NDN, Technical Report NDN-0039, 2016. http://named-data.net/techreports.html
Revision 1: [6/9/16]

PartialSync: Efficient Synchronization of a Partial Namespace in NDN

Minsheng Zhang
mzhang4@memphis.edu
University of Memphis

Vince Lehman
vslehman@memphis.edu
University of Memphis

Lan Wang
lanwang@memphis.edu
University of Memphis

***Abstract*—Named Data Networking (NDN) is an evolving Future Internet Architecture where data is a first-class entity. Data synchronization plays an important role in NDN similar to transport protocols in IP. Some distributed 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. In this paper, we propose PartialSync which aims to efficiently address this synchronization problem. We use names in PartialSync messages to carry producer's 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 represent the latest names in a producer's data streams using an Invertible Bloom Filter (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 PartialSync in the NDN codebase and are using it to develop a prototype building management system where users can subscribe to any subset of data generated by the many sensors and actuators in buildings.**

I. Introduction

Data synchronization is becoming a basic requirement for a large number of applications such as Calendar and Dropbox. Some applications require that everyone in a synchronization group receive all the new data produced by everyone else. For example, participants in a chat group typically receive everyone's chat messages. If we consider the data produced by everyone as the full dataset, we can call this a *full data synchronization* problem. In other applications, such as news and weather subscription services, each user may be interested in a subset of the produced data and receive new data only related to their interests. For example, a consumer may only wish to know the most up-to-date weather information for the cities she and her relatives reside in. We call this a *partial data synchronization* problem. In fact, it is a generalization of the full data synchronization problem, as a user' interests (or subscriptions) can be anywhere from an empty set to a full set of the data.

To further illustrate this demand for partial synchronization, consider a smart-phone app store. Users of mobile devices often use app stores to download applications to their phone, but due to their mobile device's limited storage capacity, the user may only install a small portion of the applications 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 users should not have to check the app store periodically for *every installed app* to get the updates, and (c) the users should be able to synchronize with *any app store* that has the same set of apps. These requirements are applicable to any application that has the partial synchronization problem.

The above requirements essentially mean that the data producers should avoid per-consumer state and consumers should avoid per-producer and per-subscription state, which is especially challenging in the current TCP/IP network architecture where any communication between producers and consumers requires end point identifiers (e.g., IP address) so state information about each other is unavoidable.

We look at the partial synchronization problem from the perspective of the Named Data Networking (NDN) architecture [11], which makes *immutable data with hierarchical names and producer signatures* a common abstraction of 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 determine how to forward consumers' requests and cache returned data for any future requests. This design is much more suitable and efficient for today's increasingly data-centric applications. Moreover, since data is the focus of NDN, communication is no longer between specific producer and consumer thus making it possible to address the above requirements of partial synchronization.

In this paper, we propose a protocol called *PartialSync* for synchronizing a partial namespace in NDN. PartialSync uses Invertible Bloom Filters (IBF) [3] to represent the latest data names in a namespace and utilizes the subtraction operation in 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 subscription information from consumers, 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 PartialSync

This work was supported by NSF Grant 1344495 and a Cisco research grant.

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) *scalability under large number of subscriptions*: efficient data representations such as Bloom Filters (BF) [2] and ranges are used to efficiently encode consumers' subscriptions so only one PartialSync Interest message is sent from each consumer regardless of how many name prefixes the consumer subscribes to; and (c) *robustness under producer failures*: because producers are stateless with respect to consumers[1], each consumer can synchronize with any producer that replicates the same data.

The paper is organized as follows. Section II gives an introduction of the NDN architecture, bloom filters, and bloom filter extensions. Sections III and IV present the design and implementation of the proposed PartialSync protocol. In Section V, we present results of our performance evaluation. Section VI presents related work and Section VII proposes future work and concludes the paper.

II. Background

In this section, we briefly review the NDN architecture [11] and Bloom Filters [2].

1) NDN architecture: Named Data Networking (NDN) [11] is a new data-centric Internet architecture that naturally supports efficient and secure data distribution. NDN uses two different types of packets, Interest packets and Data packets, which are used to request and return named content, respectively. NDN also binds a data packet's name and content using a producer's key, so a receiver can verify the authenticity of the data packet.

