Kua: A Distributed Object Store over Named Data Networking

Varun Patil
varunpatil@cs.ucla.edu
UCLA
Los Angeles, USA

Hemil Desai
hemil10@cs.ucla.edu
UCLA
Los Angeles, USA

Lixia Zhang
lixia@cs.ucla.edu
UCLA
Los Angeles, USA

ABSTRACT

Applications such as machine learning training systems or log collection generate and consume large amounts of data. Object storage systems provide a simple abstraction to store and access such large datasets. These datasets are typically larger than the capacities of individual storage servers, and require fault tolerance through replication. In this paper, we present Kua, a distributed object storage system built over Named Data Networking (NDN). The data-centric nature of NDN helps Kua maintain a simple design while catering to requirements of storing large objects, providing fault tolerance, low latency and strong consistency guarantees, along with data-centric security. Our prototype Kua implementation provides easy-to-use primitives to let applications store and access data securely, and our initial evaluation suggests that Kua can leverage NDN's capabilities of multicast data delivery and in-network caching to achieve higher efficiency than existing object storage systems.

CCS CONCEPTS

• **Networks → Cloud computing**; • **Information systems → Distributed storage**; *Storage replication.*

KEYWORDS

Named Data Networking, Object Store, Distributed Storage

ACM Reference Format:
Varun Patil, Hemil Desai, and Lixia Zhang. 2022. Kua: A Distributed Object Store over Named Data Networking. In *9th ACM Conference on Information-Centric Networking (ICN '22), September 19–21, 2022, Osaka, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3517212.3558083

1 INTRODUCTION

Data is ubiquitous in software systems, and the size of this data has grown exponentially over the last two decades, so have the systems that store and process this data. Initial distributed storage systems included distributed file systems, such as the Hadoop File System (HDFS) [19] and Google File System (GFS) [5], scaled storage to petabytes of data, yet were restricted to a single tenant. With the advent of the cloud platform, new storage systems such as the Amazon Simple Storage Service (S3) [7] emerged, which provided the same scale, fault-tolerance and reliability in a multi-tenant architecture. Similar systems are now offered across different cloud platforms such as Google Cloud Storage, Azure Storage, etc.

Amazon S3 and similar systems provide data storage services to users as an object store, where data is stored as objects referenced by a unique key. These objects can be very large in size, and data can be written to and read from objects using simple primitives provided by the store. Object storage services have grown in popularity over the past decade, and have become data storage of choice for a variety of workloads, including datasets for machine learning, media items, unstructured data lakes, large file caches, etc. Object storage systems can range from on-disk storage like S3 that persist objects to disk, to in-memory systems like Plasma [13] and Redis [3] for high performance applications. Some of these systems, e.g. Plasma, are part of a bigger distributed system like Ray [13], showcasing the importance of object storage in software systems we use today.

To meet the scalability requirements of today's applications, data object stores need to be implemented as distributed systems running on multiple storage servers, or nodes. A set of nodes, or a *cluster*, is typically controlled by a single administrative entity, and can have hundreds of nodes. Users of the object store communicate with nodes in the cluster for data storage and retrieval, and the nodes also communicate internally for bookkeeping and data replication. As a result, efficient communication, both internally and with the users of the system, is the key for a fast and efficient distributed storage system.

Today, the majority of these systems communicate using the TCP/IP [15] protocol stack. Unfortunately, IP's point-to-point datagram delivery service is incongruent with the data-centric nature of object storage services. Since IP identifies end hosts and applications identify chunks of data, IP-based storage systems need to consistently map application data identifiers to the locations of the data, i.e. the IP addresses of the storage nodes. Further, several of these systems require a large number of TCP connections, in the worst case $N \times N$ connections with N being the number of storage nodes (such as in the case of Redis Cluster [1], a distributed implementation of Redis). This represents a scaling challenge. IP multicast, a key enabler for efficient data replication, is not widely deployed in IP networks, and lacks performant transport protocols and usable security implementations.

In this paper, we present the design of a distributed immutable object store, *Kua*, that uses Named Data Networking [2] as its network layer. NDN's data-centric nature enables multicast data delivery and in-network caching, providing promise of superior performance in distributed object storage systems as compared to existing designs over IP. We utilize the basic concepts from existing distributed database and storage systems, such as consistent hashing and chain replication, and apply them in the context of an information centric network to realise the design of Kua.

One distinguishing factor of Kua from existing object stores built over TCP/IP is the simplicity of the design. NDN's data-centric architecture allows clients to fetch data using semantically meaningful identifiers directly at the network layer, without a layer of indirection to locate the data first. Kua also provides strong consistency guarantees without complex distributed consensus or locking mechanisms. We also discuss how Kua can utilize NDN's security model to provide strong guarantees about the authenticity of data along with confidentiality and access control, without relying on trusted storage servers and secured channels through the use of Transport Layer Security (TLS).

The rest of the paper is organized in the following manner – in §2, we provide some background on networking in object stores and NDN. In §3, we discuss the design goals and applicability of Kua, along with an overview of the design. We then discuss the design in detail in §4, the results of the evaluations in §5, followed by further discussion in §6, and the conclusion.

2 BACKGROUND

In this section, we provide the requisite background of object stores and a perspective from the network layer.

2.1 Object Storage

Early research on distributed storage led to the development of network attached disks [6] as an alternative to distributed filesystems. The concept of moving the functions of storage, namely reads and writes, and more complex functions such as namespace manipulation into separate layers further evolved into object storage systems. Separating the filesystem functions from the underlying storage offers both better performance and scalability compared to distributed filesystems, such as the Network File System (NFS) [16] and Andrew File System (AFS) [8] protocols.

Object stores represent storage as a collection of binary blobs and associated metadata, with the storage namespace defined by the storage system rather than the application. Applications using object stores perform operations such as creating new objects, writing to and reading from objects [4]. Some of the key aspects in which objects in an object stores differ from files in a filesystem are enlisted below.

(1) **Flat Identifiers** Objects are not structured hierarchically in directories[1]. Instead, the object store assigns a unique identifier to a newly created object, which the application can then use to perform further operations on the object. Modern object stores typically provide APIs to store and retrieve objects using application-defined keys, similar to a key-value database. As a result of using flat identifiers, the application must keep track of what data is stored in which object. In file systems, the equivalent function is achieved through the use of semantically meaningful directory structures, which in turn is enabled by the storage module itself that may be part of the operating system or the distributed filesystem.
(2) **Immutability** Once an object is created and written, the contents of the object cannot be altered. A new object must be created if the data needs to be updated[2]. In filesystems, on the other hand, the user may make a partial update to any part of the file any number of times. A result of this difference is that objects, unlike files, do not require read and write locks during operations.
(3) **Simplified Security** In object stores, operations can be secured easily by enforcing access control to individual objects through the use of application credentials, and individual objects can also be independently encrypted using different keys for security at rest. In filesystems, the complexity of hierarchical access control can increase significantly when multiple users belonging to different semantic groups need to access the same piece of data. Data encryption is usually performed at a block or disk level, encrypting all files with the same key.

