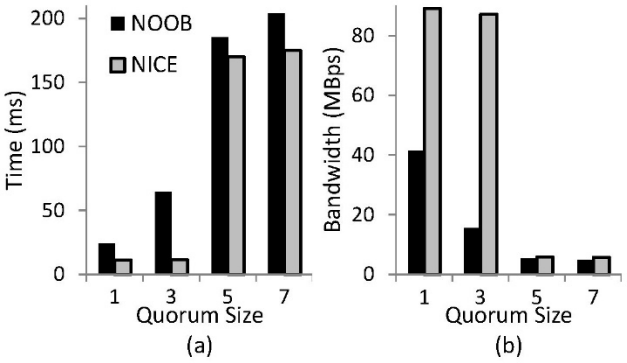

Figure 8. Quorum-based Replication Evaluation. Put operation performance using the quorum design. The experiment uses a replication level of 7 while varying the quorum size. The figure shows the put operation time (a) and bandwidth (b).

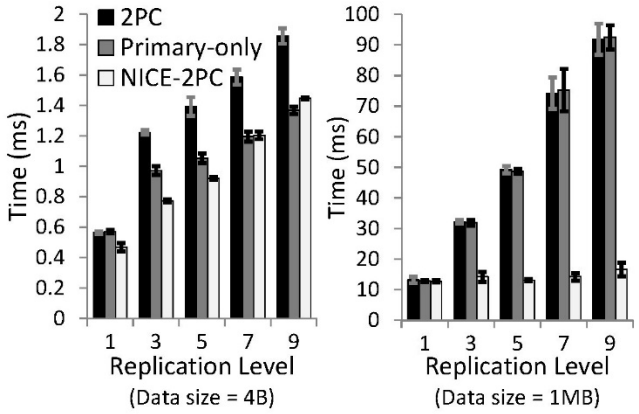

Figure 9. Consistency Mechanism Performance. The put performance while varying the replication level, with 4-byte (a) and 1MB (b) objects. Error bars represent standard deviation.

6.4 Consistency Mechanism Evaluation

We compare NICE storage to two NOOB storage configurations: primary-only and 2PC. To efficiently support highly popular objects, storage systems often create multiple replicas. This experiment evaluates the efficiency of the put operation while varying the replication level. NOOB storage use RAC request routing. We show the results for the two ends of the spectrum of object sizes, small 4-byte objects and large 1MB objects.

Figure 9.a shows the put operation time with 4-byte objects. NICE achieves up to 1.3× better performance than NOOB-2PC. NICE achieves comparable performance to NOOB primary-only replication, although it has an extra phase of communication. This is because of the multicast-based replication that reduces not only the data transfer time but also the overhead of creating and maintaining up to 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-only

design achieves better performance than NOOB-2PC due to 2PC protocol overheads.

Figure 9.b shows the put operation time with 1MB objects. NICE achieves up to 5.5× better performance than NOOB systems. The primary-only and 2PC achieve comparable performance since, with large objects; performance is dominated by replication cost. While NOOB performance degrades considerably: by 7× when 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).

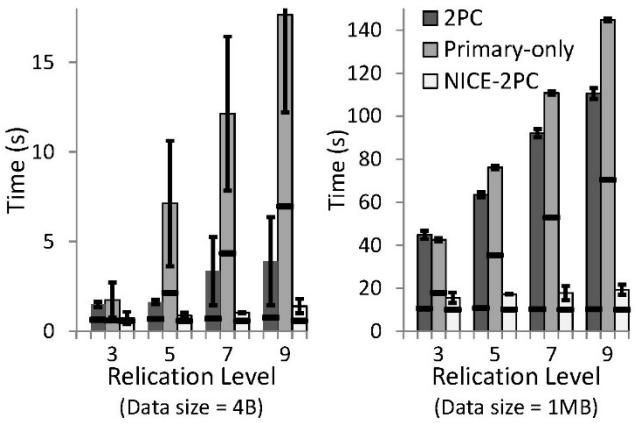

Figure 10. Load Balancing Evaluation. The three systems performance under the load balancing workload while varying the replication level and number of clients. The figure shows results with (a) 4-byte objects and (b) with 1MB objects. Bold markers show the performance of the get-only workload. Error bars represent standard deviation.

6.5 Load Balancing

This experiment measures the performance of NICE storage and two NOOB storage configurations (primary-only and 2PC) when serving highly-popular frequently-updated objects. We design a weak scaling experiment: we increase the number of clients proportional to the replication level. In each configuration 1 client puts the same object 1000 times, while *R-1* clients get the same object 1000 times.

Figure 10 shows performance with 4-byte objects (a) and 1MB objects (b). NICE storage achieves better performance than NOOB storage systems: up to 7.5× better than the primary-only configuration, and up to 5.5× better than the 2PC configuration in both object sizes. The line markers on the bars in Figure 10 show the performance of the workload without updating the shared key (i.e., without the put client). The marker shows that NICE and 2PC are able to load balance the get requests across replicas, while the primary-only design performance degrades with the increased workload as no load balancing is used. The figure also shows the significant overhead added by 2PC consistency mechanism (the difference between the marker and the top of the bar).

