Scalable Name-based Data Synchronization for Named Data Networking

Minsheng Zhang, Vince Lehman, Lan Wang
{mzhang4, vslehman, lanwang}@memphis.edu
University of Memphis

***Abstract*—In Named Data Networking (NDN), data synchronization plays an important role similar to transport protocols in IP. Many distributed applications, including pub-sub applications such as news and weather services, require a synchronization protocol where each consumer can subscribe to a different subset of a producer's data streams. However, existing Sync protocols support only full-data synchronization, which is a special case of this problem. We propose PSync to efficiently address different types of data synchronization. Names are used in PSync messages to carry producers' latest namespace information and each consumer's subscription information, which allows producers to maintain a single state for all consumers and enables consumers to synchronize with any producer that replicates the same data. We have implemented PSync in the NDN codebase and used it to develop a prototype pub-sub module for building management. Our experimental results show that PSync scales well as the number of consumers, subscriptions, and data streams increases and it outperforms the state-of-the-art Sync protocol in supporting full-data synchronization.**

I. Introduction

Data synchronization is a basic requirement for a large number of distributed applications such as calendars, Dropbox, and email. Some applications, e.g., multi-user chat, require every participant in a group to receive all the new data produced by everyone else. We call this scenario *full-data synchronization*. Alternatively, each user may be interested only in a **subset** of the produced data, which is very common in pub-sub applications, such as news and weather subscription services, where producers publish a diverse set of data and consumers each subscribe to a subset. For example, a news subscriber may read only about technology and travel, but not entertainment or other news categories. This is what we call *partial-data synchronization*, which is a *generalization* of the full-data synchronization problem, as a user's interest (or subscription) can range from an empty set to a full set of the data.

In this work, we investigate the partial-data and full-data synchronization problems in the context of the *Named Data Networking (NDN)* architecture [1]. Motivated by today's increasingly data-centric applications, the NDN design addresses the Internet's lack of support for scalable data distribution, mobility, and security. It makes *immutable data with hierarchical names and producer signatures* a common abstraction for both the network layer and application layer. Producers publish data under unique names, consumers use data names to request data, and the network uses names to forward consumers' requests and cache returned data for future requests.

This work was supported by NSF Grant 1344495 and a Cisco grant. Minsheng Zhang is currently with Google and Vince Lehman is with ReTrans.

Since NDN uses names, not end point identifiers, to fetch data, it does not use TCP-like channel-based protocols for reliable delivery. Instead, name-based data synchronization (Sync) has become the equivalent of TCP to bridge the network layer and application layer in NDN. While existing Sync protocols in NDN, e.g., ChronoSync [2] and iSync [3], address full-data synchronization, they do not handle partial-data synchronization. Therefore, the current Sync protocols in the NDN platform cannot support pub-sub applications efficiently. One alternative is to start with an Internet architecture that is based on the pub-sub model (e.g., PSIRP/PURSUIT [4]). Our ultimate goal, however, is to design an architecture that supports *a wide range of applications including pub-sub applications* based on a **generic abstraction** as provided by NDN. Therefore, this work focuses on designing the right Sync protocol for NDN to support pub-sub applications. Note that an in-depth discussion on the architectural design choice is out of scope for this paper.

To illustrate the requirements for partial-data synchronization, consider a smart-phone app store. Due to their mobile device's limited storage capacity, users may only install a small portion of the apps available to them. In such a scenario, whenever any installed application is updated, the mobile device should receive notifications of the update in order to fetch it. Since there can be a large number of apps and millions of users for each popular app, a *scalable* design for updating the apps needs to satisfy the following requirements: (a) the app store should not have to keep track of the *users of each app* in order to send proper notifications, (b) the phones should not have to check the app store periodically for *every installed app* to get updates, and (c) the phones should be able to synchronize with *any app store* that has the same set of apps. These requirements are applicable to many pub-sub applications with partial-data synchronization semantics.

Based on the above requirements, we propose a protocol called ***PSync*** to efficiently synchronize a subset of data name prefixes, i.e., a sub namespace, in NDN. PSync uses Invertible Bloom Filters (IBF) [5] to represent the latest data names in a namespace and utilizes the subtraction operation of IBF to efficiently discover the list of new data names that have been produced in the period between an old IBF and new IBF. Using the list of new data names and consumers' subscription information, a producer can notify a consumer if new data matching the consumer's subscription has been produced.

Our design satisfies the aforementioned requirements as follows: (a) *scalability under large number of consumers*: a PSync Interest message from each consumer carries the consumer's subscription information and previously received producer state, so that the producer has all the information needed to process the message without having to keep track of every consumer. Moreover, the producer maintains a single IBF of its state for all consumers rather than one IBF per consumer; (b) *robustness under producer failures*: because producers do not maintain state about consumers, each consumer can synchronize with any producer among a group of producers that replicate the same data; and (c) *scalability under large number of subscriptions*: efficient data representations such as Bloom Filters (BF) [6] and ranges are used to encode consumers' subscriptions so only one PSync Interest message is sent from each consumer, irrespective of the number of subscribed name prefixes.

Our evaluation using a large ISP's point-of-presence topology shows PSync performs well with different numbers of consumers, subscriptions, and data streams. Furthermore, it has lower delay than ChronoSync in supporting full-data synchronization. Finally, we implemented a prototype pub-sub module for building management using PSync and evaluated the scalability of PSync in this application.

The paper is organized as follows. Section II reviews the NDN architecture, Bloom Filter, and associated extensions. Sections III, IV and V present the design, implementation, and evaluation of the proposed PSync protocol. We discuss related work in Section VI. Section VII concludes the paper.

II. Background

A. NDN Architecture

NDN [1] is a new data-centric Internet architecture that supports efficient and secure data distribution. NDN uses two types of packets, *Interest* and *Data* packets, to request and return named content, respectively. NDN also binds a data packet's name and content using the producer's key, so any receiver can verify the authenticity of the data packet.