Popularity of object stores has increased massively over time, and are used for a variety of workloads today. Requirements from object stores ranges from the ability to store very large objects for long term to achieving ultra-low latency through the use of in-memory storage, and the ability to run on heterogeneous hardware with varying storage capacities. As a result of these requirements, most object stores are distributed systems, which makes efficient communication a key component of their design.

2.2 Networking in Distributed Storage

Much of existing research geared towards improving storage systems places focus on local optimizations, such as in-memory caching of frequently accessed objects, speculative access [25], or faster reads using techniques such as RDMA [10]. Some of these optimizations may further increase the cost of communication, both in terms of network overhead and complexity of the system. This work takes a different approach towards improving distributed storage, by revisiting communication in these systems.

In a distributed data store, a large amount of overhead is incurred due to bookkeeping and coordination between the nodes. In context of object stores, since applications only understand object identifiers and the network only understands IP addresses, stores need functions to keep track of which object is stored where. Maintaining a consistent view of such information requires coordination between the nodes, contributing significantly to the increasing complexity of data storage systems.

The root cause of this problem is the host-centric nature of IP. The knowledge of the exact location where the data is stored is largely irrelevant to users of a data-centric system such as an object store. However, users are forced to locate the data before they can fetch it, since the network layer only provides point-to-point communication. The overhead of communication is also aggravated by the lack of multicast in TCP/IP, over which existing systems are built. Several functions of data stores such as replication and state change notifications require the same data to be delivered to multiple nodes. This is generally achieved by unicasting the information to each node, typically through a central master node or one or more message brokers. Further, the point-to-point nature of communication also entails that requests for accessing data stored in a distributed storage must first pass through multiple gateways, such as an authentication layer or one or more load balancers.

NDN, as described in the next section, provides a data-centric network layer in lieu of IP's host-centric model. In NDN, Data is

[1]Object stores may provide filesystem-like overlays to help applications track objects using familiar development primitives. Internally, objects are still referenced and manipulated using flat identifiers.
[2]Object stores may provide APIs for overwriting entire objects keeping the same identifier; such changes are tracked internally through versioning.

identified, requested and secured using semantically meaningful application layer identifiers at the network layer. As a result, applications running over NDN do not need to know what data is stored where, as long as they can identify what data they need. This property of NDN opens up possibilities for designing simpler distributed storage systems over NDN.

2.3 Named Data Networking

In the NDN architecture [2], individual data packets are identified using semantically meaningful names. Consumers that desire a certain Data packet send an Interest packet with the name of the data, and the network attempts to find this data and return it to the consumer. Thus, NDN uses application layer identifiers of data, i.e. the name, as opposed to using separate addresses at the network layer like IP.

A key difference between NDN and IP is the absence of the address of destination in a data request. Because an NDN network directly uses application derived data identifiers in network routing, a data store design over NDN does not need to provide clients with a mapping between data identifiers and addresses of storage nodes. NDN also provides a mechanism to forward requests of data in the right direction even when data name are not used in routing and forwarding. It does so by including in the Interest packet a *Forwarding Hint* which contains a name that is in routers' forwarding table. Consumers can include in an Interest the name of the requested data, together with a Forwarding Hint that directs the Interest towards (one of) the node with the requested data.

Fetching data by name naturally results in multicast data delivery, which is a key requirement for efficient data replication. In systems over IP, copies of the same data must be unicast to each replica, increasing unnecessary network traffic. NDN enables us to avoid this problem by supporting both synchronous multicast delivery and asynchronous delivery through in-network caching. Multicast also helps alleviate server load when a large number of clients request the same piece of data from the storage.

NDN also provides a data-centric security model that simplifies security and access control. Every Data packet in NDN is signed by the producer of the data. As a result, the contents of each packet are cryptographically bound to the name and the producer. Such a model makes it straightforward to implement verification of data at the consumer using trust schemas, regardless of whether the data was obtained from the producer, a data store or network cache. Confidentiality and access control is enabled by Name-based Access Control (NAC) [26], which works by encrypting content at the time of production and automating the distribution of keys.

With this background, we design a new distributed object store that can leverage NDN's capabilities, leading to improved performance and reduced system complexity.

3 OVERVIEW

In this section, we outline the intended usage, assumptions and design goals for building Kua, along with an overview of the design.

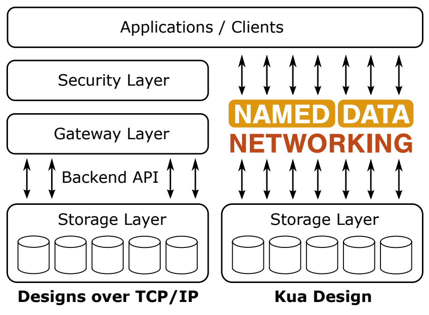

Figure 1: Design simplification with NDN[3]

3.1 Applicability & Assumptions

While large corporations running data centers may have the technical expertise required to manage complex systems, smaller organizations typically do not. Enterprises such as hospitals require solutions for reliable on-premise data storage, with properties such as fine-grained access control, while having low maintenance overhead. We target the design of Kua towards such enterprises with data storage requirements and security considerations. At the same time, we attempt to keep the design equally applicable for other scenarios, such as the storage and processing of large amounts of data in datacenters, especially with further development of high performance NDN forwarders [17]. We also note that the store may run in an NDN overlay on an IP network, using TCP or UDP as the link layer of NDN.

Since the target usage is inside a single enterprise, we assume that all storage nodes in the enterprise are controlled by a single administrative entity. This is a key simplifying assumption in the design, since it also entails that individual nodes have no preference on what pieces of data they store. Note that this does not make assumptions in regards to heterogeneity of hardware, storage capacities of individual nodes or network topology. We also assume the usage of an existing security solution, entailing that every node in the storage system as well as all producers and consumers of data are configured with appropriate certificates, and can authenticate each other.

3.2 Design Goals

In order to meet the requirements described above, we identify the following design goals.

Simple design Our foremost goal of building a new object store is to simplify the designs of both the store itself and the applications using it. A simple design can help reduce network and processing overhead, while making maintenance easier for operators running the system at the same time. Reducing application complexity also makes finding and fixing bugs easier, thus lowering development related costs.

Automated operation Similar to plug-and-play, easy and automated operation is an important goal of the design of a distributed storage system. The store should be able to function with minimal configuration even if used on heterogeneous hardware with varying processing, storage and network capacities across nodes.

