Synchronizing Namespaces with Invertible Bloom Filters [*]

Wenliang Fu[+], Hila Ben Abraham[*] and Patrick Crowley[*]
[+]School of Computer Science, Beijing Institute of Technology
[*]Computer Science and Engineering, Washington University
fuwenl@bit.edu.cn, {hila, pcrowley}@wustl.edu

ABSTRACT

Data synchronization–long a staple in file systems–is emerging as a significant communications primitive. In a distributed system, data synchronization resolves differences among distributed sets of information. In named data networking (NDN), an information-centric communications architecture, data synchronization between multiple nodes is widely used to support basic services, such as public key distribution, file sharing, and route distribution. While existing NDN synchronization schemes are functional, their implementations rely on log-based representations of information, which creates a limitation on their performance and scalability. This paper presents iSync, a high performance synchronization protocol for NDN. iSync supports efficient data reconciliation by representing the synchronized datasets using a two-level invertible Bloom filter (IBF) structure. A set-differences can be found by subtracting a remote IBF from a local IBF. The protocol can obtain multiple differences from a single round of data exchange, and does not require prior context in most application scenarios. We evaluated iSync's performance by comparing it to the CCNx synchronization protocol. Experiments show that iSync is about eight times faster across a range of network topologies and sizes, and that it reduces the number of packets sent by about 90%.

Categories and Subject Descriptors

C.2.2 [**computer-communication networks**]: Network Protocols—*Applications*

Keywords

Named data networking; high performance; data synchronization; invertible Bloom filters

1. INTRODUCTION

Many of today's popular applications, such as cloud storage, group communication, media sharing, and even routing protocols, use data synchronization as their fundamental and core operation.

In named data networking (NDN) [1], data is represented by a namespace, similar to the representation of URIs in HTTP. A content item (data) is stored, requested, and retrieved by specifying its namespace. A namespace can represent any type of data, such as file name, application state, chat message, or a video chunk. Keeping namespaces synchronized has recently emerged as a basic service required by many NDN applications, from Dropbox-style file sharing to supporting mobile and ad-hoc vehicular communication [2]. Moreover, data synchronization serves as a core service to exchange public keys in the key-based trust model used by the NDN architecture.

Keeping a dataset up-to-date among distributed participants, the essential requirement of a synchronization protocol, is achieved by replicating the dataset or the status of the dataset among participating hosts. An easy example of a synchronized dataset is all the files in a Dropbox [3] directory. In this example, a replication of each file is created and kept up-to-date among the user's devices and a Dropbox server. Another example could be the chunks of a shared video file. In this example the synchronization protocol must find the existing and missing chunks of each participant, and the up-to-date dataset must consist of the complete list of all the chunks found in possessions of the participating parties.

We can describe data synchronization as three main tasks: 1) Understanding whether a set is up-to-date or out-of-date, 2) finding set differences, and 3) retrieving missing items. The second task lies at the heart of the synchronization process.

The invertible Bloom filter (IBF) was recently proposed in [4, 5]. IBF provides a theory and foundation for a high performance and scalable data synchronization protocol. In brief, IBF is a hash-indexed, redundant table in which each item is hashed into at least two cells. It supports item insertion, deletion, and limited inversion (i.e., retrieval of stored keys). The ability to retrieve the stored items differentiates IBF from other compact set representations. Moreover, the difference set of two IBFs can be obtained by subtracting their respective IBF tables.

This paper presents iSync, an IBF-based synchronization protocol. iSync uses this unique subtraction operation to discover multiple differences in a single subtraction, which leads to an efficient implementation with a low computation time and a small communication overhead. iSync can be used as a synchronization service by multiple applications at the same time.

Several limitations of IBF must be overcome to use it as the data structure for a synchronization protocol:

- The original IBF design has limited support for membership queries.
- There is a strong relationship between IBF capacity and decoding inaccuracy [5]. A control mechanism is required

[*]This work was done at Washington University in St. Louis. Hila Ben Abraham is the corresponding author.

978-1-4673-6633-5/15/$31.00 © 2015 IEEE

to ensure an acceptable balance between size and decoding accuracy.

- As with all hash-based structures, hash collisions must be mitigated.

To address those limitations, iSync consists of a hierarchical encoding and recording scheme, a size-difference control scheme, and a recovery scheme. The main contributions of this paper are summarized as follows:

- We present a practical IBF design which addresses the challenges articulated above. Our design consists of a two-level IBF structure, aimed at supporting efficient data reconciliation of multiple applications. The first level finds which datasets have updates; the second level reconciles the differences among the changed datasets. This feature allows the difference reconciliation process to efficiently skip datasets that have no updates.

- We describe and evaluate a functional prototype that executes in CCNx-compatible environments. The evaluation compares the performance of the CCNx Sync and iSync protocols.

While the iSync protocol can be used generically, this paper is mainly focused on data synchronization in the NDN context. Compared with the current synchronization schemes used in CCNx [6], iSync does not require prior context in most application scenarios, and it scales well.

The remainder of this paper is organized as follows. The next section introduces background and related work. Section 3 presents a detailed design of iSync. Section 4 provides the evaluation results and comparisons of iSync and CCNx Sync. Section 5 summarizes our work and discuss future developments.

2. BACKGROUND AND RELATED WORK

As mentioned in the introduction, the main task of a synchronization protocol is to identify the set-difference of two remote datasets. Data synchronization is widely used and plays an important role in traditional and emerging network premises. Different approaches to identify the set-difference are taken by different applications. The well known rsync [7] was the first to suggest a solution that does not transfer the complete dataset over the network but send only the deltas instead. Rsync is a pairwise algorithm that synchronizes remote files and directories by sending only the missing file chunks. Rsync discovers the set-difference by calculating the block checksums of a synchronized file on one host, and sending the list of the calculated checksums to the remote host. The remote host goes over its local copy of the file and compares its local checksums to the received list. Then rsync identifies the missing blocks in its local file and requests those blocks from the remote host.

Another common approach to identify the set-difference is a timestamped log. One host notes the changes to its local dataset in a local timestamped log, and therefore the set-difference can be identified by transferring the log notes added after the last synchronization cycle. While log-based synchronization solutions are practical and easy to implement, their performance dramatically decreases when the number of parties scales up, due to the complexity of transferring, parsing, and comparing multiple files in every sync cycle.

A recent academic work takes the information-centric approach and presents a new scalable routing model, named Custodian-Based Information Sharing (CBIS) [8]. CBIS presents a synchronization model that doesn't rely on a centralized server and can be used as a synchronization service to share data. CCNx sync protocol and ChronoSync [9, 10] are additional information-centric synchronization protocols that do not rely on a centralized server. The CCNx synchronization protocol, which serves as our baseline, will be described in detail at the end of this section.

ChronoSync is a distributed synchronization protocol for NDN that synchronizes the knowledge about the status of the datasets, and leaves the application to decide whether to fetch the new data. ChronoSync uses a digest tree to represent the application dataset, and a digest log to store previous digests of this dataset. While both ChronoSync and our proposed iSync implement a synchronization protocol over the NDN network, their infrastructures are fundamentally different. ChronoSync uses hash trees based on the concept of Merkle trees [11], while iSync uses IBFs [4] to store and reconcile the declared datasets.

Additional research efforts have focused on synchronization protocols such as surveys [12] and on the synchronization of two nodes in scenarios of a small set of differences [13]. Most of the recent works in this field are well-known commercial projects such as BitTorrent Sync service [14], DropBox [3], and Google Drive [15].