When an interest arrives at a router, if there is no cached data matching the interest or 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 which interface it was forwarded to. When the interest reaches the producer, the data matching the interest 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.

2) Bloom filter and extensions: A Bloom Filter [2] is an efficient data structure that uses a bit array to succinctly represent a dataset. Bloom Filters allow for queries of whether an element is in the dataset or not. Bloom Filters use a list of 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 and each bit position is checked for a 1. If each bit position is set to 1, the Bloom Filter shows the element as a member of the set. But, due to the compactness of the data structure, answers to membership queries may not be correct, as elements may be hashed to some of the same bits. It is important to note incorrect membership answers will only be false positives (not false negatives). 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 [2]. 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 [2].

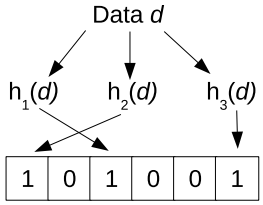

Fig. 1. Data *d* is passed to each hash function which maps the data to a bit position in the bloom filter.

There are some drawbacks to using a Bloom filter. In particular, there is no 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) [4]. 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) [3] introduce a new data structure to solve this problem. Instead of maintaining a bit array to represent set membership, IBFs use the hash functions to map the element to cells that maintain keys, values, and a count for each element mapped to the cell. Each cell maintains an idSum and a hashSum to track the keys and data mapped to the cell, respectively. The idSum tracks inserted elements' values and hashSum tracks hashes of inserted element's values. The cell also maintains a count similar to CBFs.

[1]All the required state information for producers is carried in PartialSync Interest messages.

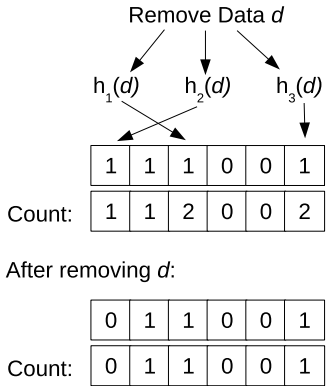

Fig. 2. Data d is removed from the Counting Bloom Filter. Note that although d maps to three bit positions, only one of the bit positions is set to 0 due to another element being mapped to two of the same positions.

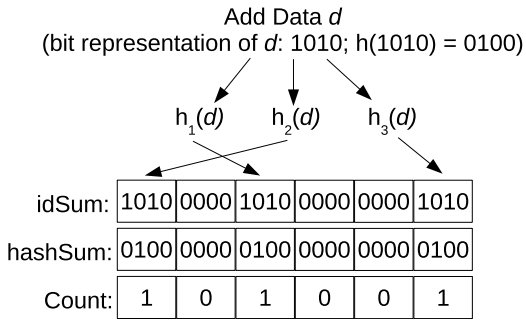

Fig. 3. Data d is added to the Invertible Bloom Filter.

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 key, 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.

Using the XOR operation to maintain multiple elements' combined keys and values allows elements to be removed from cells. 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. When a pure cell is found, it is highly likely that the cell's idSum represents a single element. This element can be added to the retrieval list and the element can be 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 [3].

IBFs also support a set difference operation through subtraction – for each cell in two IBFs, the corresponding count bits are subtracted and both the idSum and hashSum are XOR'ed. Resulting pure cells will have a count of 1 or -1, and the difference between the two sets can be determined as follows. Suppose we calculate $IBF_1 - IBF_2$, a pure cell with a count of 1 means that it contains an element only in IBF_1 while a count of -1 indicates that the element is only in IBF_2. Along with the idSum and HashSum from the subtraction operation, we can determine the specific different elements in the two IBFs as well as to which IBF each different element belongs.

Since only fixed-length numbers (KeyID) can be insert into an IBF, if we want to insert other types of elements such as NDN names, we first need to hash each element into a fixed-length number (KeyID) and then keep a mapping table to associate each KeyID with the original element identifier. 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 element identifier in the mapping table using the KeyID.

