NDN Technical Report NDN-0053. http://named-data.net/techreports.html
Revision 1: May 15, 2017

A Survey of Distributed Dataset Synchronization in Named Data Networking

Wentao Shang,[*] Yingdi Yu,[*] Lijing Wang,[†] Alexander Afanasyev,[*] and Lixia Zhang[*]
[*]UCLA, [†]Tsinghua University
{wentao,yingdi,aa,lixia}@cs.ucla.edu, wanglj11@mails.tsinghua.edu.cn

***Abstract*—Distributed synchronization of a shared dataset (sync for short) provides a powerful abstraction for connection-agnostic multi-party communication in NDN. In recent years, several sync protocols have been proposed, each featuring different design choices in data naming, namespace representation, and state propagation mechanisms, which lead to different design tradeoffs. In this report, we survey these protocols and, through detailed analysis and side-by-side comparisons, highlight their commonalities and fundamental differences. We also articulate the remaining issues that must be addressed to make the sync protocols available to all applications, shedding the light on future work directions in this important area of NDN architecture research.**

I. Introduction

Named Data Networking (NDN) [1], [2] is a proposed new Internet architecture that shifts the communication model from host-centric, as in today's TCP/IP networks, to data-centric. At the network layer, NDN provides the communication primitive that allows a data consumer to send an Interest packet with a name or name prefix and retrieve a Data packet which is named under that prefix and can be verified. This use of Interest-Data data exchange to retrieve desired data differs from data retrieval via a TCP connection in three fundamental ways. First, it supports data retrieval among multiple parties, while TCP supports data exchange between two parties only. Second, it does not require all communicating parties to be inter-connected at the same time as TCP does. Third, it does not care from where the data is returned since the security is attached to the data instead of its container or communication channel.

Today's Internet applications often require much more sophisticated communication models, which typically involve some form of data or state sharing and synchronization among multiple parties. For example, file sharing, collaborative editing, and group messaging all collect and distribute state and data among groups of participants. However, with today's TCP/IP network architecture, whenever communication involves more than two parties, the applications have to rely on (at least logically) centralized infrastructure to support multiparty dataset synchronization.

The data-centric nature of the NDN architecture provides a foundation for dataset synchronization (*sync* for short) in a completely distribution fashion. Sync can be implemented on top of NDN's Interest-Data exchange primitives to provide an important layer of abstraction to support distributed applications and services. Applications that use sync can easily publish data without worrying about how others would discover the newly available data, and can easily consume up-to-date information without worrying about how and from where to get it.

NDN can achieve distributed dataset synchronization by synchronizing the *namespace* of the shared dataset among a group of distributed nodes (called *sync nodes*). To share new data, a producing (side of) applications injects its names into the dataset. After learning the new names, the consumer (sides of) application decides whether to fetch the new data according to its own needs and available resources. One may view sync as playing a *transport* layer role in the NDN architecture, bridging the gap between the functionality required by the distributed applications and the one-Interest-one-Data datagram retrieval semantics offered by NDN network-layer primitives.

In this report, we present a survey of the distributed data synchronization protocols that have been developed for the NDN architecture in recent years. They include CCNx (pre-1.0) Sync [3], iSync [4], CCNx 1.0 Sync [5], ChronoSync [6], RoundSync [7], and PSync (or PartialSync) [8]. We identify four most important design questions in a sync protocol design, and examine each of the above five protocols by how it answers these design questions, to identify common design patterns as well as differences in the approaches. We conclude this survey with a discussion on a set of identified remaining issues.

II. Existing Sync Protocols Over NDN

In this section, we examine the set of existing sync protocols that have been developed for the NDN architecture. Our goal is to extract common design patterns for NDN sync protocols and identify different design choices and tradeoffs made in different protocols. Our analysis focuses on the following key design aspects:

a) ***Data naming:*** Thanks to the unique binding between names and immutable data object in NDN, a shared dataset can be uniquely identified by the namespace containing the hierarchical names of all data packets in the dataset. Therefore the dataset synchronization problem in NDN is conveniently reduced to the synchronization of the corresponding namespace. The data packets published by a sync node in the shared dataset are typically named under the topological prefix of the network where the node resides so that the Interests carrying the data names will be forwarded to the data producers by the NDN network.

Besides the namespace of the shared dataset, sync protocols also require a separate group communication namespace for the sync nodes to publish and exchange protocol messages. Due to the nature of group communication, this namespace is typically under a multicast prefix shared by the entire sync group.

*b) **Sync state representation**:* The data structure that represents the state of the shared dataset namespace is often referred to as the *sync state*. Every sync node keeps a local copy of the sync state and uses the sync protocol to keep up with the changes generated by other nodes in the sync group. This requires the sync state to encode the namespace accurately without loss of information and allow sync nodes to reconcile the differences in the shared namespace between distinct states.

*c) **State change notification**:* When a sync node publishes new data in the shared dataset, it needs to notify the rest of the group about the sync state changes in order to keep the group synchronized. The notification contains either the complete information about the new updates or a summary of the updates.