The rest of this section is organized as follows: We first present the basic characteristics of Named Data Networking. We continue with a description of the invertible Bloom filter, and conclude with an overview of the CCNx Sync protocol, our baseline.

2.1 Named Data Networking

Named Data Networking (NDN) [1] is one of four projects funded by the NSF under the Future Internet Architecture program [16]. Unlike the IP paradigm, NDN focuses on what the content is and not on where the content is located. In other words, in the NDN architecture, a packet is delivered according to its name rather than its destination address.

The NDN architecture introduces two types of packets: Interest packet and Data packet. The Interest packet is a pull request sent by a consumer to request content for a specific name. The Data packet carries the content, and is generated by the producer as a response to an Interest packet. Data packets are cached in every NDN node and can be used to satisfy future pull requests for the same name. Upon receipt of an Interest packet, the NDN node looks in its Content Store (CS) to see if it can satisfy the request using previously cached information. In case the CS does not hold the content, the node forwards the Interest to its next hop by looking up the name in its Forwarding Information Base (FIB) table. In addition to forwarding the Interest, the node notes the forwarded Interest packet and its incoming face in the local Pending Interest Table (PIT). Once the Interest packet arrives at a node that can satisfy the requested name from cached content or a producer, it replies with a Data packet. The Data packet is forwarded to the consumer by following the reverse path of the Interest packet. Forwarding can be done by using the information stored in the PIT. If more than one consumer asks for the same data, the PIT entry of the requested name will contain the faces of all the requesting consumers. Therefore, on its way back, the Data packet will be forwarded to all the requested faces. Additionally, the NDN nodes on the forwarding path store a copy of the forwarded Data packet in their Content Store (CS), and use it to satisfy future Interest packets expressed for the same name.

2.2 Invertible Bloom Filter

The invertible Bloom filter (IBF), introduced in 2011 as an extension of the original Bloom filter [4], is a simple and space-efficient probabilistic data structure. The Bloom filter answers whether an item exists in a dataset or not. The implementation of the Bloom filter consists of a simple array and a number, k, of hash functions. The k functions map an item into k cells in the array. The mapped cells are marked as occupied upon the insertion of an item. A Bloom filter tests for the existence of an item by looking into its hashed cells, and answers yes if all the cells are marked as occupied. The original design of the Bloom filter does not support either deletion of an item or querying the stored items.

IBF was designed to address those limitations. As in the counting Bloom filter, IBF uses *count* to indicate the number of items that have been indexed to each cell. In addition, the IBF algorithm introduces two new values in each cell: *idSum* and *hashSum*, to represent key-value pairs in each cell. The insertion process of an item is similar to the original insertion process of the Bloom filter. A set of k hash values is generated to map the item into k cells in the array. However, unlike the standard Bloom filter, the value of the inserted item is added, using a XOR operation, to the value of *idSum* in each of the mapped cells. Also, another hash function adds the hash value of the inserted item to the value of *hashSum* in each of these cells. This addition is also achieved by XORing the hash value of the inserted item with the value of *hashSum*. An item insertion incremens *count* in each of the k mapped cells.

In a similar way, we can delete an item from an IBF by subtracting it and its hash value from *idSum* and *hashSum* respectively, and by decrementing *count* in each of k hashed cells.

We can retrieve the items from an IBF by looking for pure cells. A pure cell is a cell that contains only one item, and the hash value of the cell's *idSum* equals the value of *hashSum*. To discover additional pure cells, and therefore additional stored items, we subtract the items found in the pure cells from the other cells indexed to store those items The subtraction process can be repeated as long as there are pure cells to retrieve.

IBF not only supports insertion and deletion of items, but also can obtain the difference set of two IBFs by subtracting one IBF from another, followed by decoding the resulting IBF to retrieve the stored items.

We present an example to illustrate the encoding process of an IBF: Upon the insertion of a new element S into a cell i, the value of *idSum* of the ith cell is XORed with S, while the value of *hashSum* is XORed with the hash value $H(S)$. Given two IBFs, A and B, we can compute the set difference (A - B) by XORing the values of *idSum* and *hashSum*, and decreasing the value of B's *count* from the value of A's *count*. In case the set includes at least one *pure* cell, we can start the encoding process and retrieve the elements in the difference set. Again, the pure cell must satisfy two requirements: Its *count* value must be equal to 1 or -1, and the hash value of its *idSum* must be equal to its *hashSum*. The encoding process can list the IBF's elements as long as there are pure cells to subtract from other cells.

2.3 CCNx and the CCNx synchronization protocol

The content-centric-networking project (CCNx)[6] is an open source implementation of the NDN architecture. The CCNx consists of two main components: the CCNx deamon (ccnd) and the CCNx repository (ccnr). The ccnd implements the forwarder as well as the FIB, PIT, and CS infrastructures. The ccnr implements the CCNx repository, which can be used by the network or by an application to preserve required data, such as a routing table, file contents, or an application state. In this work, we compare the performance of iSync to the performance of the *CCNx Synchronization protocol* (CCNx Sync).

The CCNx Sync protocol defines a collection as a set of content items, all sharing the same name prefix. The protocol operates between two neighbor nodes that declared the same collection, and keeps the collection up-to-date by synchronizing the differences. For synchronization, CCNx Sync uses a tree based structure called the *Sync Tree*. A single *Sync Tree* represents the prefixes of all the content items in a single collection.

Upon the addition of a content item to the CCNx Repository, ccnr checks if the content name answers the definition of an existing collection. If it does, then the content's name is added to a *Sync Tree* leaf of the corresponding Sync collection. A hash value is computed for each of the inserted names. Thus, each leaf holds a list of content names and a combined hash representing the arithmetic sum of the local names' hash values. The other *Sync Tree* nodes hold the combined hashes of their children. Using this tree structure, the root node of each *Sync Tree* holds the combined hash of all the names stored under the represented collection.

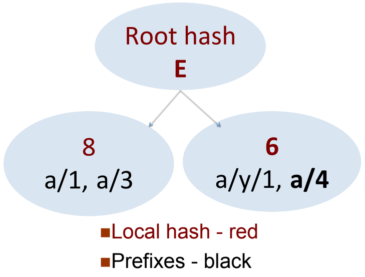

Figure 1: CCNx Sync Tree.