[3]This simplified view only showcases how NDN applications do not require several access gateways for storage, and does not imply e.g. the absence of a security layer.

The system should also be fault tolerant and continue to function without human intervention in the event of failures.

Ease of use To reduce application complexity, we recognize the requirement of simple APIs as a design goal. Applications should not need to perform operations such as opening connections, obtaining locks and performing extensive bookkeeping. Since the store may be used for mission-critical data, we recognize providing strong consistency as a design goal to simplify application logic. Any read operations that take place after a successful write must always succeed and return the correct data.

Security Security is generally considered as an afterthought in data storage systems. We aim to design Kua to be secure from ground up, instead of adding security as an overlay over data storage. Using a security framework, applications should be able to identify and verify the producer of each piece of data, providing strong authenticity guarantees. The security framework should also provide means for access control and confidentiality through encryption for sensitive data.

3.3 Design Overview

To realize the goals described in the previous section, we design Kua as a distributed immutable object store running over NDN. We identify three questions for a data store design, and answer these with the Kua design.

What to store? We define the storage unit or *Object* as the smallest Application Data Unit that has semantic meaning to an application. As a result, the application can assign a semantically meaningful identifier to the object. Applications use this identifier directly to store and access data from a Kua cluster. At a lower level, applications insert objects into the store as a collection of NDN Data packets, which are semantically identified across layers using this identifier as the name. Larger objects first need to be segmented into multiple packets, with the name of each packet including a segment number. These packets, and hence the entire object, can then be fetched later by using the same identifier, i.e. by sending an Interest for the object name. As every NDN Data packet must be signed by the producer, this process preserves the cryptographic binding of the name and contents, since the packets include the producers' signatures. The identification and storage of objects in Kua is described in further detail in §4.1.

Where to store? In a distributed store, we need a mechanism to determine which piece of data is stored at which node. To realize this, we use a hash function to divide the namespace of data into a predefined number of subsets called *buckets*, and assign each storage node one or more buckets to store. To perform this assignment, we design the Auction protocol, as described in §4.2.

How to locate? When applications desire to access data in existing systems that run over TCP/IP, they need to first locate the data in order to establish a TCP connection to request the data. In the Kua design, data is located simply by computing which bucket the data belongs to; a trivial task. The storage nodes in a Kua cluster register a route for each bucket that they are assigned. As a result, an NDN network can forward Interests for the data to the correct nodes using a Forwarding Hint containing the bucket identifier, as described in §4.3.

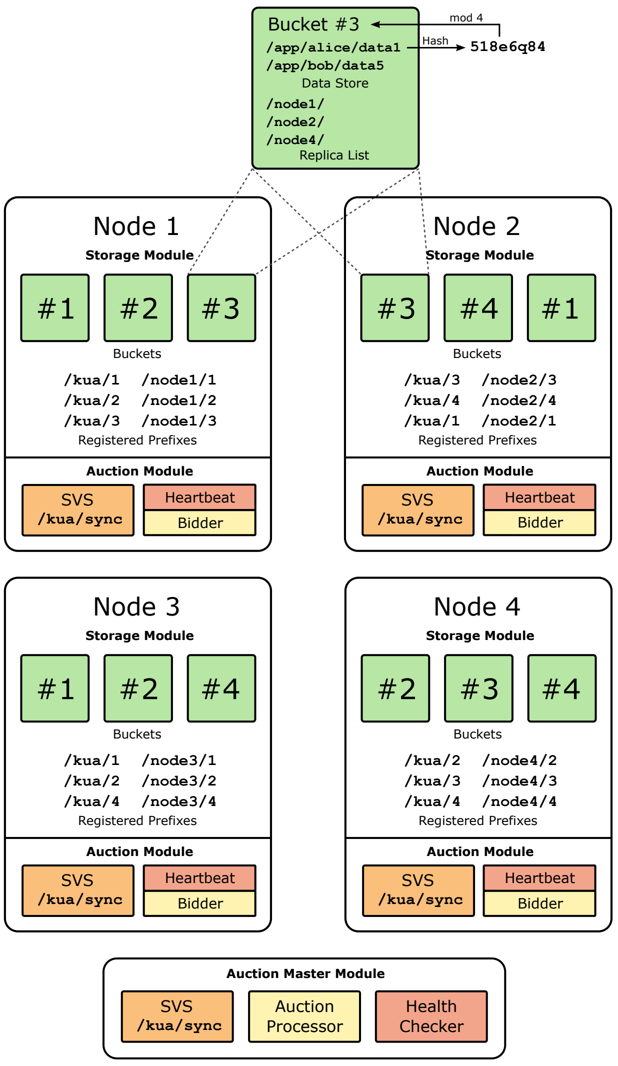

Figure 2: Kua Design Overview

Thus, the three important components of Kua's design are the storage unit and identifier, the distribution of data in the store, and the storage and access interfaces. We describe each of these components in detail in the next section.

4 KUA DESIGN

In this section, we discuss the design of Kua in further detail. We elaborate on the naming system used by objects in a Kua store, and its advantages, specify the distribution and replication of data inside Kua through the Auction protocol in further detail, followed by the object storage API and functionality, and some optimizations.

4.1 Object Identification & Storage

Since NDN uses application layer names at the network layer, in building Kua over NDN, we apply the same paradigm to storage, where the identifier of the object is a meaningful description of the data. Consequently, the same semantic identifier is used across layers, i.e. at the network layer and for addressing storage. As a result, an NDN Data packet is the smallest addressable unit of data stored in a Kua cluster. To realize this design, Kua directly stores the original data packets produced by a client application. Since packets in NDN are immutable, Kua is naturally an immutable object store, and changes to objects must be tracked by versioning the object names. Applications must also ensure that identifiers for objects are not reused; such a requirement is considered reasonable similar to existing key-value databases.

Being able to store and address data directly has several key advantages. Since the names of objects (and thus packets) already have semantic meaning to the application, developers do not need to worry about having a separate namespace for object identifiers. Since each Data packet is signed by the producer, storing the packet also allows consumers to directly verify the signatures of the data. This allows applications using Kua to directly utilize existing NDN security solutions, as discussed further in §6.1. Since queries are made directly for semantic data names, the system can also fully leverage NDN's in-network caching for frequently accessed data.

It must be noted here that while the network MTU limits the size of the addressable storage unit, i.e. an NDN Data packet, this is expected to have minimal impact on performance due to the optimizations described in §4.4. This includes the use of signing manifests to reduce the overhead of storing public key signatures for each packet, by only signing a list of digests of multiple packets.

For effective distribution and replication of the stored data, Kua needs a mechanism to determine which node should store what data. We describe the Auction protocol next, which enables allocation of subsets of data to storage nodes in a distributed manner.