Consumers send interests containing the name (or name prefix) of the desired data. Names are hierarchical with variable number of components, e.g., /youtube/video105/frame1. The name in an interest can be the same as or a prefix of the name in a matching data packet. For example, the interest /youtube/video105 can fetch any data with this prefix, e.g., /youtube/video105/frame1/segment1, but only one data packet is returned for each interest. When an interest arrives at a router, if there is no cached matching data and the router has not forwarded such an interest before, the router decides how to forward the interest by looking up the interest name in its forwarding information base (FIB). The router also maintains a pending interest table (PIT) that records which interface the interest arrived on and to which interface it was forwarded. When the interest reaches the producer or a node that has cached the matching data, the data will be sent back to the consumer on a symmetric return path using the information recorded in the PIT. Data packets are cached in the Content Store (CS) on each node on the return path which can be used to satisfy future interests that request the same data.

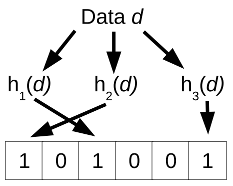


Fig. 1: Data d is passed to each hash function which maps the data to a bit position in the bloom filter.

B. Bloom Filter and Extensions

A Bloom Filter [6] is an efficient data structure that uses a bit array to succinctly represent a dataset. It allows for queries of whether an element is in the dataset or not by using multiple hash functions to map each element in the set to certain bits. To insert an element into the Bloom Filter, the element is passed to each hash function to get a list of bit positions. These bit positions are set to 1 to indicate that the element is a member of the set (Figure 1). To perform a membership query for an element, the element is passed to each hash function to get a list of bit positions. If each bit position is set to 1, the Bloom Filter shows the element as a member of the set. Due to the compactness of the data structure, answers to membership queries may be false positives (not false negatives), as elements may be hashed to some of the same bits. The false positive rate (p) of a Bloom Filter is approximately $(1 - e^{-kn/m})^k$, where m is the size of the bit array, k is the number of hashes, and n is the number of elements inserted into the Bloom Filter [6]. This means that a larger bit array and a smaller element set lead to lower false positive rate. Moreover, given n and m, $k = (m/n) * ln2$ produces the lowest false positive rate [6].

The basic Bloom Filter does not support a removal operation, since multiple elements may set the same bit to 1. If two elements set the same bit to 1 and one of the elements is removed, setting the bit to 0 would make it appear as if the other non-removed element is also not in the Bloom Filter. In order to support element deletion, a count array is added to the regular Bloom Filter to record the number of times that a bit is set by an insertion – such a data structure is called a Counting Bloom Filter (CBF) [7]. Every time an element is inserted, the bits in the bit array are set to 1 and the corresponding bits in the count array are incremented by 1. When an element is removed, the corresponding bits in the count array will decrease by 1. When a bit's count is equal to 0, the corresponding bit in the bit array can safely be set to 0. The removal process for a CBF is shown in Figure 2.

Although a BF or CBF supports membership testing, one cannot invert either one of them to determine the specific elements that set the bits. ***Invertible Bloom Filters (IBF)*** [5] are a new data structure designed to solve this problem. Instead of maintaining a simple bit array to represent set membership, IBFs use the hash functions to map the elements to cells that maintain an idSum, hashSum, and count to track the sum of

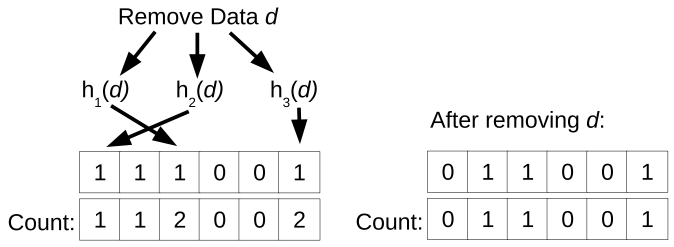


Fig. 2: Data d is removed from the Counting Bloom Filter. Although d maps to three bit positions, only one of the bit positions is set to 0 due to another element being mapped to two of the same positions.

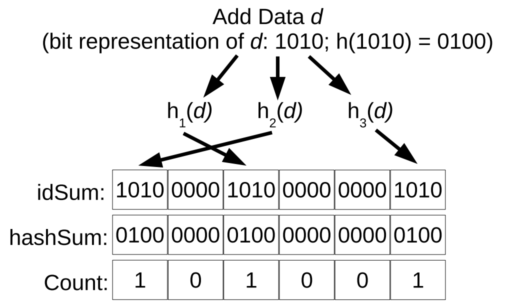

Add Data *d*
(bit representation of *d*: 1010; h(1010) = 0100)
h1(d) h2(d) h3(d)
idSum: 1010 0000 1010 0000 0000 1010
hashSum: 0100 0000 0100 0000 0000 0100
Count: 1 0 1 0 0 1

Fig. 3: Data d is added to the Invertible Bloom Filter.

inserted elements' KeyID, sum of their hashes, and number of inserted elements, respectively.

When an element is inserted, the hash functions are applied to the element to get a list of cells to which the element corresponds. For each corresponding cell, the cell's idSum is XOR'ed with the inserted element's KeyID, and the cell's hashSum is XOR'ed with the inserted element's hash value; the cell's count is also incremented. The insertion process is shown in Figure 3. When an element is deleted, the operations are similar to those for element insertion except that the count is decremented in each cell the element corresponds to.

A list of elements can be retrieved 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 the cell's hashSum. This element can be added to the retrieval list and deleted from all its corresponding cells. This deletion may remove a collision from existing cells thus producing new pure cells. This process continues until no more pure cells can be found in the IBF. To use an IBF effectively, *if there are* d *elements in the IBF,* $1.5 * d$ *cells are required to decode the IBF with a low decoding failure probability* [5].

IBFs also support a *set difference operation* through subtraction – for each cell in two IBFs, the corresponding count bits are subtracted, and the corresponding idSums and hashSums are XOR'ed. Suppose we calculate $IBF_1 - IBF_2$, a cell with a count of 1 (-1) means that it contains an element only in IBF_1 (IBF_2). Along with the idSum and HashSum from the subtraction, we can determine the specific different elements in the two IBFs and to which IBF each different element belongs. Removing the different elements may reveal more pure cells, so this process can be repeated to extract more differences.