Fig. 1 presents an example of a *Sync Tree* containing four NDN style names: *a/1*, *a/3*, *a/y/1*, and *a/4*. In this example, the first two names compute a combined hash of 0x8, while the last two names compute a combined hash of 0x6. The root hash in our example is 0xE. Fig. 2 shows the protocol timeline upon the insertion of *a/4* into Alice's collection. Thus, Alice's sync tree of a specific collection includes the inserted name *a/4*, while Bob's collection is represented by the same tree without *a/4*. To keep the collection up-to-date, Alice sends a periodic *Root Advise* interest to Bob, including the collection name and its root hash. Upon reception of a *Root Advise* interest, Bob compares its local root hash with the remote node root hash. Equal root hashes imply the collection is up-to-date in both nodes, while dif-

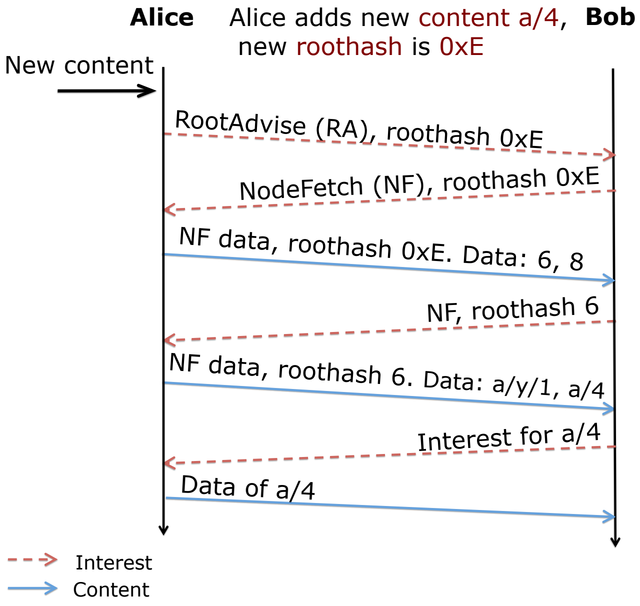
Alice
Alice adds new content a/4, new roothash is 0xE
Bob
New content
RootAdvise (RA), roothash 0xE
NodeFetch (NF), roothash 0xE
NF data, roothash 0xE. Data: 6, 8
NF, roothash 6
NF data, roothash 6. Data: a/y/1, a/4
Interest for a/4
Data of a/4
Interest
Content

Figure 2: CCNx Sync Timeline.

ferent hashes imply a collection difference. In our example, Bob understands that its local collection is out-of-date. To reconcile the differences, Bob sends a *Node Fetch* interest that contains the collection name as well as the unrecognized hash value. Alice responds with the children's hash values list of the unrecognized hash. In our example the children of 0xe are 0x8 and 0x6. Bob recognizes 0x8, but does not recognize 0x6, and therefore sends an additional *Node Fetch* interest to requests for the content of 0x6. Now, the unrecognized hash represents a leaf in Alice's sync tree, and therefore the list of the names stored in that leaf is sent as the data response. Here, Alice responds with *a/y/1* and *a/4*. At this point, Bob understands that the set-difference of its collection is *a/4*. The remote and the local nodes can exchange *Node Fetch* interests and data packets until the data packet includes the list of all the missing contents names and, hence, the set difference. Once a node reconciles the names, it sends a regular interest packet to fetch the content of the reconciled names. In this example, seven packets are sent to reconcile the addition of a single name. An update to another collection will result in an additional set of packets as described in Fig. 1.

It's important to note that the names described in the example were simplified to present names that share the same *a/* prefix. In reality, the prefix used by a CCNx Sync collection consists of an additional component to indicate the forwarding routes.

3. PROTOCOL DESIGN

In this section, we describe the detailed design of iSync. Sections 3.1 and 3.2 introduce the synchronization model and describe the designed data structure. In section 3.3 we show how iSync encodes variable size names into fixed size file IDs by using an encoding and recording scheme. Section 3.4 describes the size-difference control mechanism. We describe the recovery scheme in section 3.5.

3.1 Data Synchronization Model

As shown in Fig. 3, iSync consists of a repository and a sync agent. iSync can be regarded as an additional data synchronization layer in the CCNx stack, and can serve as a synchronization service to CCNx applications. The repository offers an interface for CCNx entities (i.e., applications) to insert files and publish sync collections. Like in CCNx Sync, iSync defines a collection as the set of content items sharing a common prefix. To synchronize a collection, an application is require to declare the same collection in each of the participating nodes. An example of a collection can be a set of music items that share a common prefix, such as John'sDocuments/Music. Possible content items in such a collection might include
John'sDocuments/Music/MichaelJackson/song1 or
John'sDocuments/Music/BestOf2010/song32. iSync repository provides an API to declare a sync collection. Applications can declare multiple sync collections, which will be synchronized independently.

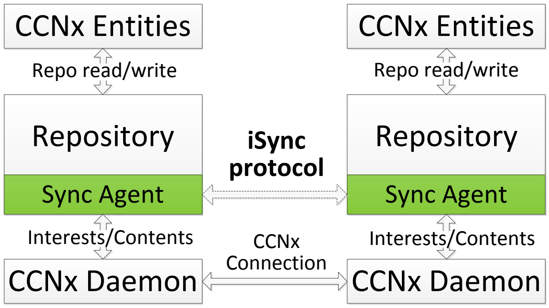

Figure 3: Data Synchronization Model in Named Data Networking Paradigm.

The iSync repository notifies the sync agent when a new content name matches one of the local declared collections. Then, the sync agent indexes the inserted content name and updates a digest that reflects all the names of the collection. The sync agent notifies the remote nodes of its local digests by sending periodic broadcasts, while receiving remote ones. Like CCNx Sync, iSync identifies whether the collections are up-to-date by comparing local with remote digests. When a remote collection digest does not match the local collection digest, a reconciling process starts, and the set difference is found by repeatedly requesting, receiving, and comparing remote IBF tables against local and global IBF tables. The notation of global IBF will be described in section 3.4.

It is important to note that, like other applications built on top of CCNx, the sync agent communicates with its peers using CCNx packets.

3.2 Protocol and Data Structure

Different types of data collections may have different update frequencies. For example, a chat application generates a few messages per second, while video streaming can produce hundreds of video chunks per second. The iSync protocol was designed to offer synchronization as a service to different applications simultanisely, and therefore is required to support different types of update frequencies. We designed the protocol data structure to address three key tasks: 1) Efficiently maintain a digest to reflect the status of the entire repository, and hence, the status of all the local collections; 2) Efficiently distinguish between up-to-date and out-of-date collections; 3) Obtain set differences quickly, and with minimal traffic overhead.

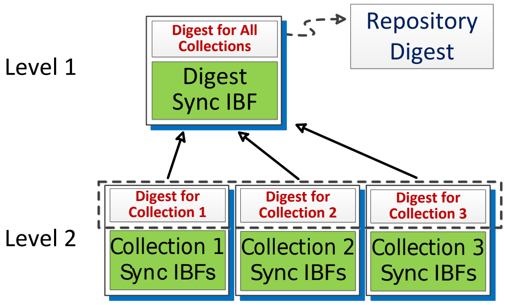

Figure 4: Hierarchical Synchronization Data Structure.

As shown in figure. 4, iSync uses a hierarchical two-level IBF: *Digest sync IBF* and *collection sync IBFs*. The higher level records the status of the entire repository, while the lower level logs file insertions or deletions of each sync collection separately. An update to a collection changes the collection's IBF in only one second level digest, by hashing the new name into the corresponding *collection sync IBF*. The change in a collection digest invokes an update to the repository IBF and digest in the first level.

Thus, the three key tasks are all achieved: 1) The first level digest holds the status of the entire repository. 2) Currency is maintaing by subtracting a remote first level IBF from the local first level IBF. While traditional NDN synchronization protocols iteratively send the digests of each collection, iSync requires only one data exchange to discover all the out-of-date collections. 3) In a similar way, the set difference of the changed collection is obtained by subtracting a corresponding remote IBF from the local second level *collection sync IBF*. iSync sequentially requests all the remote out-of-date *collection sync IBFs*, subtracts them from the local ones, obtains the updated namespaces, and fetches the content.