4.2 Auction Protocol

In a distributed storage system, it is essential to determine which data is stored and replicated on which machine. While a real system would require more guarantees, such as data not being replicated on servers inside the same rack, our initial design simply attempts to distribute the data of an object store efficiently to nodes in the system. In several storage systems such as HDFS, storage nodes are assigned data for storage by a central master controller. The master can collect information, such as the available storage space from all nodes and make storage decisions with a view of the entire system. However, such a system has the obvious disadvantage of having a single point of failure, or needs to achieve locking and consistency through complex mechanisms between multiple master nodes. In contrast, the protocol presented next works in a decentralized fashion, where storage nodes express willingness to store data, and the data is assigned to the most willing nodes.

Kua divides the namespace of all addressable units in the store equally into a fixed number of partitions using consistent hashing. This can be achieved simply by hashing the identifier of the addressable unit, i.e. name of the data packet, and performing a modulo operation. Such a partition is referred to as a storage *bucket*. Since the size of each addressable unit, i.e. the packet, can be assumed to be relatively small, consistent hashing ensures that resources at all nodes are utilized nearly equally. Existing systems such as Cassandra [9] employ similar techniques, which have been scaled to thousands of nodes.

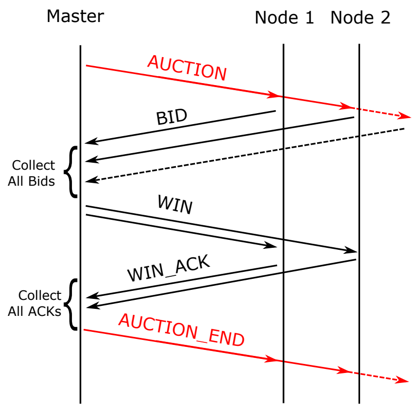

Figure 3: Kua Auction Protocol

Kua distributes data among storage nodes at the granularity of buckets. At a high level, each bucket is auctioned by an elected auction master node periodically. Each storage node then places a bid for the particular bucket by considering only the local conditions at the node, such as available resources and the amount of data the node is currently storing for the auctioned bucket. The master collects all such bids and announces the winners of the bucket, which are the nodes corresponding to the top *n* bids, where *n* is the replication factor. Object operations for objects in the auctioned bucket are then routed to the winning nodes. The master repeats this process for each bucket, thus auctioning the entire namespace.

The auction protocol is illustrated in Fig. 3, and is executed over the distributed pub/sub framework, SVS-PS[4] [12]. The auction master first announces a new auction for a bucket using the `AUCTION` message. All storage nodes in the cluster subscribe to messages from the master node, and receive this announcement. The nodes then reply to the auction message with a `BID` message, containing the auction identifier and the bid amount. The master collects all such bids and announces each winner using the `WIN` message. The winners acknowledge the win using a `WIN_ACK`, following which the master ends the auction using an `AUCTION_END` message. The `AUCTION_END` message also carries a list of winners, so that the storage nodes can keep track of the replicas of a bucket.

The auction process brings up two important requirements that must be addressed. First, storage nodes must decide how much to bid for an auctioned bucket. To determine the amount to bid, a node only considers its local state and resource availability. The final bid amount is engineered to be inversely proportional to the number of buckets currently stored by the node, and directly proportional to the amount of data that the node stores for the bucket being currently auctioned, and the resources available on the node. A random component is used for tie-breaking. The amount to bid for a certain bucket *b* can be calculated as,

$$\mathrm{Bid}(b) = \frac{\alpha}{|C|} + \beta \cdot H_b + \lambda \cdot S(b) + r \tag{1}$$

$$S(b) = F - W_{\mathrm{avg}}(1 - H_b) - \sum_{i \in I} W(i) + H_b \sum_{o \in O} W(o) \tag{2}$$

[4]Since SVS-PS provides reliable delivery of all publications, factors such as latency and/or losses do not affect the auction process significantly.

C = set of buckets currently stored
I = set of incoming buckets
O = set of outgoing buckets
$$H_b = \begin{cases} 1 & b \in C \\ 0 & \text{otherwise} \end{cases}$$
$W(b)$ = storage consumed by bucket b
W_{avg} = average size of stored bucket
F = available free storage space
r = small random number
α, β, λ = constants

Such a mechanism ensures stability in steady state and recovery from faults and overloads. In steady state, since the existing replicas have the largest amount of data for an auctioned bucket, they would always win the auction for that bucket. On failure of a node, buckets at the node would be transferred to other nodes most willing to accept new data, e.g. by virtue of having the least number of buckets stored, or the highest amount of available free space. In case a node is overloaded, it may also choose to lower its bids for some buckets it currently stores, thus leading to another node winning the bucket in the next auction[5]. This would initiate a transfer of the bucket out of the overloaded node, providing automatic load balancing. Further discussion on the behavior of the Auction process during failure recovery can be found in §6.3. We note here that the precise engineering of the bidding function is critical to the performance of the Auction protocol.

The second and simpler requirement is that the master must know the number of expected bids before declaring the results of an auction. Kua employs a periodic heartbeat that is multicast from each node to all the other nodes in the cluster for discovery and health monitoring. This heartbeat also enables the master to determine the number of bids it should expect. Since SVS-PS provides reliable and fast delivery of messages, the auction can be expected to have minimal delay and overhead. However, if all bids are not received within a timeout period, the master may cancel an auction and attempt to detect any potential failures.

It must be noted here that while the Auction protocol does employ a master node for collecting bids and announcing results, the master node is elected only periodically and ephemerally. In the event of failure of the master node, the system continues to function as normal, since the only state carried at the master is that of the currently ongoing auction. In this event, the Auction can be triggered again by re-electing[6] the master node as long as the rest of the nodes can achieve quorum. Since the master does not perform any other operations and maintains no state, it does not become a single point of failure in the system.

With the Auction protocol, the storage nodes in a Kua cluster can reach consensus about the buckets they must store, along with the locations of other replicas of the buckets. Next, we describe the protocol used by applications to store objects inside these buckets.

4.3 Object Protocol

Kua implements a very simple interface for the basic functions of an object store. Since NDN does not name end hosts for data

[5]All buckets are auctioned periodically by the master or on request. Steady state stability of the bidding function ensures the same nodes win a bucket at each auction.
[6]An election is triggered when no heartbeat is detected from the master.