Only fixed-length numbers (KeyID) can be inserted into an IBF. *If we want to insert NDN names, we first need to hash each name into a fixed-length number (KeyID) and then keep a mapping table to associate each KeyID with the original name.* After that we can do the normal IBF operations by using the KeyIDs. Retrieving an element requires first getting the KeyID from the IBF and then looking up the corresponding name in the mapping table using the KeyID.

Data Stream 1: /<prefix1>/1, /<prefix1>/2, ...
Data Stream 2: /<prefix2>/1, /<prefix2>/2, ...
Data Stream 3: /<prefix3>/1, /<prefix3>/2, ...
...

Fig. 4: Data Streams

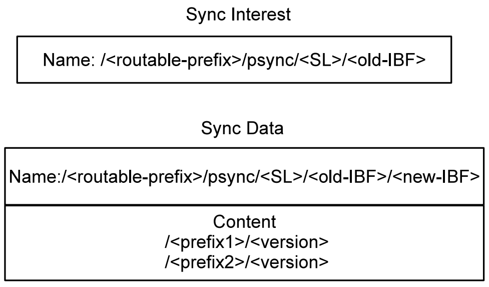

Sync Interest
Name: /<routable-prefix>/psync/<SL>/<old-IBF>
Sync Data
Name:/<routable-prefix>/psync/<SL>/<old-IBF>/<new-IBF>
Content
/<prefix1>/<version>
/<prefix2>/<version>

Fig. 5: Sync Messages (*SL* represents the consumer's subscription list; *old-IBF* and *new-IBF* represent the producer's old state and current state in IBF.)

III. Design

In this section, we first give an overview of the PSync design. Next, we introduce how state information is encoded and then present the protocol details.

A. Overview

We first define a *data stream* to be a set of data which have the same name prefix but different sequence numbers (Figure 4). We assume that each producer generates a set of data streams, and each consumer is interested in a subset of the data streams. For example, a building management system may have electricity data produced by many devices each forming a data stream with the name prefix /⟨building⟩/Electricity/⟨panel⟩/⟨device⟩ (within the data stream, each data point's name starts with the name prefix and ends with a sequence number). Suppose a consumer is interested in a data stream, it can subscribe to the corresponding name prefix through PSync. Then it will be informed whenever new data points are generated in the subscribed data stream. It can also subscribe to multiple devices' data streams using their name prefixes.

PSync uses regular NDN interest and data messages for subscription and notification. We call these messages *Sync Interest* and *Sync Reply*, whose names begin with a routable prefix to reach the producer(s) and a component for demultiplexing, e.g., /⟨routable-prefix⟩/psync (see Figure 5). Each consumer sends Sync Interests to the producer in order to learn about newly produced data in their subscribed data streams (Step 1 in Figure 6). The Sync Interest contains the consumer's subscription list in its interest name (not payload) which is used by the producer to check for updates to the subscribed

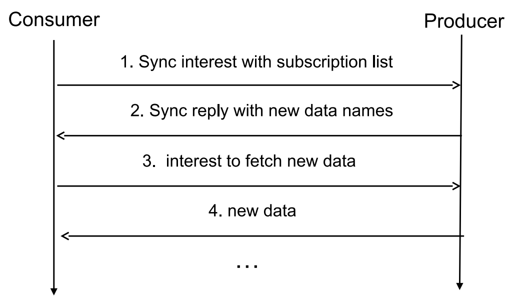

Consumer
Producer
1. Sync interest with subscription list
2. Sync reply with new data names
3. interest to fetch new data
4. new data
...

Fig. 6: Overview of PSync Protocol Exchanges

data. If any data stream in the consumer's subscription list has new data items, the producer will generate a Sync Reply to the consumer containing *a list of new data names in the subscribed data streams* (Step 2 in Figure 6). This is much more efficient than blindly sending all the new data names to all the consumers, since the producer may have many data streams and consumers and each consumer may be interested in only a small set of data streams.

Upon receiving the Sync Reply, which contains a list of new data names, the consumer will further check whether the data names indeed belong to its own subscription list. If a data name is a false positive, which means the producer returned a data name to which the consumer has not subscribed, then the name is ignored by the consumer. Otherwise, the consumer sends an Interest to the producer to fetch the new data and the producer or an intermediate cache will return the data (Step 3 and 4 in Figure 6). If no new data matching the consumer's subscription list has been produced when the Sync Interest is received, the Interest will be maintained by the producer separately from the PIT in the NDN forwarder. The producer will respond immediately if any subscribed data stream has new data before the Interest expires. When the interest expires, the consumer will send a new Sync Interest. Note that the above operations are actually handled by PSync in the consumer and producer applications through library calls.

B. Data Representation

PSync uses a number of representations including *Bloom Filters (BF)* and *ranges* for consumers to express their *Subscription List* in their Sync Interests. Moreover, it uses *Invertible Bloom Filters (IBF)* to represent producers' latest datasets, i.e., *Producer State*. Bloom Filters and their extensions (Section II) are space efficient data structures that enable consumers and producers to exchange their information in a compact form, identify new data names efficiently, and match those names with consumers' subscriptions quickly.

1) Subscription List: Suppose a producer has n data streams with name prefixes $P = \{p_1, p_2, ...p_n\}$, and a consumer is interested in a subset of the data streams $Q = \{q_1, q_2, ..., q_j\} \subseteq P$. The set Q can be hashed into a Bloom Filter f.[1] When $|Q|$ is large, we use ***Compressed Bloom Filter*** [8] to make the encoded subscription list smaller at the cost of a slightly higher false positive rate (Section V). Alternatively, if Q includes all the prefixes from p_i to p_j in P (ordered alphabetically), then this set can be simply represented as a range $[p_i, p_j]$. There are also other special cases, e.g., $|Q| = 1$ or $P = Q$, which can be encoded using simpler representations than Bloom Filters. When the consumer sends a Sync Interest, it selects the most compact format for its subscription list and sends the format information along with the encoded subscription list in the interest name to the producer so that the producer can decode correctly.