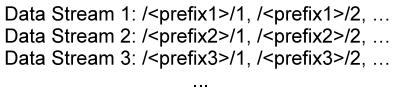

Fig. 4. Data Streams

III. Design

In this section, we first give an overview of the PartialSync design and then introduce how state information is encoded at each consumer and producer. Next, we present our protocol design and illustrate it in failure scenarios. Finally, we describe our support of synchronization with multiple producers and discuss simultaneous update generation in this scenario.

A. Overview

We assume that each producer produces a set of data streams with different name prefixes and each consumer is interested in a subset of the data streams – a data stream is set of data which have the same name prefixes but different sequence number (Figure 4). For example, a building management system may have electricity data under ⟨prefix⟩ = /⟨building⟩/Electricity, which contains data streams with name prefixes of the form /⟨prefix⟩/⟨panel⟩/⟨device⟩. Suppose a consumer is interested only in the data from the devices connected to Panel 2 in Building 1, it can subscribe to those devices' name prefixes, e.g., /Building1/Electricity/Panel2/heater, through PartialSync so that it will be informed whenever new data points are generated under those name prefixes.

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 5). The Sync Interest contains the consumer's subscription list which is used by the producer to check for updates to the subscribed data. If any data stream in the consumer's subscription list has new data items,

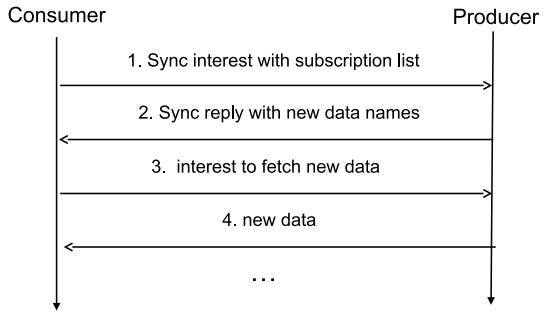

Fig. 5. Basic Design Concept of PartialSync

the producer will generate a Sync Reply to the consumer containing a list of new data names (Step 2 in Figure 5).

Upon receiving the Sync Reply, for each new data name, the consumer will further check whether the data name indeed belongs to its own subscription list. If the 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 5). If no new data matching the consumer's subscription list has been produced when the Sync Interest is received, the Interest will stay pending for its lifetime and 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.

B. Data Representation

PartialSync uses a number of representations including *Bloom Filters (BF)* and *ranges* for consumers to express their subscriptions, i.e, *Subscription List*, in their Sync Interests. Moreover, it uses *Invertible Bloom Filters (IBF)* to represent producers' latest dataset, 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. 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 $P = Q$, that can be encoded using simpler representations than Bloom Filters. When the consumer sends a Sync Interest message, it selects the most compact format for its subscription list and sends the format information along with the encoded subscription list to the producer so that the producer can decode correctly.

While a regular BF can be used to represent a subscription list, the consumer should use a Counting Bloom Filter (CBF) *locally* to support deletion of subscriptions efficiently (regular BFs support only insertions). However, when sending its subscription list to a producer (Figure 7), the consumer needs to send only the regular BF (i.e., not including the count array maintained in the CBF), since the producer only needs to check whether a given data name prefix is subscribed to by the consumer.

2) Producer State: PartialSync adopts ChronoSync's approach of letting each producer name data sequentially [12]. The latest dataset can be represented by an IBF which contains only one data name from each data stream, which is 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 $i+1$, PartialSync will remove the name p/i from the IBF and add $p/(i+1)$ to the IBF. In doing so, the IBF contains only N items, where N is the number of data streams.

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 last IBF it received from the producer as an additional name component (Figure 7). Upon receiving the Sync Interest, the producer can easily determine names of new data that have been produced from the difference between the IBF in the consumer's Sync Interest and its own IBF. The producer can then further check whether these new data items are in the consumer's subscription list.

C. Protocol Design

There are two phases in PartialSync. 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 new data in the data streams and synchronizes with 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 6. Upon receiving this Hello Interest, the producer will send a Hello Reply with the latest names in its data streams. 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).

To ensure that every consumer gets the latest producer state, we set the cache policy of the Hello Reply to be NO_CACHE. If there is a failure in the transmission of the Hello Interest or the Hello Reply, the consumer will send another Hello Interest once the previous one expires.