We illustrate the operation of the protocol using an example in which two applications declare collections to be synchronized. The first application is a dropbox style application that requests synchronizing all the files in a specific directory. To achieve this, the application declares a collection using the namespace "MyFiles/Pictures" in all the participating hosts. The second application is a media streaming application that requests synchronizing all the chunks of a video. This application declares a collection using the namespace "Movies/Frozen/". When a new file with the prefix "MyFiles/Pictures" is added to one of the hosts, iSync automatically indexes the file name in the IBF corresponding to the dropbox style applications. In a similar way, the name of the video chunk "Movies/Frozen/chunk12" is indexed to the corresponding media streaming IBF. A collection digest is calculated to represent each of the modified IBFs and the repository's IBF.

Fig. 5 shows Alice's hierarchical data structure after adding new contents, while fig. 6 presents the timeline of our discussed example.

In our example Alice and Bob are the participating hosts, and Alice adds ten new chunks of the movie Frozen to her local repository. All the chunks have the same shared prefix, Movies/Frozen/. In addition, Alice uploads her camera photos and names the uploaded files using the "MyFiles/Pictures" prefix. As will be explained in section 3.3, the names of the movie chunks are indexed and added to the 'Frozen' IBF, while the pictures are indexed to the 'Pictures' IBF. The digests of the two IBFs are changed according to the added names, while the digest of the repository IBF is changed according to the 'Frozen' and 'Pictures' digests. Upon the expiration of the sync timer, Alice sends an interest packet consisting of her repository digest. Bob doesn't recognize the digest in the incoming interest, and therefore he requests Alice's *Digest sync IBF*. When it is received, Bob subtracts the incoming *Digest sync IBF* from its local one, and discovers all the out-of-date collections. Bob then iteratively asks for the IBFs of the outdated collections, the IBFs of the 'Frozen' and 'Pictures' collections. As he did for the *Digest sync IBF*, Bob decodes and requests the added file names by subtracting the remote collections' IBF from the local ones. As will be described in section 3.4, when the subtraction fails to resolve all the differences, Bob requests Alice's previous (local) IBFs. After Bob receives and decodes the missing names, he indexes the changes in his *collection sync IBFs* and updates the collections and the repository digests. To store the content of a video chunk or a picture, Bob sends an interest with the chunk or picture name.

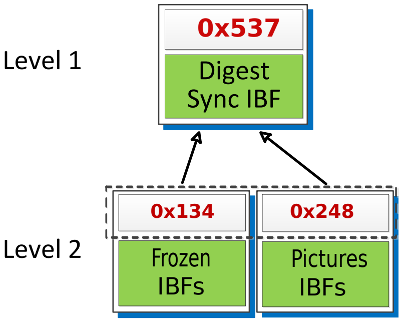

Figure 5: Alice's Hierarchical Data Structure.

We emphasize that while traditional synchronization protocols require multiple data retrievals and comparisons for a single update, iSync can reconcile multiple differences using a single data request. In the example, Alice uses a single interest packet to notify Bob that multiple changes were made in two collections. Also, a single request for a collection IBF results in the discovery of all the updates made in the collection. Therefore iSync's workload is concentrated in computation, which reduces time and traffic overheads.

It is important to note that while a single packet is used to notify a participant about multiple changes, and a single packet is used to reconcile multiple differences, the retrieval of the content names as well as the content items is done by sending one packet per name. Therefore, iSync can leverage the NDN characteristics and receive the requested content from a cached router or from a third user who already holds the missing content.

3.3 Name Encoding and Recording Scheme

The design of IBF handles fixed-length item names and does not support content lookup very well [4]. To sup-

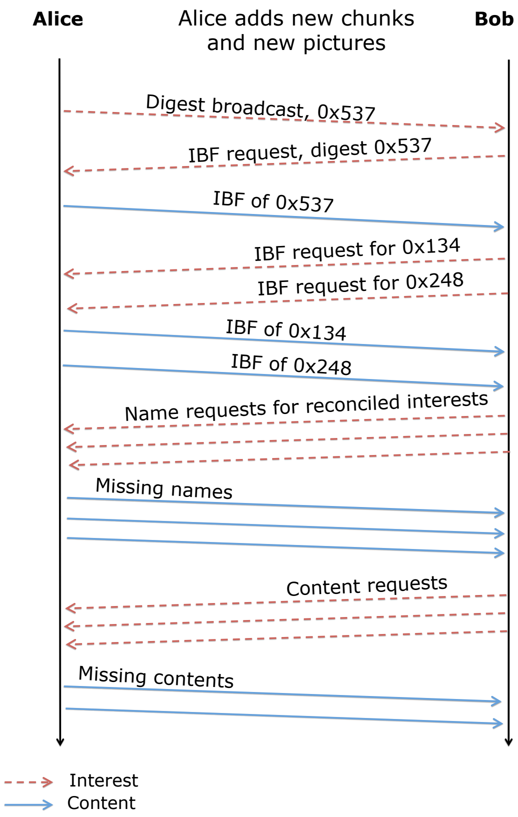

Figure 6: iSync Timeline Example.

port namespace synchronization in NDN, iSync introduces a name encoding and recording scheme.

The name encoding and recording scheme is part of the second level table described in section 3.2 and in Fig. 4. The scheme is responsible for two tasks: mapping variable-length file names into fixed-length IDs, and recording what items have been inserted. Mapping is carried out by using a hash-indexed table to support bidirectional mapping relations between file names and file IDs; Recording is based on an invertible Bloom filter to support file insertions, deletions, and queries. This IBF represents the collection digest. During the entire synchronization process, file names are replaced by fixed-length file IDs. Moreover, the scheme can be regarded as a name compression scheme that reduces traffic overhead. The insertion process of the name encoding and recording scheme is described in Algorithm 1.

Algorithm 1 Name Encoding and Recording Scheme

```
File_ID $\leftarrow$ Hash_Function(File_Name)
if File_ID is not found in Member Recording Table then
    Adding File_ID $\rightarrow$ Member Recording Table
    Insert File_Name $\rightarrow$ Name Encoding Table
else if File_Name Found In Name Encoding Table then
    Report File_Aleady_Inserted_Error
else
    Report Hash_Collision_Error
end if
```

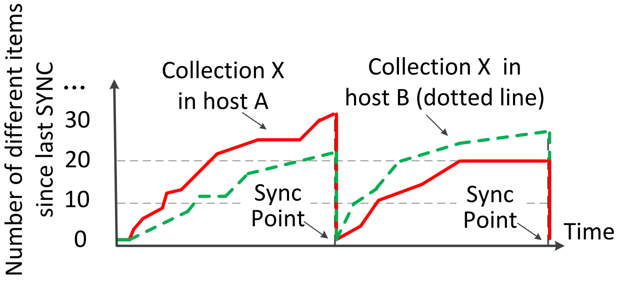

Figure 7: Periodical Synchronization.

To simplify the discussion, we refer to content name as a File name in Algorithm 1. However, the name could represent different types of data, as explained in the Introduction. Algorithm 1 distinguishes between two errors: Already existing content names and hash collisions. The first error implies that a content with the same name already exists in the collection. The second error indicates that the content name is new, but its hash value is already mapped by another content name. Both errors could be easily fixed by prompting a renaming request to the user. False positive errors such as the hash collision error could occur when mapping a long namespace into a shorter one. However, this probability is well studied and can be controlled by choosing the right algorithms and length of content IDs [17, 18]. In addition, by considering relevant application requirements and the potential number of names, collision resolutions can be maximized [19, 20]. The name encoding and recoding scheme is implemented in the second level of the iSync hierarchical design, and therefore each application could make its own decision regarding the length of the encoded IDs.