[1]While regular BFs are sent in Sync interests, consumers can use Counting Bloom Filters (CBF) *locally* to support deletion of subscriptions more efficiently (regular BFs support only insertions).

2) Producer State: PSync adopts ChronoSync's approach of letting each producer name data sequentially [2]. The latest dataset can be represented by an IBF which contains only one data name from each data stream, i.e., the data stream's name prefix plus its latest sequence number. When a data stream with the name prefix p generates a new data item and increases its sequence number from i to $i + 1$, PSync will remove the name p/i from the IBF and add $p/(i+1)$ to the IBF. In doing so, the IBF encodes only N items, where N is the number of data streams. Note, however, that because PSync decodes the differences between two IBFs, *the IBF size should be proportional to the expected number of **updated** data streams in a Sync period, which can be much smaller than N.*

The producer sends its IBF information to every consumer through its Sync Reply. Every time a consumer sends a Sync Interest to a producer, it will add the previous IBF it received from the producer as the last name component (Figure 5).

C. Protocol Message Exchanges

There are two phases in PSync. In the *Initialization Phase*, a consumer needs to know what data streams to subscribe to and also get the producer's latest IBF (Section III-C1). After receiving the producer's state information, the consumer enters the *Sync Phase* in which it subscribes to some (or all) of the data streams and receives notifications from the producer (Section III-C2).

1) Initialization Phase: Assuming the producer is reachable via the name prefix /⟨routable-prefix⟩, the consumer first sends a Hello Interest to the producer using the name /⟨routable-prefix⟩/psync-hello, as shown in Figure 7. Upon receiving this Hello Interest, the producer will send a Hello Reply with its IBF as the last component in the name and the set of latest data names (one per data stream) in the content. To ensure that every consumer gets the latest producer state, the FreshnessPeriod of the Hello Reply should be very short (e.g., a few seconds). Based on the Hello reply, the consumer then chooses the data streams to subscribe to, retrieves their latest data items, and enters the Sync Phase (Section III-C2). There are two optimizations to reduce the message overhead: if the consumer already knows which name prefixes to subscribe to, it can use a different Hello Interest /⟨routable-prefix⟩/psync-hello/get-IBF to retrieve only the IBF, not the name set; otherwise, the consumer can compress the name set using any existing compression technique.

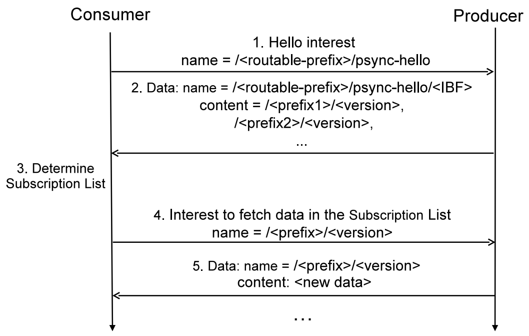

Consumer
Producer
1. Hello interest
name = /<routable-prefix>/psync-hello
2. Data: name = /<routable-prefix>/psync-hello/<IBF>
content = /<prefix1>/<version>,
/<prefix2>/<version>,
...
3. Determine Subscription List
4. Interest to fetch data in the Subscription List
name = /<prefix>/<version>
5. Data: name = /<prefix>/<version>
content: <new data>
...

Fig. 7: Initialization Phase in PSync

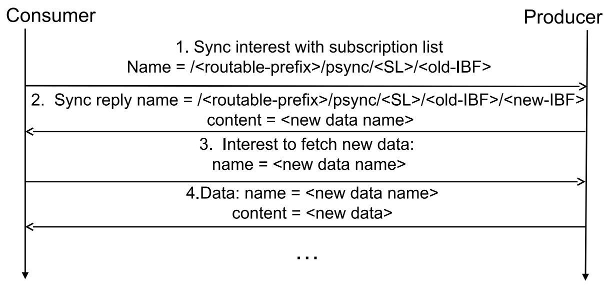

Consumer
Producer
1. Sync interest with subscription list
Name = /<routable-prefix>/psync/<SL>/<old-IBF>
2. Sync reply name = /<routable-prefix>/psync/<SL>/<old-IBF>/<new-IBF>
content = <new data name>
3. Interest to fetch new data:
name = <new data name>
4.Data: name = <new data name>
content = <new data>
...

Fig. 8: Sync Phase in PSync

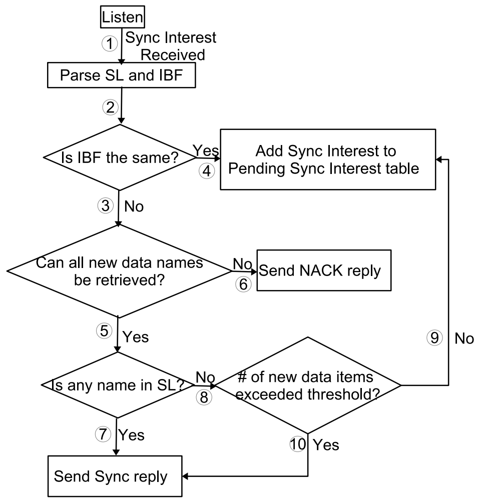
2) Sync Phase: After the initialization phase, the consumer will be able to send a *Sync Interest* to the producer (Figure 8). As explained below, there are three situations that would trigger the producer to send a Sync Reply depending on the IBF value in the Sync Interest.

First, as shown in Figure 9a, if the IBF in the Sync Interest is different from the producer's current IBF, the producer will retrieve all the new data names using the differences between the two IBFs.[2] If any of these new data names matches the consumer's Subscription List, the producer will immediately send a Sync Reply with such new data names. Note that if the total size of the new data items is small, PSync can piggyback the data in the Sync Reply message to eliminate one round trip time to fetch the data.