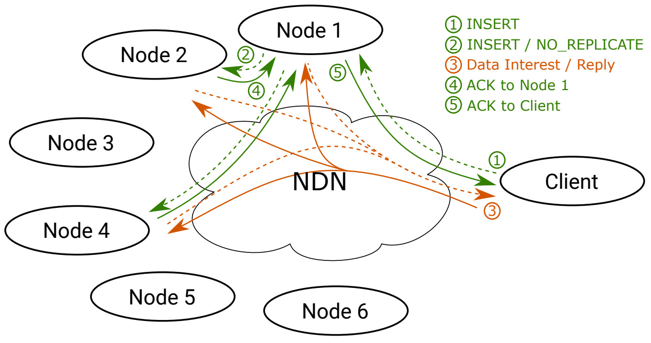

Figure 4: Object Insertion

transfers, Kua also adopts a data-centric design for the storage API. This enables minimal configuration and setup at the client before accessing a Kua cluster, since the client does not need to know the IP address of a master or each node of the cluster for storing and accessing data.

On winning a bucket, a Kua storage node registers two prefixes with the network as soon as the `AUCTION_END` message is received. The first prefix identifies the cluster and the bucket identifier, while the second identifies the individual node and the bucket identifier. In subsequent examples, we assume that the cluster runs under the prefix `/<kua-prefix>/` which is common to all nodes, and a representative node uses the prefix `/<node-prefix>/`, which is unique to each node in the cluster. Thus, on winning an auction for bucket number `<bucket-id>`, the node registers the prefixes:

(1) `/<kua-prefix>/<bucket-id>`
(2) `/<node-prefix>/<bucket-id>`

The second prefix is only used for internal Kua communication, and is not disclosed to clients using the object store, which use only the cluster prefix for all commands.

Object Insertion When a client wants to insert an object into the store, it sends one `INSERT` Interest for each packet in the object. An insertion Interest contains the name of a single NDN Data packet. The store fetches this Data from the client and returns an acknowledgement on successful insertion. Since the hashing algorithm and number of buckets can be pre-configured into the client, the client can calculate the target bucket for a particular packet name. The identifier of the bucket, prefixed by the cluster prefix for forwarding and suffixed by the name of the Data packet to be stored, form the insertion Interest. All `INSERT` Interests are signed by the client for authentication [14]. As a result, the signed portion of the name of the Interest looks like:

`/<kua-prefix>/<bucket-id>/INSERT/<data-name>`

As described earlier, each Kua node only registers routing prefixes for buckets they are actively storing. As a result, when the client sends such an insertion Interest to the network, NDN forwards it to the nearest storage node in the cluster which is a replica for the corresponding bucket. On receiving this Interest, the node duplicates the request and sends it to all replicas for the bucket including itself. This interest is prefixed with `<node-prefix>`, and is thus forwarded to only a particular replica. A `NO_REPLICATE` flag in the application parameters of the Interest is used to prevent loops of chain replication.

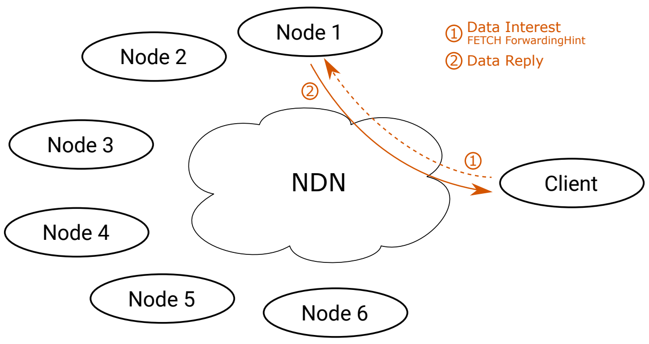

Figure 5: Object Fetching

When a node receives an `INSERT` with the `NO_REPLICATE` flag set, it fetches the `<data-name>` packet directly from the client and stores it in the storage backend. Since all replicas fetch the object together, only one request is forwarded to the client due to Interest aggregation and in-network caching. On successful completion, the storage node replies to the insertion Interest with a positive acknowledgement, indicating the object was inserted successfully. The node receiving the original `INSERT` from the client tracks acknowledgements from all replicas. When all acknowledgements have been received, it replies to the Interest sent by the client with an acknowledgement, completing the insertion process. Thus, an acknowledgement received by the client guarantees that the object is inserted and replicated in the system, thus providing strong consistency. If a replica does not respond to an insertion Interest before the Interest timeout, the client's Interest is also allowed to timeout, following which the client may retry the request. The insertion process has been illustrated in Fig. 4.

Object Retrieval To fetch an object from Kua, the client sends an Interest for the original data name[7] of a stored segment. The Interest is routed to the Kua cluster using a Forwarding Hint, carrying the cluster prefix and the bucket identifier, which the client can compute since it knows the name of the data. Thus, a data fetching Interest takes the form of:

```
Name: /<data-name>/
FwHint: /<kua-prefix>/<bucket-id>
```

NDN forwards such an interest to the nearest node in the cluster that is a replica for the corresponding bucket. Since this node is guaranteed to have a copy of the data if it was inserted, the node receiving the fetching interest looks up its local storage backend and replies to the Interest if the data is found. Since the name of the stored Data packet matches the name of the `FETCH` Interest, NDN directly forwards it back to the client(s) requesting the data. If the requested data is not found, the node replies with an application layer NACK containing an error code.

The nature of the object fetching protocol has three properties. First, since all objects are fetched using the actual data names of the packets, the network can cache packets with semantically correct names. As a result, any Interests for the data intended for the producer application may also be satisfied by the network cache if the same data was previously fetched from a Kua cluster. Interests to fetch the same object are also aggregated in the network, leading to efficient multicast of data. Second, since NDN forwards the fetching interest with anycast to the nearest node which, in the absence of failures, is guaranteed to have the corresponding object, latency of fetching an object is kept at a minimum. The client is guaranteed to start receiving the object in one network round trip if it exists, without the need of extra metadata queries. Third, since data packets are stored directly, the corresponding signatures are also stored and returned in replies to `FETCH` Interests, ensuring that data is always secure in transit and at rest.

[7]The means for obtaining this name is left to the application or Sync transport.

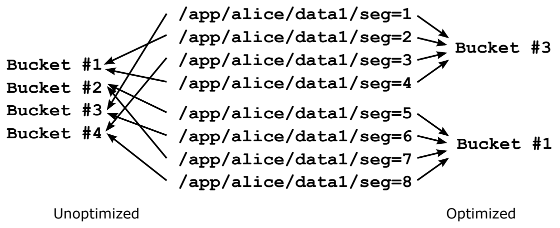

Figure 6: Chunk Optimization

Reclaiming Space While Kua is an immutable store, practical applications require the ability to reclaim space from unused objects. To delete an object no longer needed by the application, the client may send a `FREE` Interest, which is then propagated to replicas similar to an insertion Interest. Such an Interest is signed and contains the name of the object to be deleted.

```
/<kua-prefix>/<bucket-id>/FREE/<data-name>
```

It must be noted here that object deletion may be asynchronous, and reusing keys may lead to undefined behavior. Further, a deleted object may continue being available to clients by virtue of being cached in the network.