3.4 Difference Size Control Scheme For Collection Sync IBFs

Using the name encoding and recording scheme, the iSync protocol utilizes IBFs to hold the set of name IDs for each sync collection. As described in the background section, it is efficient to compute differences between two IBFs by subtracting them and decoding the resulting IBF. However, the decoding process can compute the differences only as long as there are pure cells in the resulting IBF. Therefore, there is no guarantee that all the differences can be decoded. For a fixed-size IBF, the more updates it holds, the less likely it can be perfectly decoded (recover all item IDs). Moreover, the number of updates varies among different sync collections, and does not follow a predicable pattern. Thus, a fixed size IBF can be inefficiency large in one application scenario, but small enough to cause decoding errors in another. To optimize the IBF size for various applications, we designed a difference size control mechanism.

First, as shown in Fig 7, hosts that have declared the same sync collections periodically confirm the consistency of their data sets. This periodic operation guarantees bounded delay of file shares and limits the potential size of differences between nodes.

Second, for any sync collection in one host, iSync creates multiple IBFs to hold the changes produced during a sync period. This scheme offers a flexible capability for recording

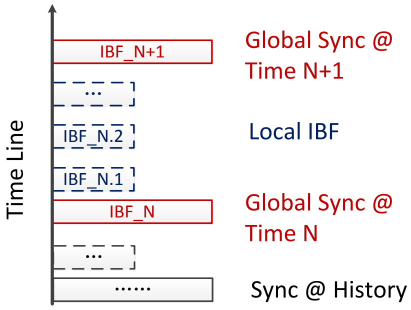

Figure 8: Local and Global IBFs for a Sync Collection.

and recovering the latest updates. As described in Fig 8, two types of IBFs are used: global and local. The global IBF can be regarded as a public version of the collection data. It is the latest local IBF in a sync cycle and the foundation of the first local IBF in the next synchronization cycle. The local IBFs support the process of reconciling the set difference at the end of a sync cycle. To find all changes made in a sync cycle, iSync first subtracts a remote global IBF from a local IBF. If it fails to obtain the complete set difference, it uses the local IBFs.

Algorithm 2 presents the details of the described difference size control scheme. When there is an update to a collection, and therefore to its current IBF, iSync checks whether the total number of updates exceeds the configured maximal number. If exceeds, the current IBF is marked as local, stored as a backup table, and the number of updates is reset to 0. In addition, to limit memory consumption, iSync checks if the number of stored IBF exceeds the maximal number allowed. When iSync timer expires the current IBF is marked as global. Each application configures its own constants for maximum number of updates and the number of history IBFs.

Algorithm 2 Difference Size Control Algorithm

```
if $updates\_count > max\_updates$ then
  if $stored\_IBFs\_count > max\_stored\_IBFs$ then
    $Dequeue\ Backup\_Queue$
    $stored\_IBFs\_count \leftarrow stored\_IBFs\_count + 1$
  end if
  $Enqueue\ Current\_IBF\_Table\ As\ Local$
  $updates\_count \leftarrow 0$
  $max\_stored\_IBFs \leftarrow max\_stored\_IBFs + 1$
end if
if $Time_{since\_last\_sync} > Time_{sync\_period}$ then
  $Globalize\ Current\_IBF\_Table$
  $Enqueue\ Current\_IBF\_Table\ As\ Global$
end if
```

The combination of the local and global IBFs makes differences between two IBFs traceable. To support the optimal decoding of different update frequencies, an application can tune the periodic sync time and the IBF size according to its specific requirements.

3.5 Recovery Scheme

False positive errors will occur if two different name IDs are mapped into the same IBF cells. Even though the possibility can be controlled, a recovery scheme is inevitably needed to provide disaster relief. In case a newcomer or returner jumps in with an empty or outdated sync collection, the original IBF design has a limited capability to recover.

To solve this problem, iSync uses a blacklist based scheme. After two hosts have gone through all their IBFs and still do not have a common view of the sync collections, the Bloom filters of the local and remote data sets (maintained by the name encoding and recording scheme) are exchanged. Each host reconciles local data sets and sends the list of files to remote nodes. Thus, local computing sources bear most of the overhead of the recovery scheme.

4. EVALUATION

This section evaluates iSync and compares its performance to the CCNx Sync. Section 4.1 introduces iSync implementation. Section 4.2 describes our experimental setup, including the ONL platform and the methods we used in the time and traffic overhead experiments. Section 4.3 presents the experimental results and analysis.

4.1 Implementation on CCNx

We built a prototype of iSync on top of CCNx and compared its performance with the CCNx Sync protocol. iSync was written in C to ensure its compatibility and performance. The prototype consists of a repository server, a sync agent and a file retriever. All components are connected by system FIFOs to ensure the protocol's potential for scaling.

The sync agent consists of the three schemes described in section 3: name encoding and decoding, difference size control, and recovery mode. To reduce false positive errors, we used a hash indexed table with 2^{24} entries to build bidirectional name-to-ID mapping, and a counting Bloom filter of 2^{24} cells. Each cell in the counting Bloom filter contained an 8-bit counter to count the number of the inserted files. As shown in Fig. 8, we used an IBF FIFO to store a history of 32 IBFs for each sync collection. Each IBF consisting of 160 cells, and guaranteed to hold 128 IDs by default. The capacity of each IBF could be configured when declaring sync collections. All communications followed the pattern of the NDN architecture, which means that all data chunks (contents) were retrieved by exchanging interests and data packets.

4.2 Experiment Setup

We used the open network laboratory (ONL) [21] to measure the synchronization time and traffic overheads of CCNx Sync and iSync. Our experiments explored the impact of four factors: 1) file name length, 2) file size, 3) number of hosts, and 4) topology type. Each host in the ONL system ran Ubuntu 12.04.2 LTS and CCNx version 0.7.2 on a Xeon CPU @ 2.5GHz with 4 GB memory.

In each experiment, we inserted a file into a local host, and waited until this file was replicated by iSync or CCNx Sync, and hence, synchronized, in all the participating hosts. We used tcpdump to capture the exchanged traffic during the experiments, and analyzed the traces to measure the traffic overhead. To measure the time overhead, we modified a small section in the CCNx code to record the content insertion timestamp. Then we calculated the differences between the insertion timestamp in the local host and the insertion timestamp in the remote hosts. We reported the average synchronization time as the time overhead.

As described in [5], the ratio between the number of inserted items and the number of IBF cells impacts the capability to reconcile the differences of two IBFs. To ensure a successful decoding rate of more than 99%, we limited this ratio to 60%. Therefore, a new backup of current IBF was created when the number of new updates exceeded 60% of its holding capacity.

4.3 Experimental Results

In this section we first describe the results of a simple topology consisting of two nodes, and show the impact of the name length and file size factors on sync time and traffic overhead. Then we show how the topology type and number of hosts affect the sync time. Last, we present the measured performance of iSync's recovery scheme.

While measuring the performance of the CCNx Sync protocol, we discovered that the API used to insert a content into the CCNx repo is not optimized, and took about 2.8 seconds. Due to this large overhead, we reported two different times: CCNx Sync and CCNx Sync Data. The former records the time overhead of the entire synchronization process, while the latter does not include the API overhead.