2) Sync Phase: After the initialization phase, the consumer will be able to send a *Sync Interest* to the producer – when a new data item in the consumer's subscription list is produced on the producer, the consumer will receive a Sync Reply containing this new data item's name (Figure 7). As explained

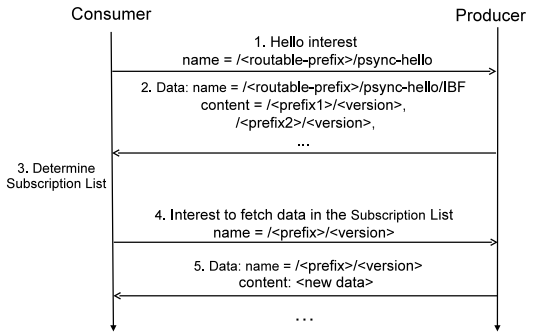

Fig. 6. Initialization Phase in PartialSync

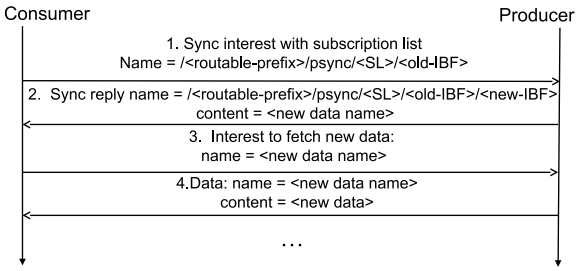

Fig. 7. Sync Phase in PartialSync (*SL* represents the consumer's subscription list; *old-IBF* and *new-IBF* represent the producer's old state and current state encoded in IBF.)

below, there are three situations that would trigger the producer to send a Sync Reply to consumer, depending on the IBF value in the Sync Interest.

First, as illustrated in Figure 8(a), if the IBF in the Sync Interest from the consumer is different from the producer's current IBF, the producer tries to retrieve all new data names between the two IBFs.[2] If any of these new data names is in the consumer's Subscription List, producer will immediately send a Sync Reply with these new data names that match the consumer's subscription.

Second, as shown in Figure 8(b), if the old IBF and new IBF are same, the producer keeps the Sync Interest in a Pending Interest table. Whenever new data is produced, the producer uses each Interest in the Pending Interest table to determine whether this new data is in the consumer's Subscription List and if so sends a Sync Reply.

Third, if the number of new data names has exceeded a pre-configured maximum in this period 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 8(a) and 8(b)). This Sync Reply will give the consumer update-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 names can be retrieved from the differences of the two IBFs, then producer sends back a NACK Reply to ask the consumer to restart the PartialSync process again as it cannot figure out the actual differences in the certain period.

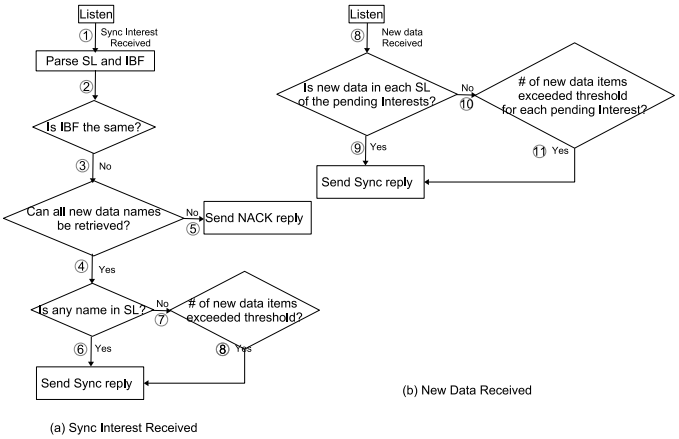

Fig. 8. Producer's Sync Phase Flow Chart

Whenever a consumer receives 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 a Sync Reply immediately or stay pending at the producer's side.

D. Failure Handling

Since NDN has two different kinds of packets, there should be two different failure scenarios: (a) failure in transmitting a Sync Interest; and (b) failure in transmitting a Sync Reply. Both types of failure have the same result – the consumer will experience a delay in learning any new data matching its Subscription List.