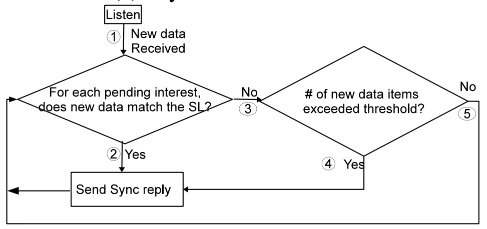
Second, as shown in Figure 9b, if the old IBF and new IBF are the same, the producer keeps the Sync Interest in a table (in the application) . Whenever new data is produced, the producer uses each interest in the table to determine whether this new data is in the consumer's Subscription List and if so sends a Sync Reply. Note that keeping the interest pending at the producer is an optimization since the consumer will resend the interest when the previous one expires. If the producer has a limited amount of memory, it can drop the interest in this case. The trade-off is that the notification of new data may experience some delay up to one sync period (Section III-D).

Third, if the number of new data names has reached a pre-configured maximum, which is set to the IBF size divided by 1.5 in our implementation as explained in Section II-B and Section V-A, and even if none of them are in the consumer's Subscription List, the producer generates a Sync Reply to notify the consumer of its latest IBF (this situation is illustrated in Figure 9a and 9b). This Sync Reply will give the consumer up-to-date knowledge of the producer's IBF, which ensures that the difference between this IBF and producer's future IBF is small enough to be decoded.

[2]If not all the differences between the two IBFs can be retrieved, the producer sends back a NACK Reply and the consumer will start the initialization phase again. This scenario should happen rarely as long as we ensure that the IBF size is 1.5 times the number of differences (see Section V-A).



(a) Sync Interest Received


Listen
1 New data Received
For each pending interest, does new data match the SL?
No 3
# of new data items exceeded threshold?
No 5
2 Yes
4 Yes
Send Sync reply

(b) New Data Received

Fig. 9: Producer's Sync Phase Flow Chart

Whenever a consumer receives a Sync Reply, for each new data name, the consumer checks whether it is in the subscription list (due to the false positive possibility of BFs) and whether its sequence number is indeed new. If so, the consumer will send an Interest to fetch the data. It will also send another Sync Interest, which will either trigger another Sync Reply or stay pending at the producer's side.

D. Sync Message Losses

Sync messages may get lost due to link failures or other problems. In such cases, the consumer will experience a delay in learning any new data matching its Subscription List.

If a Sync Interest from a consumer fails to be delivered, the producer will not send notifications of new data to the consumer. However, after the Sync Interest's lifetime expires, the consumer will send another Sync Interest. Upon receiving this Sync Interest, the producer will process the Interest using the regular procedures discussed earlier. Therefore, if one Sync Interest is lost, the notification is delayed for up to the lifetime of a Sync Interest. The lifetime setting thus needs to take into account the loss rate and the application's tolerance of delay.

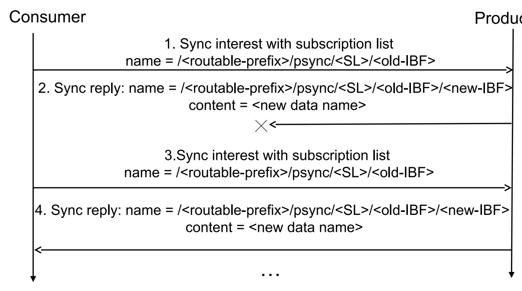
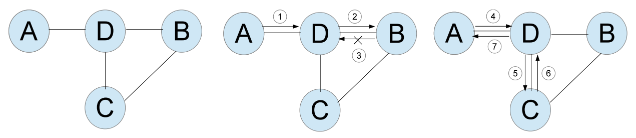

Consumer
Producer
1. Sync interest with subscription list
name = /<routable-prefix>/psync/<SL>/<old-IBF>
2. Sync reply: name = /<routable-prefix>/psync/<SL>/<old-IBF>/<new-IBF>
content = <new data name>
3.Sync interest with subscription list
name = /<routable-prefix>/psync/<SL>/<old-IBF>
4. Sync reply: name = /<routable-prefix>/psync/<SL>/<old-IBF>/<new-IBF>
content = <new data name>
...

Fig. 10: Failure of One Sync Reply



(a) Consumer A with Producer B and C (b) B's Sync Reply gets lost. (c) A syncs with C

Fig. 11: Synchronization with Multiple Producers

The loss of a Sync Reply has a similar effect as illustrated in Figure 10. Note that the second Sync Interest may be satisfied by the content store in an intermediate node where the matching Sync Reply is cached (not shown in the figure).

E. Synchronization with Multiple Producers

In general, it is desirable to have multiple pub-sub servers for robustness and load distribution. *PSync allows one or more consumers to subscribe to any one of multiple producers that host the same data* so that even if some of the producers fail, the consumers can continue to receive subscribed data.

We assume that (1) the producers synchronize their data sets using a *full* data synchronization protocol, (2) they announce the *same Sync name prefix* into the routing system, and (3) Sync Interests are forwarded using a *non-multicast* strategy (e.g., the BestRoute strategy in NFD [9]). For example, in Figure 11(a), consumer A can reach two producers, B and C, which synchronize their datasets with each other. In most cases, A sends Sync Interests to D which are forwarded to B, but due to congestion or link failure, D may forward A's Sync Interest to C. *Our design ensures that A receives subscribed data regardless of which producer receives A's Sync Interest.*

Consider the following scenario in Figure 11(b), A sends a Sync Interest and B responds with a Sync Reply after it produces new data. However, the link between B and D goes down and B's Sync Reply gets lost. Meanwhile, B and C synchronize their dataset so they have the same IBF and C has B's new data. After the Sync Interest expires, A sends another Sync Interest which D forwards to C (Figure 11(c)). C finds that A's knowledge of the IBF and its own IBF are different (because of B's new data). Then C generates a Sync Reply with the new data name along with its IBF. A receives the Sync Reply and then fetches the new data.