4.3.1 *Time Overhead of iSync in a Simple Topology*

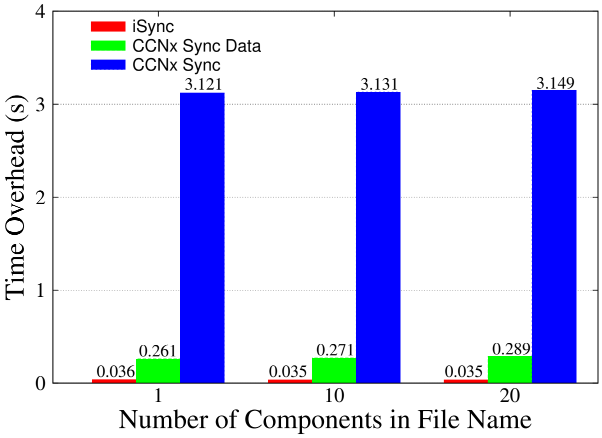

Figure 9: Impact of Number of Components in File Name on Synchronization Time.

Fig. 9 shows the synchronization times of a 128 KB file with different name length using the CCNx Sync and iSync protocols. The file *song1* consist of one component, while the file *John'sDocuments/Music/Maroon5/song1* consists of four components. This experiment was performed on a two node network to provide a basic measurement.

The figure shows that synchronizing one file with one name component takes about 3.1 and 0.036 seconds by CCNx Sync and iSync respectively. This time overhead includes file insertion time, reconciling time, and file retrieval time. CCNx Sync Data time is approximately 0.261 seconds. Therefore, iSync as an end-to-end system is about 86 times faster than CCNx Sync, while it is 8 times faster than the CCNx Sync Data when ignores the CCNx insertion overhead. We found that the number of components in a file name does not impact the sync time results.

File insertion and resolution play critical roles in iSync. We evaluated the time overhead of the two operations under different intensities to further understand their effect on the synchronization times. As shown in Fig. 10 it takes about 3 ms and 330 ms to insert 80 and 20480 files respectively. In these experiments, we increased the IBF table sizes to ensure all file names could be perfectly resolved. Therefore, the time overhead was affected by the IBF table size, number of inserted files, and computing power of the host.

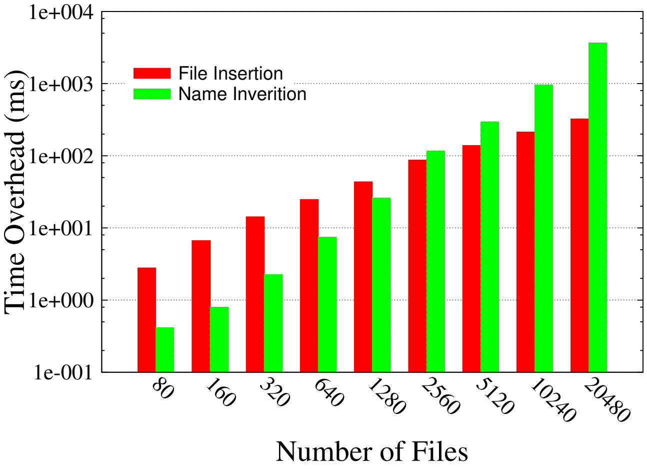

Figure 10: File Insertion and Recovery Time for iSync Protocol.

Fig. 11 shows the effect of the file size on the synchronization time consumptions. The left figure shows the measured results, while the right figure shows the ratio between CCNx Sync and iSync. As expected, for both protocols, the synchronization time increases as file size increases. Synchronizing a file of 65536 KB took about 9 seconds for iSync and 63 seconds for CCNx Sync. Moreover, iSync was about 86 times faster than CCNx Sync for small files, and about 8 times faster for large files. The figure also shows that iSync is still 8 to 10 times faster than CCNx Sync, even if we do not consider file insertion time.

Synchronization performance can be divided into two tasks: reconciling content names and retrieving the file contents. As file size grows, the weight of data retrieval grows. iSync uses a more efficient scheme to find the set difference and optimized size of packets to deliver data, and therefore is faster then even CCNx Sync data, for large files.

4.3.2 *Traffic Overhead*

Traffic overhead impacts the scalability and efficiency of a synchronization protocol. We synchronized files of different sizes (from 128 KB to 64 MB) and measured the traffic overhead by capturing the network traffic, using tcpdump traces.

Fig. 12 shows the number of packets transmitted by CCNx Sync and iSync for different files sizes, and the ratio between the protocols' overheads. In synchronizing a small file of 128 KB, iSync and CCNx Sync transferred 10 and 182 packets respectively. The numbers increase to 1032 and 49589 when the file size increases to 64 MB. From the ratio point of view, iSync is about 18 and 48 times more efficient than CCNx Sync on number of packets while sharing files of 128 KB and 64 MB respectively. We explain this ratio by looking at the number of packets sent by CCNx per

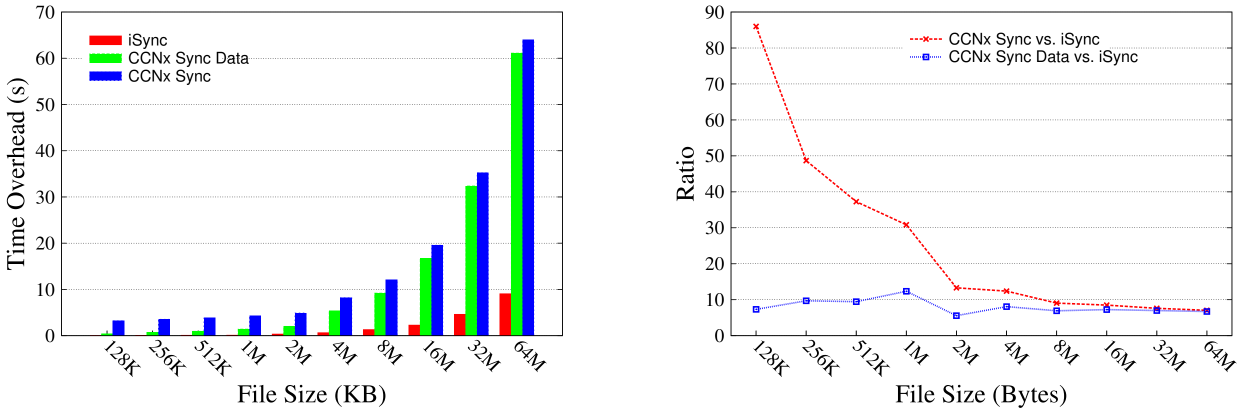

Figure 11: Impact of File Size on Time Cost (left) and Ratios of CCNx Sync vs. iSync (right).

one update as shown in the example demonstrated in the background section.

Fig. 13 shows the number of bytes exchanged during the synchronization process. For synchronizing a file of 128 KB, iSync and CCNx Sync exchange about 160 KB and 204 KB. For a file of 64 MB, the traffic overheads increases to 65.5 MB and 83.6 MB. In all tests, iSync exchanges a smaller number of bytes than CCNx Sync. However, the advantage of iSync over CCNx Sync in traffic amount is not as significant as in the exchanged number of packets. We believe that this might be the result of smaller packets sent by CCNx Sync compared to iSync.