Client API The Kua library provides simple primitives for applications to interact with the data store, masking operations such as calculating the bucket number. The object insertion API accepts a binary blob and an object name, along with a data signer, and inserts the object into the store. Similarly, the object retrieval API accepts an object name and data validator, and returns the binary blob from the store. These familiar primitives enable application developers to easily interact with the data store.

4.4 Optimizations

In NDN, large data blobs are segmented into smaller Data packets and named using the NDN segment naming convention. As a result, large objects are uniformly distributed across all nodes in the Kua cluster, leading to faster read and write speeds, similar to existing systems such as Ceph [24]. However, since each packet is fairly small, objects may become highly fragmented and require a large number of random lookups during access. Storing asymmetric signatures with each packet also adds to the storage overhead. Further, having to insert each segment individually with an acknowledgement as described earlier can result in a large volume of control messages, leading to poor performance and issues with congestion control. To counter these effects, we implement a few optimizations.

Chunk storage To reduce random accesses to disks, we tweak the consistent hashing function to ensure that segmented objects are stored in larger chunks rather than packets, as illustrated in Fig. 6. With an assumption that that all objects are segmented using

NDN naming conventions, the hash function is modified to generate the same hash for a range of contiguous segment numbers. This enables optimizations to reduce the number of random lookups to disk that must be made for fetching a single object, by caching locations of recently fetched chunks.

Fast Insertion To mitigate the high overhead of insertion Interests required to insert large objects, we allow multiple[8] segments to be inserted with a single request. With the `IS_RANGE` flag, a single `INSERT` Interest may insert multiple objects belonging to a single large segmented object, and receive a single acknowledgement. Such an Interest includes start and end segment numbers to be inserted along with the name prefix of the data excluding the segment number. It must be noted here that the `FETCH` protocol still retrieves objects one segment at a time. This does not have performance implications, since it allows direct usage of existing and future NDN congestion control solutions at the transport layer. We recognize that reads may be further optimized, e.g. with in-memory caching of data for expected future queries for sequential reads.

Signing Manifests Producers may use signing manifests to reduce the overhead due to asymmetric signatures on individual packets. When using manifests, each packet only contains a digest of its contents that can be directly verified. The digests of multiple such packets are enlisted in a special packet called a *signing manifest*, and this packet is then signed with the private key of the producer. As a result, the processing and storage overhead due to packet signing can be significantly reduced.

Our preliminary evaluation showed us that these optimizations can have a large impact on the performance of Kua.

5 EVALUATION

We implemented Kua in C++ to cater to the requirements of high performance. The implementation uses the ndn-cxx library and implements storage functions, the Auction protocol as well as some of the optimizations described earlier. Our implementation allows usage of any key-value storage backend for the storage of actual data, such as in-memory, file or database storage.

We evaluate Kua by comparing it with two very popular and successful distributed storage systems – Hadoop Distributed File System (HDFS) and Redis Cluster. By highlighting the differences between the systems in the context of efficient communication, we demonstrate how NDN helps Kua achieve its function with relatively lower overhead.

5.1 Comparison with HDFS

We note here that while HDFS is a file system rather than an object store, it serves as a good general comparison for distributed storage due to its popularity and simple design. We only perform an analytical comparison of the Kua and HDFS designs here.

HDFS works on a master/slave architecture. A single central node called the NameNode acts as a master that manages the filesystem namespace and access control. Each storage node is called a DataNode and stores the actual contents of the data. The NameNode executes namespace operations such as opening and closing files, and stores the mapping of the data blocks to the DataNodes.

[8]The number of segments inserted with one request should not be too large, and can be controlled with an algorithm similar to congestion control.

For replication, HDFS divides files into equally sized blocks. These blocks are then replicated across multiple nodes. The NameNode centrally determines which blocks go to which DataNode during replication. The NameNode tracks state at the DataNodes through a period heartbeat and block report, containing a list of all blocks stored by the DataNode.

The simple design of HDFS by virtue having a central master node comes at a cost – all file operations must pass through the NameNode. When a client wants to write to a file, it first needs to obtain an exclusive lease on the file from the NameNode, which must be periodically renewed. The client can then directly interact with the DataNode to write content to the file blocks, which are chain replicated. If a new block is created, each replica DataNode must inform the NameNode about creation of the new block, so the NameNode can maintain a consistent global state.

One obvious disadvantage of having a master node is that it becomes a single point of failure and a potential performance bottleneck. If the NameNode of Hadoop fails, the cluster cannot function until the NameNode is restarted. In contrast, Kua does not maintain any global state. Since the network automatically routes requests to the correct storage nodes, it eliminates the need for a central master node that keeps track of what data is stored where. If a node fails then the rest of the system can continue functioning in an alarm state, e.g. having only two replicas for a few buckets, until the system is re-balanced. Further, most read requests will continue to succeed if a node fails, since only the the clients closest to the failed node will be affected. As soon as the failure is detected and routing updates propagate through the network, requests from these clients will also be forwarded to alternative storage nodes. Further, chain replication in HDFS happens over point-to-point TCP/IP connections between replicas, increasing outgoing traffic at each node. Replication of data is much more efficient in Kua, since it uses NDN's multicast data delivery to directly fetch data from the client at each replica.

Another advantage of the Kua design is the lack of redirects. To access a file in a HDFS cluster, a client must first open a TCP connection to the NameNode, request the location of the blocks, and subsequently open more TCP connections to the DataNodes. The latency may be increased further if the protocol requires any form of authentication after the connections are opened. In Kua, the network directly forwards requests for data to the storage node containing the data. As a result, the overall latency of Kua for fetching a file can be expected to be much lower than HDFS, especially when the dataset has a large number of small files.

The above comparison demonstrates how Kua can perform better than HDFS due to NDN's data-centric properties. Next, we compare the Kua design with Redis Cluster, a distributed version of the popular in-memory store, Redis.

5.2 Comparison with Redis Cluster

Redis [3] is a very popular in-memory key-value data store known for its very high performance. Redis Cluster [1] is a distributed implementation of Redis for horizontal scaling, which distributes and replicates data across multiple nodes. Data is sharded by hashing the key and distributing keys among 16384 hash slots. Each hash slot is assigned to one master and one or more slave nodes, and data

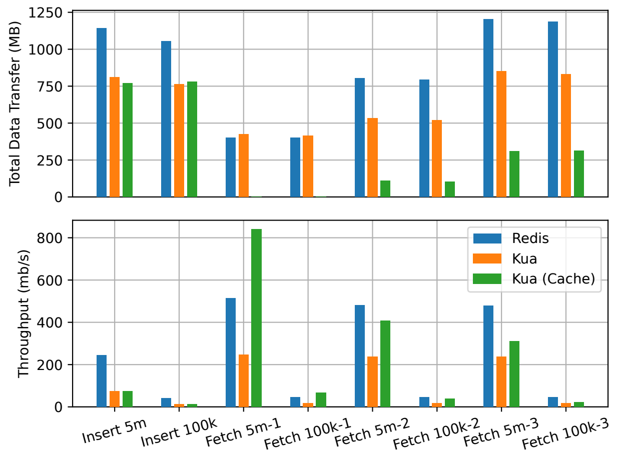

Figure 7: Comparison of Kua with Redis Cluster

is replicated from the master to slave asynchronously. To insert or retrieve data, users may contact any node in the cluster, and are redirected to the correct node for the specified key by the contacted node. Redis Cluster employs $N \times N$ TCP connections for internal communication using a gossip protocol.