F. Simultaneous Updates on Multiple Producers

In the multiple-producer case, if simultaneous updates occur on the producers and different consumers receive Sync Reply messages from different producers, the consumers may be partitioned into several groups where each group has different knowledge of the producers' IBF and data. *Our protocol handles this situation without any special provision*. Due to space constraint, we cannot present the full analysis, but below is a high-level explanation: in each group, the consumers first synchronize with that group's producer to get its new data. Meanwhile, the producers in different groups also synchronize with each other and receive each other's new data. Then they further synchronize with the consumers in their group. This process continues until all the interested consumers have obtained all the new data from the different producers.

G. Full-Data Synchronization

Full data synchronization, in which every participant's subscription covers the entire set of data produced by all the participants, is a *special case* of partial-data synchronization. *PSync can easily support full-data synchronization* with appropriate settings. First, every participant registers the same Sync name prefix to receive Sync Interests from everyone else. Second, every participant's subscription list is set to the entire namespace shared by all the participants (represented by a special name in our implementation). Third, each participant needs to have both a consumer component subscribing to the entire namespace and a producer component generating data and computing IBF. Fourth, the Sync Interests are forwarded using a *multicast* forwarding strategy so that they will be forwarded to all the participants. Note that when all the participants are synchronized, they will generate the same IBF and also receive the same IBF from others so their Sync Interests will be exactly the same. Because NDN nodes aggregate Interests with the same name before forwarding them, there will be only one pending Sync Interest on each link in each direction. Whenever a node produces new data, it will send a Sync Reply with the new IBF and new data name. Other nodes will then fetch the new data.

IV. Implementation

We implemented the proposed PSync protocol in C++ using the ndn-cxx [10] library to ensure compatibility with the NDN Forwarding Daemon (NFD [9]). Both the *initialization phase* and the *sync phase* were implemented.

Since IBFs handle only fixed-length KeyIDs (Section II), we need to do efficient encoding and decoding between variable length data names and KeyIDs in the IBF. After comparing different hash functions, we chose Murmurhash 3 [11] for hashing data names into KeyIDs. Compared with other hashing methods, it has the advantages of supporting different hash sizes, fast hashing speed and fast lookup speed.

We use 32-bit integers for idSum, hashSum and Count in IBF and allow applications to configure the IBF size. To make sure that the Sync Interest and Reply messages will not exceed the maximum NDN packet size, we use *Compressed Bloom Filter [8]* to represent the Subscription List if needed, which may slightly increase the computational cost and the false positive rate of the Bloom Filter.

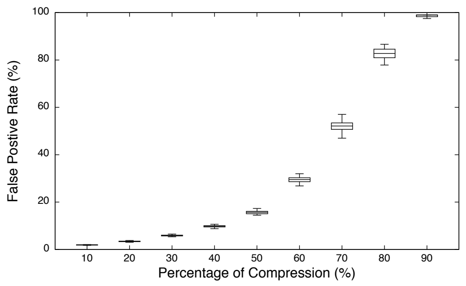


Fig. 12: Compressed Bloom Filter False Positive Rate

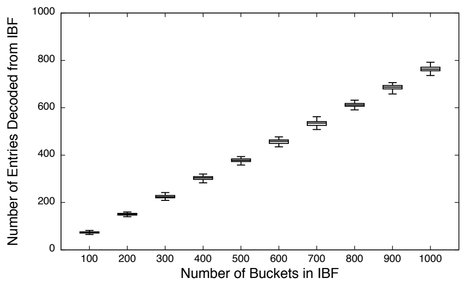


Fig. 13: Number of Entries Decoded from IBF

V. Evaluation

In this section, we first verify the feasibility of employing Compressed Bloom Filter and IBF in PSync. We then evaluate how well PSync can support full-dta and partial-data synchronization as well as a pub-sub application. We performed our evaluatlon in Mini-NDN [12], an emulator that runs multiple NDN nodes on a single machine using real NDN software.

A. Compressed Bloom Filter and IBF

We first investigate how much Compressed Bloom Filters can reduce the size of Subscription Lists without significantly affecting the false positive rate of new data names in Sync Reply messages. We generate a regular Bloom Filter (BF) using 2000 data elements with an expected false positive rate of 1%. We then compress the BF to reduce its size by 10% to 90% and perform 100,000 queries to evaluate the false positive rate. Figure 12 shows that the false positive rate increases to 5.9% and 15.7% with a compression rate of 30% and 50%, respectively. Therefore, we can compress Subscription Lists considerably if applications with strict packet size requirements can tolerate a higher false positive rate.

In the second experiment, we vary the number of buckets in IBF from 100 to 1000 and evaluate the maximum number of data elements that can be decoded. Figure 13 shows that the number of entries decoded from IBF is higher than the number of buckets divided by 1.5. For example, with 1000 number of buckets in IBF, the median number of entries decoded from IBF is around 783. This means that, given a data production rate and Sync time period, we can set the IBF size to be 1.5 times the expected number of new data items in a Sync period to avoid decoding failures.

B. Performance of PSync

We use the Sprint point of presence topology [13] with 52 nodes and 94 links to evaluate the performance of PSync. We focus on the *data fetching delay*, i.e., the time from when a data item is produced to when the data is obtained by a consumer that has subscribed to the data.