The main reasons for the advantages of iSync over CCNx Sync can be summarized as follows. First, while CCNx Sync concentrates more on exchanging node information, iSync consumes more local computing resources for the IBFs' decoding process. It also causes the inconsistent differences between the number of exchanged packets and bytes in the experiments (most packets transmitted by CCNx Sync are smaller than 80 bytes). Second, the max data unit limitations of the protocols are different. Both CCNx Sync and iSync use jumbo packets to deliver the files. However, the max size of CCNx Sync content packets is about 22KB (which might be disassembled according to the MTU limit of switches and routers). The smaller the size limit, the more packets that need to be sent. To improve the performance, iSync uses content packets of 64KB, which is the optimal value the current CCNx daemon can handle. It significantly reduces the number of packets, as is shown in Fig. 12. Third, extra interest and content packets are exchanged by the CCNx repository, and therefore we can expect a larger traffic overhead.

4.3.3 Scalability Test

In this section, we evaluated and compared the performance of the iSync and CCNx Sync protocols in multiple network topologies with number of nodes ranging from 2 to 32. As described in the background section, interest packets are satisfied by a cached content or a producer. To verify the correctness of the protocol in the NDN architecture, we tested it in chain, ring, star and full mesh topologies, representing weak to strong connectivity. Files were inserted into the start node of the chain topology, and the core node (which has direct connections to all other nodes) of the star network.

Fig. 14 shows the scalability results. We found that the synchronization times of CCNx Sync were very unstable; therefore, we plotted the average results in the bar charts and added annotations for worst results on the top of each bar. To provide a better display of the protocols' performance, we also plotted the results of CCNx Sync data to ignore the API overhead.

As expected, the results show that the chain topology produces the slowest synchronization times in both protocols.

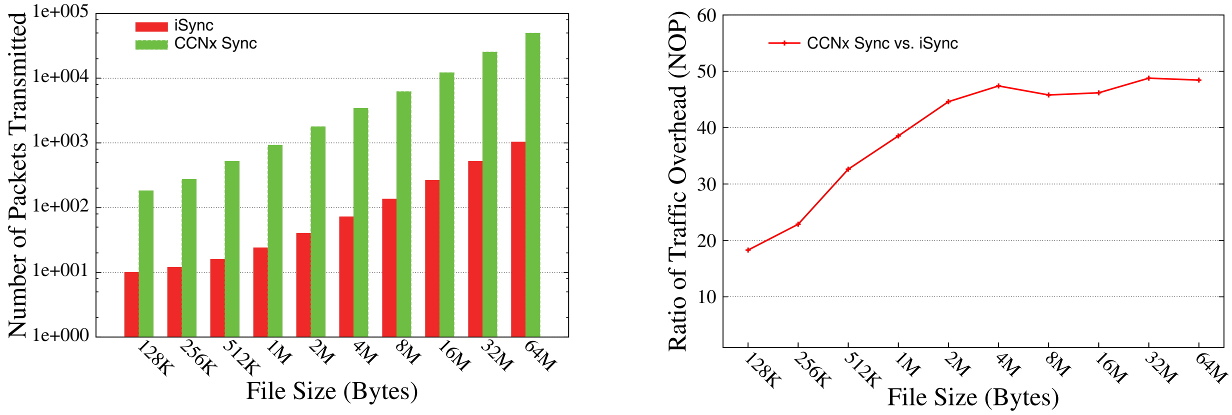

Figure 12: Traffic Overhead for Various File Sizes.

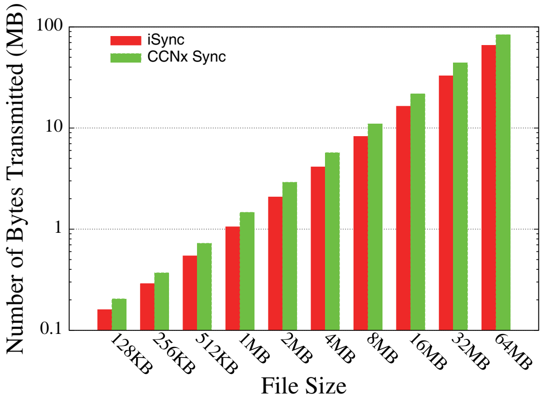

Figure 13: Number of bytes vs. File Size.

Since we inserted the files into the start of the chain, each file should be delivered one by one until all nodes have received it. Moreover, the time overhead shows an approximately linear relationship with the number of nodes. The variation in the results of the chain topology is large, especially for CCNx Sync. The average and max times for iSync to synchronize a 128 KB file in a 32 nodes network are 0.37 and 0.44 seconds respectively, and CCNx Sync results are 12 and 30 seconds. Therefore, the max time is about 2.5 times larger than the average time. Ratio similar to that is found in the results for iSync and CCNx Sync Data (about 2.9 times).

The results show that synchronizing a file in ring topology is much faster than in a chain topology, because the parallel synchronization starts from two directions. For example, the synchronization time in a ring network of 32 nodes is nearly half of that in a chain network. The variation is also much smaller.

The star topology achieves the fastest synchronization time. In our experiment, we inserted the file into the central node that has direct access to all the other nodes; therefore, the data could be delivered to all nodes at the same time. During the experiments, we found that large variations occasionally occurred in the results of CCNx Sync in a 32 node network. After carful consideration, we believe that this variation is caused by the forwarding plane of CCNx and the design of CCNx sync. As illustrated in the example in the background section, CCNx Sync is a pairwise protocol that operates between neighbors. Therefore, when adding a file into the central node in a star topology, the protocol operates on each two neighbors separately. Moreover, the behavior of the CCNx forwarding strategy delays the RootAdvise notification sent to a subset of the neighbors.

Compared to other types of topologies, full mesh topology networks have the strongest connectivity. However, the time overhead of data synchronization in full mesh topology is no better than that in star topology. All nodes publish digests after receiving any updates, and each of them has to process redundant data units from synchronization protocols (iSync or CCNx Sync).

4.3.4 Recovery Overhead

The recovery overhead of iSync from false positive errors is mainly concentrated in local computing. Specifically, the effectiveness of this scheme is highly dependent on the hash algorithm and local computing power. To evaluate the ef-