The design of Kua has several similarities to Redis Cluster. Both storage systems use semantically meaningful identifiers to store and retrieve data, and distribute data between nodes using a hash function. However, since Redis runs over TCP/IP, all data is transferred over point-to-point connections. Unlike Kua, Redis also requires redirects to other nodes for reads and writes. Clients opening connections to each node they need to access data from, along with the internal $N \times N$ connections may make scaling challenging.

We evaluated Kua against Redis by inserting and fetching objects from both stores in an emulated topology over MiniNDN [11]. We used a datacenter-like topology for the evaluation, with nodes connected to rack switches, which are interconnected by multiple core switches. Each “switch” and storage node was configured to run instances of the NDN Forwarding Daemon (NFD) and the NDN Link State Routing (NLSR) [23] daemon for forwarding and routing respectively. We set the link delay to 2ms for all links, with no bandwidth limits or link losses. Both data stores were configured to replicate each piece of data three times across 9 nodes, connected to switches in groups of three. The Kua cluster was configured to use a total of 32 buckets.

We ran three experiments for the comparison. In each experiment, 100MB data was inserted into both stores. The data was inserted as objects of 5MB and 100KB size in separate runs. In the first experiment, we fetch the data from the client that inserted the data. In the second and third experiments, we simultaneously fetch the data from one and two other clients in the network respectively, along with the client that originally inserted the data. The results in terms of total network traffic are shown in Fig 7. For Kua, the orange bars represent results after clearing the network cache at all forwarders between operations, while the green bars represent results without clearing the cache.

We observe that for the same degree of replication, Redis Cluster needs to transfer approximately 45% extra data for insertions compared to Kua. This is a result of data multicast in NDN – since each Kua replica fetches the same data at approximately the same time, the Interests are aggregated inside the NDN network, reducing total traffic. A similar effect can be observed when the data is simultaneously fetched by multiple clients. Since all clients send fetch Interests with the same name, data is multicast from the storage to the clients, reducing network traffic. We note that fetching overhead is very low when the caches are not cleared between operations, since almost all data can be cached in the network. It can also be noted that all NDN data is signed, while Redis does not provide any form of security in this experiment.

Finally we observe that Redis provides 2-5x throughput compared to Kua. However, we note here that Redis is a very mature system known for its high performance, while Kua is still in early stages of development. We can also attribute the lower throughput of Kua to the processing overhead of emulating a large number of NFDs, the lack of mature congestion control in NDN and the additional guarantees of strong consistency provided by Kua. At this stage, we consider being able to provide throughput of similar orders of magnitude compared to Redis as encouraging.

6 DISCUSSION

In this section, we present further discussion on some issues, including the security model of Kua, functions of locating data and some related work.

6.1 Data-Centric Security

In existing data store designs over TCP/IP, authenticity of data is typically established by verifying the identity of the node transmitting the data. This is achieved by securing the transport pipe using TLS, and verifying the certificate provided by the data store using public key infrastructure. Such a model implicitly requires data consumers to trust the data store and its operators, ultimately providing limited guarantees about authenticity of the data.

The design of Kua allows for direct usage of existing NDN security solutions. Since every Data packet generated by the producer application is directly stored and returned in response to requests, consumers can directly verify fetched data using the producers’ certificates. Since certificates in NDN are also simply Data packets, these may also be inserted into the Kua store, enabling offline access. Such a model does not need to secure transport channels, and consequently eliminates the implicit trust relationship between consumers and the data store.

Kua can also be used with Name-based Access Control (NAC) for confidentiality and access control. Producers can encrypt objects before they are inserted into the store, and only authorized consumers can decrypt these objects using keys provided by NAC. As a result, all data is end-to-end encrypted, providing confidentiality both during transit and at rest. With Attribute-Based Encryption (NAC-ABE), access control can also be scaled to large groups of consumers while providing fine-grained access to data. We omit further details regarding the usage of NDN security solutions with Kua in this paper due to space limitations.

Kua also provides a starting point for enforcing accountability and resource allocation through the use of signed insertion and deletion Interests. We consider the design of these features as part of our future work.

6.2 Locating Data in Distributed Storage

Many existing data storage systems use some form of a global index to locate data. Such an index may be stored in a single master node or distributed inside the entire system. When users need to access data in the store, they must first look up this index to locate the data, and then open separate channels to the storage nodes to fetch the actual data pieces.

In the Kua design, because NDN directly use application derived names in network routing and forwarding, Kua can directly forward requests towards the nodes which contain desired data, without reliance on any redirection. This design eliminates the need for an index to locate objects, and reduces overhead and complexity. As a result, Kua enables users to fetch data from the store with low latency, while requiring almost no configuration. Furthermore, since Kua routing is based on object identifiers, and nodes holding replicated objects announce the same identifiers, all object requests are anycast, which improves availability of the data store.

We note here that the performance and reliability of such a design is dependent on routing support at the network layer, including low propagation delay for failover and routing scalability for a large number of buckets. The interplay between routing and Kua's performance needs further investigation.

6.3 Failure Recovery

The bidding function described in (1) provides a starting point for engineering failover functions using the Auction protocol. We describe two scenarios requiring failover support and describe the intended behavior of the Auction protocol.

In the first scenario, a node fails completely. We note here that first, all write requests for buckets stored by the node would now fail due to the missing replica, but read requests would still succeed, especially as soon as routes to the failed node expire. On detecting the failure, if the remaining replicas can achieve quorum, they may continue accepting write requests in a compromised state[9]. Next, the currently elected master would initiate a new auction for the buckets in the node. The new node winning the bucket must indicate in the `WIN_ACK` message that it needs to transfer data from the bucket. The master would then indicate the same in the `AUCTION_END` message (Fig. 3).