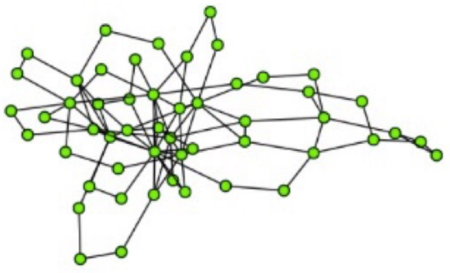
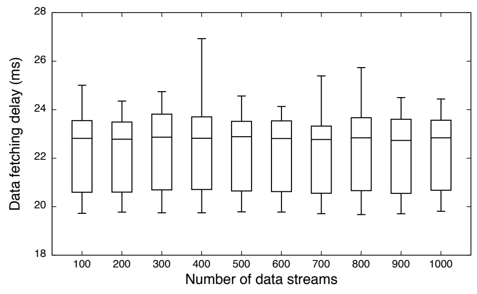
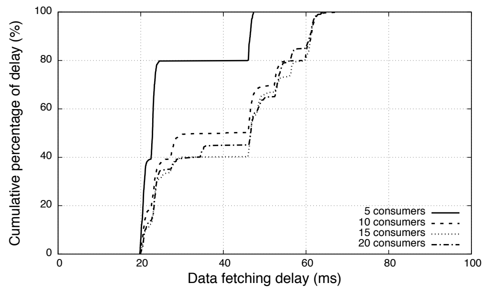


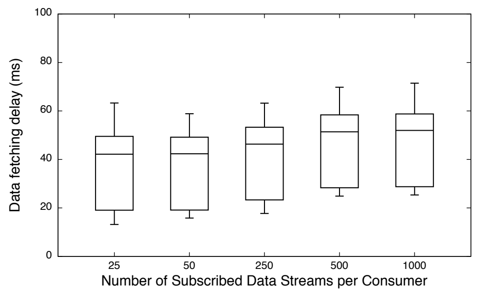
Fig. 14: Sprint Point of Presence Topology



(a) Different Number of Data Streams per Producer



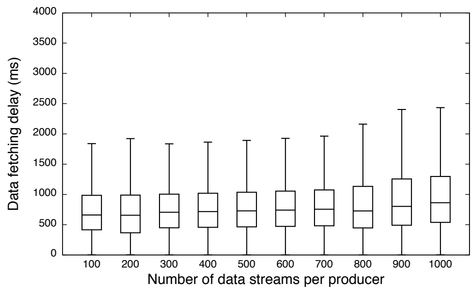
(b) Different Number of Consumers



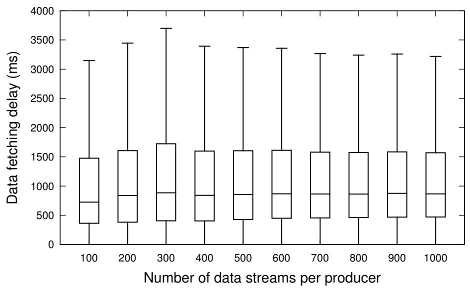
(c) Different Number of Subscribed Data Streams per Consumer

Fig. 15: Data Fetching Delay in Partial-Data Synchronization

1) Partial-Data Synchronization: We first evaluate how well PSync supports *partial-data synchronization*. We pick one of the nodes as a producer and let it serve many data streams, each generating data at a random interval between 1 and 5 minutes. Each consumer subscribes to a random set of 100 data streams. In the first experiment, we randomly pick 5 nodes as consumers and vary the number of data streams served by the producer from 100 to 1000. Figure 15a shows that the median data fetching delay stays around 23ms regardless of



(a) PSync



(b) ChronoSync

Fig. 16: Data Fetching Delay in Full-Data Synchronization

the number of data streams. In the second experiment, we fix the number of data streams to be 1000 and randomly choose 5, 10, 15, and 20 nodes to be the consumers. Figure 15b shows that the data fetching delay increases slightly as the number of consumers increases, but the maximum delay is below 100ms and median delay is below 50ms in all the runs. Finally, we randomly choose 20 nodes as consumers and vary the subscription size per consumer from 25 to 1000 data streams. Again, the results (Figure 15c) show that the data fetching delay changes very little even though the subscription size is increased by a factor of 40.

2) Full-Data Synchronization: We now compare the performance of PSync and ChronoSync [2] in supporting *full-data synchronization*. All the 52 nodes synchronize with each other and we vary the number of data streams produced by each node from 100 to 1000 with each data stream generating data at a random interval between 1 and 5 minutes. Figure 16 shows that PSync can achieve lower data fetching delay than ChronoSync which is specifically designed for full-data synchronization. We believe that this is mainly due to PSync's better handling of simultaneous updates – unlike the Digest in ChronoSync, the IBF in PSync messages allows two nodes in different synchronization states to find their differences instantly through the IBF subtraction operation without going through a long recovery process. A more in-depth analysis of the factors contributing to PSync's better delay performance is part of our future work.

C. Scalability in Supporting Pub-Sub Applications

We implemented a Building Management System (BMS) [14] using PSync. The system consists of multiple data repos (producers) and consumers. Every repo collects data from a number of BMS panels, each generating a data

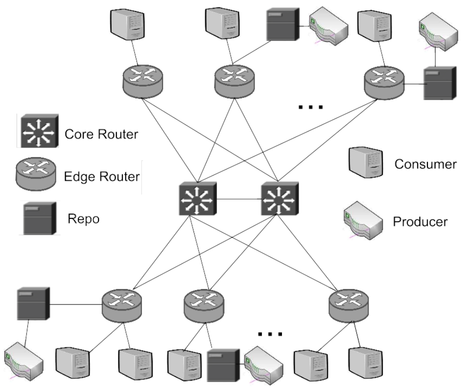


Fig. 17: Evaluation Topology for BMS System (There are two core routers connected to border routers (not shown) that peer with ISPs. Every edge router is connected to both core routers for robustness. The edge routers' locations are generated randomly over a 25 square kilometer region simulating a large campus. Link speed between routers is 1Gbps and propagation delay is calculated using geographic distance divided by $2 * 10^8$ m/s.)

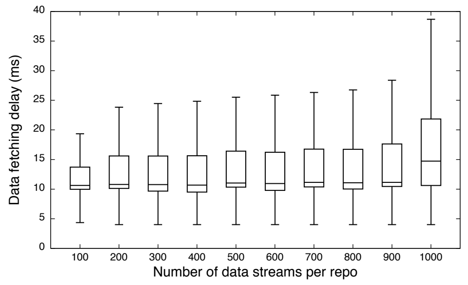


Fig. 18: Data Fetching Delay with Different Number of Data Streams in BMS (UCB Topology)

stream based on sensor readings in a building. The repos use PSync to perform full-data synchronization with each other, and each data consumer also uses PSync to subscribe to a set of data streams hosted by the repos. Figure 17 shows our experiment topology which is based on the UC Berkeley campus core network [15]. The BMS repos and data consumers are connected to the edge routers. Each repo can collect data from BMS panels in different buildings. Each panel generates data with random inter-arrival times between 1 and 5 minutes and inserts the data into the nearest repo (under the best-route forwarding strategy). To evaluate how well the system scales with the number of monitored buildings and end users, our experiments include varying number of data streams and data consumers. The number of repos is set to 5 in all the experiments. Below are our experiment results.