1) Sync Interest Transmission Failure: If a Sync Interest fails to be delivered, the producer will not receive the consumer's Sync Interest and will not send notifications of new data. 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.

2) Sync Reply Transmission Failure: Figure 9 illustrates a failure in transmitting a Sync Reply. After the first Sync Interest expires, the consumer will send another sync Interest. This Sync Interest may be satisfied by data in an intermediate node's the content store (from a previous Sync Reply) or reach the producer which can then return any new data names. Again in this case, the consumer is able to synchronize with the producer after some delay.

E. Synchronization with Multiple Producers

Since NDN is data-centric, not host-centric as in IP, we want our protocol to be able to synchronize with multiple producers which produce the same data. For example, in

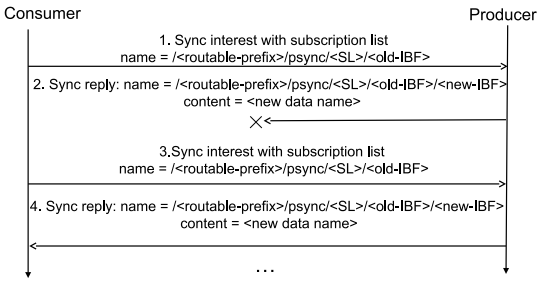

Fig. 9. Failure of One Sync Reply

Figure 10(a), consumer A can reach two producers, B and C, which synchronize full datasets with one another using a dataset synchronization protocol (e.g. ChronoSync [12]). 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.

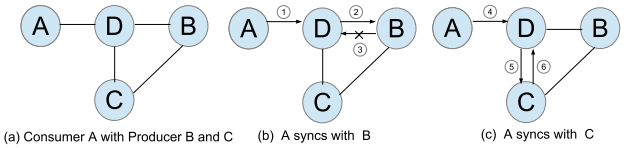

Fig. 10. Synchronization with Multiple Producers

Consider the following scenario in Figure 10(b), A sends a Sync Interest to B. After new data is produced at B, B responds to A's Sync Interest because this data matches A's Subscription List. However, the link between B and D goes down and A fails to receive the Sync Reply sent from B. Meanwhile, B and C synchronize their dataset. After the previous Sync Interest expires, A sends another Sync Interest which D forwards to C (Figure 10(c)). Upon receiving the Sync Interest from A, C finds that A's knowledge of the IBF and its own IBF are different, because of the new data produced at B. It generates a Sync Reply with the new data name and sends it to A along with its IBF. After receiving the Sync Reply and fetching the new data, A is synchronized.

F. Simultaneous Updates on Multiple Producers

If simultaneous updates occur on multiple producers and all of them have received the same Sync Interest from one consumer, they may all respond to the Sync Interest. In NDN, since consumers receive only one data packet for each Interest, the consumer will only receive one Sync Reply. Thus, the network is partitioned into different groups where each group has different knowledge of the producers' IBF and data.

This situation is illustrated in Figure 11 where A and B are consumers who subscribe to the same dataset produced by both C and D. At the beginning (Figure 11(a)), A and B have the same knowledge about the producers' IBF (IBF_1), and C and D are synchronized and have the same IBF (IBF_1). By chance, C and D simultaneously produce new data to which A and B are both subscribed. Then C's IBF changes to IBF_2 and D's IBF to IBF_3. C and D will respond to A and B's Sync Interests. Suppose D's Sync Reply reaches B while C's Sync Reply reaches A (Figure 11(b)), then A and B will now have different producer IBF, i.e., IBF_2 for A and IBF_3 for B.