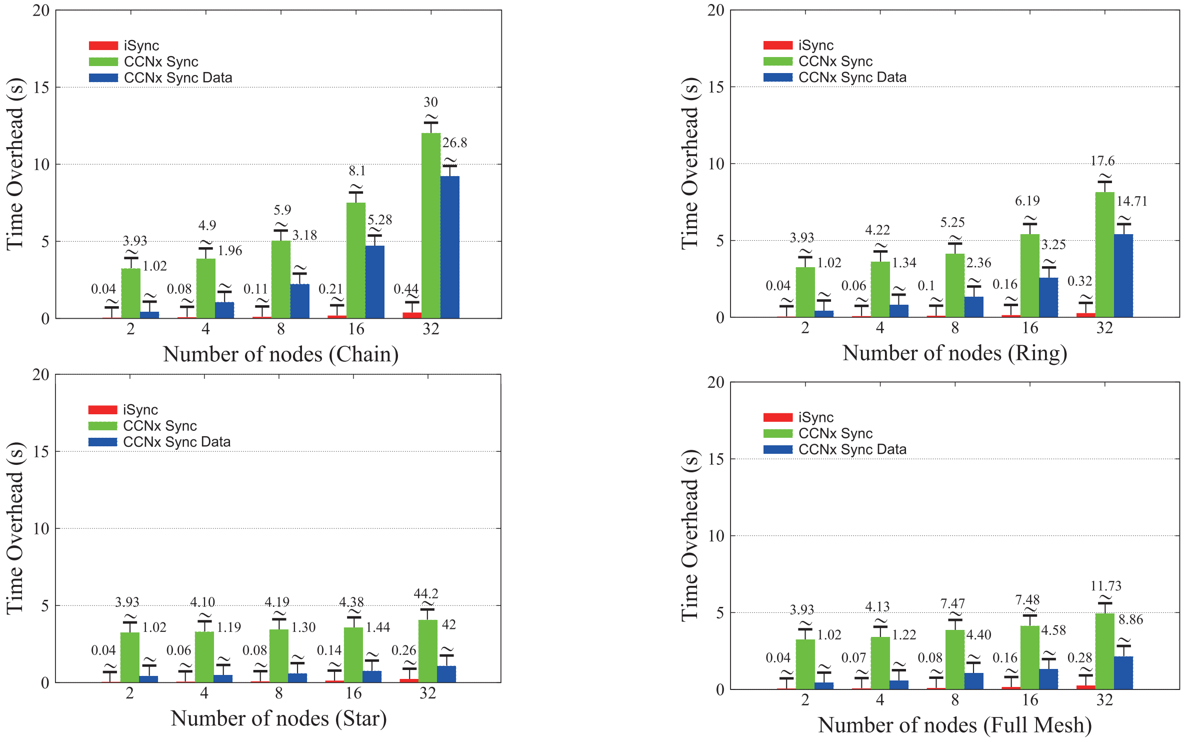

Figure 14: Average Synchronization Time of iSync and CCNx Sync in Networks of Various Topology Types (with max recorded results on top of each bar).

fectiveness and efficiency of the recovery scheme, we randomly deleted a few file contents in the defined collections and measured the time it took the the protocol to reconcile the differences. We manually deleted up to 10% of the collection files, and repeated each test ten times.

As shown in Fig 15, the time overhead of the recovery scheme shows a linear relationship with the size of the collection. It costs about 10 ms for the iSync host to recover missed file names from a data collection of 1K files. The time overhead grows to about 3.5 seconds for recovering all names from a collection of 1024K files. In those experiments, and as guaranteed in theory, iSync could always recover and reconcile discarded items.

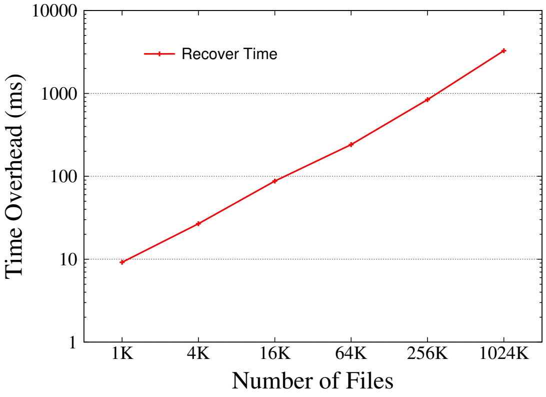

Figure 15: Recover Time vs. Number of Items That Have Already Stored.

5. CONCLUSIONS AND FUTURE WORK

This paper presents iSync, a high performance synchronization protocol, which utilizes the invertible Bloom filter (IBF) to reconcile sync collections between nodes. The protocol was designed to provide synchronization as a service to CCNx applications. Due to the use of hierarchical IBFs structure, each application can configure its own IBF size, and therefore, iSync can simultaneously support different applications with various update frequencies. The protocol stores a history of IBFs, and therefore does not require prior context in most application scenarios, and can reconcile a number of differences in a single comparison.

We evaluated the performance of iSync and compared it with the CCNx synchronization protocol across a range of network topologies and sizes. While CCNx Sync is the only synchronization service running on the CCNx stack, its study and performance indicate that it is not yet optimized. Our experiments showed that iSync was about 8 times faster in most of the test cases, and that it reduced the synchronization traffic (i.e., packets sent) by about 90%.

In the future, iSync could be integrated into a comprehensive NDN platform to allow additional utilization and broader examinations. Further evaluation is required to measure the performance of iSync in lossy networks, and to compare the protocol?s performance with the performance of the popular ChronoSync protocol. More schemes should be developed to improve iSync?s performance, such as the decoding time and the overhead caused by sending the entire IBF. There is a strong relationship between IBF capacity and decoding inaccuracy. The current version of iSync allows the application to determine the exact IBF size and to limit the size of the set-difference before creating a new local IBF. To simplify the deployment of iSync and to improve the traffic and time overheads, the relationship between IBF size and an application?s characteristics and update rate should be further studied.

Acknowledgement

This work was supported by NSF grants CNS-1040643 and CNS-1345282. The authors would also like to thank John DeHart for his support on the Open Network Laboratory.

6. REFERENCES

[1] Lixia Zhang, Deborah Estrin, and et al. Named data networking project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.

[2] Giulio Grassi, Davide Pesavento, Lucas Wang, Giovanni Pau, Rama Vuyyuru, Ryuji Wakikawa, and Lixia Zhang. Vehicular inter-networking via named data. *arXiv preprint arXiv:1310.5980*, 2013.

[3] The dropbox website. https://www.dropbox.com/.

[4] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, pages 792–799. IEEE, 2011.

[5] David Eppstein, Michael T Goodrich, and et al. What's the difference? efficient set reconciliation without prior context. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 218–229. ACM, 2011.

[6] Content centric networking (CCNx) project website. http://www.ccnx.org.

[7] Andrew Tridgell and Paul Mackerras. The rsync algorithm,"australian national university. Technical report, TR-CS-96-05, 1996.

[8] Van Jacobson, Rebecca L Braynard, and et al. Custodian-based information sharing. *Communications Magazine, IEEE*, 50(7):38–43, 2012.

[9] Zhenkai Zhu and Alexander Afanasyev. Lets chronosync: Decentralized dataset state synchronization in named data networking. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*.

[10] Zhenkai Zhu and Chaoyi et al. Bian. Chronos: Serverless multi-user chat over ndn. Technical report, Technical Report TR008, UCLA Named Data Networking Project, 2012.

[11] Ralph C Merkle. A certified digital signature. In *Advances in Cryptology Proceedings*, pages 218–238. Springer, 1990.

[12] Sachin Agarwal, Starobinski, and et al. On the scalability of data synchronization protocols for pdas and mobile devices. *Network, IEEE*, 16(4):22–28, 2002.

[13] Cedric Westphal. Synchronizing state with strong similarity between local and remote systems. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, pages 15–20, 2012.

[14] Bittorrent sync website. http://www.bittorrent.com/sync.

[15] The Google Drive website. http://www.google.com/drive/about.html.

[16] NSF future Internet architecture project website. http://www.nets-fia.net/.

[17] Ralph C Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3(1):43–58, 1990.

[18] Jean-Sébastien Coron, Yevgeniy Dodis, and et al. Merkle-damgård revisited: How to construct a hash function. In *Advances in Cryptology–CRYPTO 2005*, pages 430–448. Springer, 2005.

[19] Gary D Knott. Expandable open addressing hash table storage and retrieval. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 187–206. ACM, 1971.

[20] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB*, volume 80, pages 1–3, 1980.

[21] Charlie Wiseman, Jonathan Turner, and et al. A remotely accessible network processor-based router for network experimentation. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 20–29. ACM, 2008.