NOOB storage system performance degrades considerably when increasing the replication level and the number of clients, with primary-only performance degrading by 10× with small objects and 3.5× with 1MB object, and the 2PC configuration degrading by 2.6× with both sizes. This performance degradation is testimony that NOOB storage designs are not weakly scalable, i.e., 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).

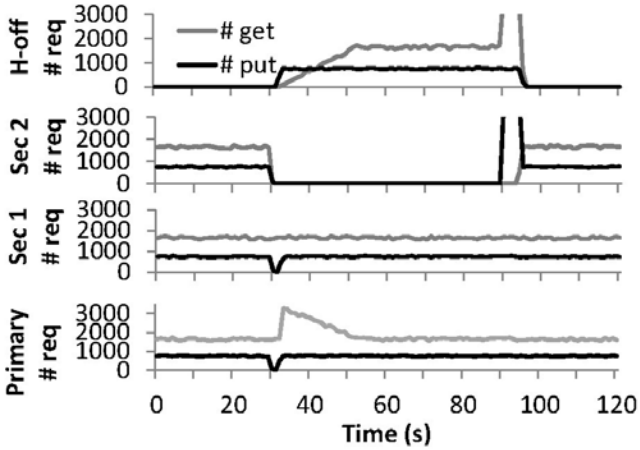

Figure 11. Fault Tolerance Evaluation. Secondary node 2 fails at 30s mark, triggering the fault tolerance mechanism, and 90s the node recovers, retrieves the missed objects from the handoff node, and starts serving client requests.

6.6 Fault Tolerance Evaluation

This experiment demonstrates the system fault tolerance mechanism. Three clients access the system with 20/80 put/get ratio and key size of 1KB. All objects are in the same partition. Figure 11 shows the number of put and get requests served per second. At the 30s mark, the secondary node 2 fails. The primary node detects the failure and informs the metadata service. The metadata service removes the failed node from the switch mappings and adds the handoff node to the replica set. This process makes the partition unavailable for put for less than 2 seconds (Figure 11 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 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.

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 the handoff node stores more objects 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 gets requests at the handoff node). Once the node has a consistent set of objects (second 95), the metadata service adds the node to the unicast switch mapping and removes the handoff node.

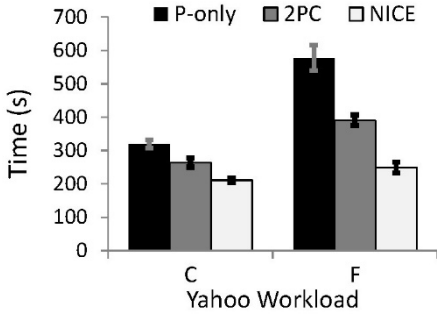

Figure 12. Yahoo Benchmark Evaluation. The three systems performance under two Yahoo benchmarks: read-only (C), and read-modify-write (F). Error bars represent standard deviation.

6.7 Real Workload Evaluation

To evaluate the system with real workloads we use the Yahoo benchmark (YCSB) [16]. YCSB includes workloads with a variety of get-to-put ratios. We use two workloads: C, the read-only workload, and F, the read-modify-write workload which generates the highest ratio (50%) of puts in YCSB. As in the majority of the Yahoo workloads, these two have a zipf popularity distribution.

The experiment compares the performance of NICE storage and two NOOB storage configurations (primary-only and 2PC). The system is accessed by 10 clients, each issuing 20K operations. We use the default YCSB configuration with 1KB objects.

Figure 12 shows the yahoo benchmark results. NICE achieves the best performance under the two workloads. Nice achieves 1.6× and 2.3× better than primary-only configuration under workload C and F, respectively. This improvement is due to the lack of load balancing in the primary-only configuration. Compared to 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 load-balancing latency and consistency-protocol overhead.

7 OTHER RELATED WORK

Request Routing. Beehive [35] 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 [21, 36, 44], access control [31], seamless VM migration [29], and to improve system security, virtualization and network efficiency [28]. These systems still use the network as a separate entity and use SDN to optimize its operations. Unlike current efforts, we co-design network operations with system operations and protocols to achieve significant benefits.

The MOM [34] and SwitchKV [46] projects are the closest in spirit to our project. MOM builds an SDN-optimized Paxos protocol by building an ordered multicast layer. Unlike MOM, we propose a new complete system architecture that co-designs network and storage support for higher performance and efficiency. SwitchKV [46] 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. Unlike NICE, SwitchKV does not use the SDN capability to optimize data replication and consistency mechanisms.

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 investigate building SDN-enabled storage systems that implement a more intelligent approaches to load balancing and a better support for more complex key-value queries.