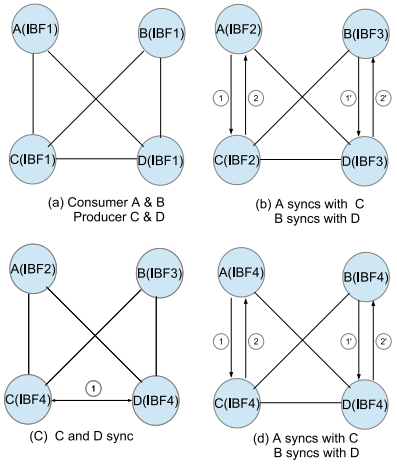

Fig. 11. Handling Simultaneous Updates

Our protocol handles this situation without any special provision. As previously illustrated, A and B first synchronize with C and D, respectively, so A receives C's new data and B receives D's new data. Meanwhile, C and D synchronize with each other (Figure 11(c)), so they should have each other's new data as well as the same IBF (IBF_4). Upon receiving the new Sync Interest from A and B, C and D generate Sync Reply according to the difference between the IBF in the Sync Interest and their own IBF. In this case, the Sync Reply from C to A will contain the name of D's new data and the Sync Reply from D to B will contain the name of C's new data. Upon receiving the Sync Reply message, A and B fetch the new data accordingly. In the end, they both receive the new data produced by C and D, as well as the producers' latest IBF (IBF_4) (Figure 11(d)).

IV. Implementation

We implemented the proposed PartialSync protocol in C++ using the ndn-cxx [8] library to ensure compatibility with the NDN Forwarding Daemon (NFD [9]). Both the *initialization phase* and the *sync phase* were implemented, and we evaluated the protocol in Mini-NDN [7], an NDN emulator.

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 for hashing data names into element IDs. 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 packet will not exceed the maximum NDN packet size, we use Compressed Bloom Filter [6] to represent the Subscription List if needed, which may slightly increase in the computational cost and the false positive rate of the Bloom Filter.

V. Evaluation

TODO: evaluation results will be added in the next revision.

VI. Related work

Data synchronization is a fundamental building block in NDN to bridge the gap between the unreliable network layer and the needs of the applications. The design of PartialSync has some similarity with existing synchronization protocols in NDN. For example, it makes use of naming conventions to keep the number of data items in IBF as low as possible similar to ChronoSync [12], and it makes use of IBF to easily detect multiple updates in one sync step, as does iSync [5]. However, ChronoSync and iSync are intended to synchronize full data sets but not to synchronize subscriptions to subsets of data, as PartialSync does. Because of the different objectives, PartialSync uses Bloom Filters and other data structures to express subscriptions, so that producers only inform a consumer the updates on the data the consumer is interested in. Furthermore, It encodes the consumer state in PartialSync interests, which frees producers from keeping consumer state and allows consumers to sync with any of the producers that replicate the same data set.

VII. Conclusion

We have developed the PartialSync Protocol to solve the partial synchronization problem in NDN where a consumer's Subscription List can be anything from an empty set to a full set of the data. It utilizes Bloom Filter and its extensions to identify new data names at the producer's side. This protocol is designed to scale well with large subscription lists, large number of consumers and multiple producers. We have evaluated our protocol using a prototype building management system in Mini-NDN. Our next step is to explore the possibility of reducing the Sync Interest/Reply message size as well as evaluating the protocol using more real applications.

References

[1] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Li, S. Mastorakis, Y. Huang, J. P. Abraham, S. DiBenedetto, C. Fan, C. Papadopoulos, D. Pesavento, G. Grassi, G. Pau, H. Zhang, T. Song, H. Yuan, H. B. Abraham, P. Crowley, S. O. Amin, V. Lehman, , and L. Wang. NFD developer's guide. Technical report, Technical Report NDN-0021 (revision 6), NDN Project, 2016.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] D. Eppstein, M. T. Goodich, F. Uyeda, and G. Varghese. What's the difference: Efficient set reconciliation without prior context. In *Proceedings of ACM SIGCOMM*, 2011.

[4] 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*, 8(3):281–293, Jun 2000.

[5] 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*, ANCS '15, 2015.

[6] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

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

[8] NDN Project Team. ndn-cxx. http://named-data.net/doc/ndn-cxx/.

[9] NDN Project Team. NFD - NDN forwarding daemon. http://named-data.net/doc/nfd/.

[10] A. Partow. General purpose hash function algorithms (open Bloom Filter source code (c++)). http://www.partow.net/programming/hashfunctions/.

[11] 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)*, 44(3):66–73, Jul 2014.

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