1) Delay vs. Number of Streams: In this experiment, we vary the number of data streams per repo from 100 to 1000 and measure the data fetching delay of 20 consumers, each subscribing to a random set of 200 data streams, over a 10-minute time period. In each scenario, around 14K Data packets are received by the consumers. Results in Figure 18 show that the median delay increases only from 11ms to 14ms while the number of streams grows by a factor of 10.

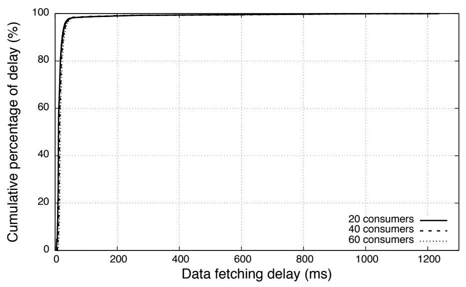


Fig. 19: Data Fetching Delay with Different Number of Data Consumers in BMS (UCB Topology)

2) Delay vs. Number of Consumers: In this experiment, we set the number of data streams to be 1000 per repo and let each consumer randomly subscribe to 200 data streams. We measure the data fetching delay with 20, 40, and 60 consumers. As shown in Figure 19, *the distribution of the data fetching delay stays almost the same when we double or triple the number of consumers*. This is due to two reasons: (1) consumers' packets reach the nearest repo in most cases as we use the BestRoute strategy for synchronization between consumers and repos; and (2) the consumers interact with the repos using PSync, which does not maintain per-consumer state and uses IBF to perform efficient set difference.

VI. Related work

Data synchronization is a fundamental building block in NDN to bridge the gap between the unreliable network layer and application needs. Below is a summary of existing data synchronization research in NDN.

ChronoSync [2] utilizes NDN features to enable full data synchronization among sync participants. It uses a digest tree to represent the latest data at all participants, and a digest log to record previous digests. Each participant can respond to a different digest from another participant with its new data name, thus allowing others to retrieve the new data.

CCNx Sync [16] and iSync [3] also support full-data synchronization. Different from ChronoSync which is a group synchronization protocol, these two protocols are pair-wise synchronization protocols – they operate between two neighbor nodes that use a Merkle tree [17] (CCNx Sync) or IBF (iSync) to detect their differences. By exchanging either the tree digest or IBF with the neighbor, each node can figure out what entries are missing from its own data collections.

The design of PSync has some similarity to existing synchronization protocols in NDN. For example, similar to ChronoSync, it makes use of naming conventions to keep the number of data items in IBF as low as possible. It also makes use of IBF to detect multiple differences in one sync step, as iSync does. However, ChronoSync, iSync and CCNx Sync are intended to synchronize full-data sets, not subscriptions to subsets of data, while PSync supports both types of synchronization.

VII. Conclusion

We have presented PSync, a name-based Sync protocol that supports both partial-data and full-data synchronization in NDN. It represents the latest names in a producer's data streams using an IBF, which allows efficient computation of set differences. By comparing the differences between its old IBF and new IBF, the producer can generate a list of new data names that have been produced in the period between the old and new IBF. Using this list and a consumer's subscription information, the producer can notify the consumer if new data matching the subscription has been produced. We have implemented PSync as a library on the NDN platform and evaluated it using a prototype Building Management System. Our results show that PSync scales well under a variety of conditions. Our future research is to explore the possibility of reducing the Sync Interest/Reply message size as well as evaluating the protocol using more real applications.

Acknowledgment

We are grateful to Ashlesh Gawande for his assistance in the protocol performance evaluation. We also thank Lixia Zhang, Wentao Shang, Hila Ben Abraham and the anonymous reviewers for their helpful comments.

References

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 44, no. 3, pp. 66–73, Jul 2014.

[2] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proceedings of IEEE ICNP*, 2013.

[3] W. Fu, H. Ben Abraham, and P. Crowley, "Synchronizing namespaces with invertible bloom filters," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '15, 2015.

[4] N. Fotiou, P. Nikander, D. Trossen, and G. C. Polyzos, "Developing information networking further: From PSIRP to PURSUIT," in *Broadband Communications, Networks, and Systems*. Springer, 2010, pp. 1–13.

[5] D. Eppstein, M. T. Goodich, F. Uyeda, and G. Varghese, "Whats the Difference Efficient Set Reconciliation without Prior Context," in *Proceedings of ACM SIGCOMM*, 2011.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[8] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking (TON)*, vol. 10, no. 5, pp. 604–612, 2002.

[9] N. P. Team, "NFD - NDN forwarding daemon," http://named-data.net/doc/nfd/.

[10] ——, "ndn-cxx," http://named-data.net/doc/ndn-cxx/.

[11] A. Apppleby, "GitHub C++ Source Code for Murmurhash 3," https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[12] NDN Project Team, "Mini-NDN GitHub," https://github.com/named-data/mini-ndn.

[13] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with rocketfuel," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 1, pp. 2–16, 2004.

[14] W. Shang, Q. Ding, A. Marianantoni, J. Burke, and L. Zhang, "Securing building management systems using named data networking," *IEEE Network*, vol. 28, no. 3, pp. 50–56, 2014.

[15] University of California, Berkeley, "Campus network core topology," https://nettools.net.berkeley.edu/pubtools/legacy/net/netinfo/newmaps/campus-topology.pdf, 2016.

[16] PARC, "Content Centric Networking (CCNx) project website," http://www.ccnx.org.

[17] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology Proceedings*. Springer, 1990, p. 218238.