The nodes storing the bucket (which are already authoritative for this bucket) would now take a snapshot of the currently stored data in the bucket[10]. The new replica can then start asynchronously fetching this snapshot to reconcile the existing data in the bucket. Simultaneously, the new node also registers the bucket route with `<node-prefix>`, but not the route with `<kua-prefix>` (§4.3). Since the registered route enables internal replication requests, new write requests to the bucket can now succeed, with the replicas including the new node. However, since the node does not register the `kua-prefix` route, it does not handle any read requests from clients. After the data transfer completes, the new node becomes fully capable of handling read requests, and thus becomes authoritative for the bucket by registering the `<kua-prefix>` route. It can then signal the other nodes to delete the snapshot, thus completing the failover process and restoring steady state.

In the second failover scenario, a node may voluntarily give up a bucket due to lack of resources or the need for maintenance. In this case, failover proceeds similarly to failures, and the node lacking resources can free up the space used for the bucket as soon as the new node becomes authoritative.

This failover mechanism ensures that every piece of data that was successfully written to the store is consistent across all replicas, and handles all potential race conditions. Since each write requires all three replicas to provide confirmation, new writes to the bucket can only succeed after the new node starts accepting write requests, including route registration and propagation. As a result, the snapshot will also necessarily contain all pieces of data from writes completed before the new node started accepting write requests. Consequently, the entire bucket will be consistent once the snapshot is fetched, regardless of the order in which the messages are received and routes propagated.

We note that the above design serves only as a starting point for implementing failover, and more investigation is required to understand the nuances of failover and the behavior of the Auction protocol under various conditions such as network partitions. We consider this investigation as a part of our future work.

6.4 Related Work

We briefly compare Kua's design with other storage systems built over NDN.

Repo Several designs of storage repositories have been implemented over NDN, such as repo-ng [20] and fast-repo [21]. However, these repositories are not designed as distributed systems, and thus their designs cannot be directly compared to the Kua design.

Object Storage The most notable attempt to build an object store over NDN is Chipmunk [18]. Kua can potentially achieve superior performance compared to Chipmunk due to its simpler protocol and lack of metadata fetching. Unlike Chipmunk, Kua leverages the network to forward requests to the nearest nodes that can handle the requests, reducing latency and improving throughput. Kua also provides strong consistency and a replication mechanism.

7 CONCLUSION & FUTURE WORK

We presented the design and evaluations from an initial version of Kua, a distributed object store running over NDN. Kua directly forwards user requests to the correct storage nodes through the use of consistent hashing of data names, improving efficiency and reducing system complexity. Kua is designed to provide high performance, strong consistency, and fault tolerance through replication. The data-centric nature of NDN also simplifies security in the data store, enabling applications to provide strong guarantees about the authenticity of data.

Our future work includes the implementation of resilient failure recovery, integration of security frameworks, a highly usable API to enable application development, as well as further investigation into efficient and robust storage backends. With these improvements, we aim to make Kua a fully functional and robust object store to cater to today's big data storage needs. The open source implementation of Kua can be accessed at GitHub [22].

[9]The replicas are required to exit the compromised state and stop acknowledging write requests with fewer replicas before bidding in the next auction for the bucket.
[10]Snapshot performance may be improved using filesystem support.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and the shepherd Marc Mosko for their valuable comments which helped us improve the paper's quality significantly. This work was supported in part by National Science Foundation under awards 2019085 and 2126148.

REFERENCES

[1] 2022. Scaling with Redis Cluster. https://redis.io/docs/manual/scaling/. Accessed: 2022-05-18.

[2] Alexander Afanasyev, Tamer Refaei, Lan Wang, and Lixia Zhang. 2018. A Brief Introduction to Named Data Networking. In *Proc. of MILCOM*.

[3] Josiah Carlson. 2013. *Redis in action.* Simon and Schuster.

[4] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. 2005. Object storage: the future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology.* 119–123. https://doi.org/10.1109/LGDI.2005.1612479

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles.* 29–43.

[6] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. 1997. File Server Scaling with Network-Attached Secure Disks. *SIGMETRICS Perform. Eval. Rev.* 25, 1 (jun 1997), 272–284. https://doi.org/10.1145/258623.258696

[7] Developer Guide. 2008. Amazon Simple Storage Service. (2008).

[8] John H. Howard. 1988. On Overview of the Andrew File System. In *USENIX Winter.*

[9] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[10] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).* USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[11] Mini-NDN Authors. 2021. Mini-NDN: A Mininet-based NDN emulator. minindn.memphis.edu/ accessed: 2021-05-10.

[12] Philipp Moll, Varun Patil, Lixia Zhang, and Davide Pesavento. 2021. Resilient Brokerless Publish-Subscribe Over NDN. In *MILCOM 2021 - Special Session on Named Data Networking (MILCOM 2021 - NDN Session).* San Diego, USA.

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* 561–577.

[14] NDN Project team. 2021. NDN Packet Format Specification version 0.3: Signed Interest. (2021). https://named-data.net/doc/NDN-packet-spec/current/signed-interest.html accessed: 2021-07-29.

[15] Jon Postel. 1981. *RFC793: Transmission Control Protocol.* Technical Report.

[16] Russel Sandberg. 2000. The Sun Network File System: Design, Implementation and Experience. (09 2000).

[17] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. 2020. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *Proceedings of the 7th ACM Conference on Information-Centric Networking* (Virtual Event, Canada) *(ICN '20).* Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/3405656.3418715

[18] Yong Yoon Shin, Sae Hyong Park, Namseok Ko, and Arm Jeong. 2020. Chipmunk: Distributed Object Storage for NDN. In *Proceedings of the 7th ACM Conference on Information-Centric Networking* (Virtual Event, Canada) *(ICN '20).* Association for Computing Machinery, New York, NY, USA, 161–162. https://doi.org/10.1145/3405656.3420231

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST).* Ieee, 1–10.

[20] NDN Project Team. 2014. *repo-ng.* https://github.com/named-data/repo-ng

[21] NDN Project Team. 2018. *Fast Repo.* https://github.com/remap/fast-repo

[22] The Kua Team. 2022. *Kua: Distributed Object Storage over Named Data Networking.* https://github.com/pulsejet/kua

[23] Lan Wang, Vince Lehman, A. K. M. Mahmudul Hoque, Beichuan Zhang, Yingdi Yu, and Lixia Zhang. 2018. A Secure Link State Routing Protocol for NDN. *IEEE Access* 6 (2018), 10470–10482. https://doi.org/10.1109/ACCESS.2017.2789330

[24] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06).* USENIX Association, USA, 307–320.

[25] Huaxia Xia and Andrew A. Chien. 2007. RobuSTore: a distributed storage architecture with robust and high performance. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing.* 1–11. https://doi.org/10.1145/1362622.1362682

[26] Zhiyi Zhang, Yingdi Yu, Sanjeev Kaushik Ramani, Alex Afanasyev, and Lixia Zhang. 2018. NAC: Automating Access Control via Named Data. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM).* 626–633. https://doi.org/10.1109/MILCOM.2018.8599774