ACKNOWLEDGMENT

We thank our shepherd Dean Hildebrand for his guidance and insightful comments, and thank the anonymous HPDC ‘17 reviewers for their feedback. We thank 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 NSERC, NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. 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] Amazon elastic compute cloud (ec2). https://aws.amazon.com/ec2. Accessed: 2015.
[2] Cloudlab. http://www.cloudlab.us/. Accessed: 2015.
[3] Google app engine. https://appengine.google.com. Accessed: 2015.
[4] Microsoft azure: Cloud computing platform and services. https://azure.microsoft.com/. Accessed: 2015.
[5] Mongodb. https://www.mongodb.org/. Accessed: 2016.
[6] Openstack swift. http://docs.openstack.org/developer/swift/overview_architecture.html. Accessed: 2015.
[7] Openvswitch: Production quality, multilayer open virtual switch. http://openvswitch.org/. Accessed: 2015.

[8] Postgresql. http://www.postgresql.org/. Accessed: 2016.
[9] Riak cloud storage. http://basho.com/riak-cloud-storage/. Accessed: 2015.
[10] Ryu sdn framework. http://osrg.github.io/ryu/. Accessed: 2015.
[11] Spark lighting fast cluster computing. http://spark.apache.org/. Accessed: 2015.
[12] Voldemort project. http://www.project-voldemort.com/voldemort/design.html. Accessed: 2015.
[13] Marcos K. Aguilera, Arif Merchant, et al., Sinfonia: A new paradigm for building scalable distributed systems. Symp. on Operating Systems Principles (SOSP), 2007.
[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2012.
[15] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, et. al., Windows azure storage: A highly available cloud storage service with strong consistency. Symposium on Operating Systems Principles (SOSP), 2011.
[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. Symposium on Cloud Computing (SoCC), 2010.
[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. Symposium on Operating Systems Design and Implementation (OSDI), 2004.
[18] Guiseppe DeCandia, Deniz Hastorun, et al.,Dynamo: Amazon's Highly Available Key-Value Store. Symposium on Operating Systems Principles (SOSP), 2007.
[19] The Open Networking Foundation. Open networking foundation: Openflow switch specification. Version 1.3.0.
[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. Symposium on Operating Systems Principles (SOSP), 2003.
[21] Nikhil Handigol, Mario Flajslik, et al., Aster* x: Loadbalancing as a network primitive. GENI Engineering Conference (Plenary), pages 1–2, 2010.
[22] J. H. Howard, M. L. Kazar, et al., Scale and performance in a distributed file system. Technical report, Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA, August 1987.
[23] Javvin Technologies Inc. Network Protocols Handbook (2Nd Edition). Javvin Technologies Inc., 2005.
[24] R. Jain, P. Sarkar, and D. Subhraveti. Gpfssnc: An enterprise cluster file system for big data. IBM Journal of Research and Development, 57(3/4):5:1–5:10, May 2013.
[25] David Karger, Eric Lehman, et al.,.Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. Symposium on Theory of Computing (STOC), 1997.
[26] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40, April 2010.
[27] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.
[28] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using openflow: A survey. Communications Surveys Tutorials, IEEE, 16(1):493–512, First 2014.
[29] Ali Jos´e Mashtizadeh, Min Cai, et al., Xvmotion: Unified virtual machine migration over long distance. USENIX Annual Technical Conference (ATC), 2014.
[30] Nick McKeown, Tom Anderson, et al.,Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, March 2008.
[31] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic access control for enterprise networks. Workshop on Research on Enterprise Networking, 2009.
[32] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. USENIX Annual Technical Conference (ATC), 2014.
[33] Diego Ongaro, Stephen M. Rumble, et al., Fast crash recovery in ramcloud. Symposium on Operating Systems Principles (SOSP), 2011.
[34] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. Symposium on Networked Systems Design and Implementation (NSDI), 2015.
[35] Venugopalan Ramasubramanian and Emin G¨un Sirer. Beehive: O(1) lookup performance for power law query distributions in peer-to-peer overlays. Conference on Symposium on Networked Systems Design and Implementation (NSDI), 2004.
[36] Brendan Cully, Jake Wires, et al., Strata: scalable high-performance storage on virtualized non-volatile memory. Conf. on File and Storage Technologies (FAST). 2014.
[37] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. International Conference on Distributed Systems Platforms (Middleware), 2001.
[38] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. ACM Transactions on Computer Systems, 2(4):277–288, November 1984.
[39] Russel Sandberg. The Design and Implementation of the Sun Network File System. USENIX Summer Technical Conference, June 1985.
[40] Ion Stoica, Robert Morris, et al., Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. SIGCOMM '01, August 2001.
[41] Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems: Principles and Paradigms (2^{Nd} Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
[42] Douglas B. Terry, Vijayan Prabhakaran, et al., Consistency based service level agreements for cloud storage. Symposium on OS Principles (SOSP), 2013.
[43] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. Symp. on OS Design & Implementation (OSDI), 2004.
[44] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE'11.
[45] Sage A. Weil, Scott A. Brandt, et al., Ceph: A Scalable, High-Performance Distributed File System. Symposium on OS Design and Implementation (OSDI), 2006.
[46] Xiaozhou Li, Raghav Sethi, et al., Be fast, cheap and in control with SwitchKV. Conference on Networked Systems Design and Implementation (NSDI). 2016.