*d) **State update retrieval**:* If the state change notification carries only a summary of the new updates, the sync node needs to further retrieve the complete updates in separate messages. Depending on the state representation, retrieving the updates may range from a straightforward single step to a multi-step process.

We summarize the commonalities and differences among the five protocols in Table I. In the rest of this section, we describe each sync protocol by focusing on the four design aspects mentioned above. At the end of the section we give a brief summary that highlights common design patterns and provide preliminary comparison on the efficiency issues including synchronization delay and protocol message size. (An extended evaluation of existing sync protocols will be reported in future revisions.)

A. CCNx (pre-1.0) Sync

The pre-1.0 CCNx Sync protocol [3] is the earliest synchronization solution proposed for the NDN/CCN architecture as a service module of the *ccnr* repo daemon. CCNx Sync allows a set of repos to synchronize a shared data collection with arbitrary names. To allow efficient synchronization, CCNx Sync represent the dataset as a *sync tree*, where each node in the sync tree is associated with a hash value. For leaf nodes, it is a hash of the corresponding data name; for non-leaf nodes, it is the sum of hashes of children nodes. For example, in Fig. 1, $H_3 = Hash(/a/b/1)$, $H_2 = H_5 + H_6 + H_7$, and $H_0 = H_1 + H_2$. The root hash (H_0 in Fig. 1) then provides a summary of the entire namespace (i.e., sum of all data name hashes). Note that the sum of hashes is not a cryptographically strong summary: in certain cases two sync trees may store different set of names but happen to have the same root hash.

Any producer connected to a repo can publish new data into the shared dataset at any time. The sync module in the repo daemon (called *sync agent*) keeps track of the insertions of new data and updates the sync tree accordingly, adjusting the

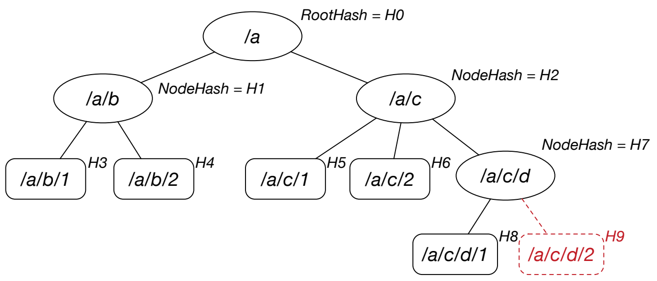

Fig. 1: Example of a sync tree in CCNx Sync

hash values along the path from the new leaf node to the root. For example, in Figure 1 the insertion of a new data "`/a/c/d/2`" (marked as the red dashed square at the bottom right) will cause the sync agent to update the hash values of the nodes "`/a/c/d`", "`/a/c`", eventually propagating the change up to the root node "`/a`".

The sync agent periodically advertises the latest root hash by sending a *RootAdvice* Interest to all the other repos that store the same data collection. The RootAdvice Interest name starts with a multicast prefix for the sync tree, which is shared by all repos, followed by the current root hash of the sync tree.[1] When a sync agent receives a remote root hash that is different from its own, it replies to this RootAdvice with its own root hash. The sync agent who receives a RootAdvice reply will send a *NodeFetch* Interest, which is also named under the multicast prefix of the sync tree, to retrieve the list of hashes for all the children under the root node of the remote sync tree. The NodeFetch process is recursively applied to all the nodes in the sync tree, skipping those with the same hash value, until all nodes with different hash values have been visited. Once it learns the names of the new data from the leaf nodes, the sync agent can fetch those data from the remote repo and insert that data to its local copy of the shared dataset. An example of the synchronization process in CCNx Sync (triggered by the update to the sync tree shown in Fig. 1) is illustrated in Fig. 2. Note that while we show the sync protocol messages only between two repos for clarity, the RootAdvise and NodeFetch Interests actually carry multicast prefix and will be received by all repos storing the same data collection.

One problem in the update propagation mechanism in CCNx Sync is that when multiple repos publish new data simultaneously, there will be more than one reply to a RootAdvice Interest and only one of them will be returned to the Interest issuer. In such case, the sync agent who sends the initial RootAdvice Interest need to issue additional Interests to fetch all of the replies. The proposed solution is to attach *exclude filters* to the Interest to list the root hashes of the remote sync trees that have already been received. This ensures that each unique remote sync tree is examined only once for missing data.

A side-effect of the CCNx Sync algorithm, which compares the local and remote sync trees and updates the local state to be the union of the two, is that the repo cannot remove

[1]In Section III-A we discuss the challenges of supporting Interest multicast in the network.

TABLE I: Comparison of existing sync protocols in NDN

	CCNx Sync	iSync	CCNx1.0 Sync	ChronoSync	RoundSync	PSync
Synchronized Namespace	Arbitrary names	Arbitrary names	Arbitrary names	Node prefix + seq#	Node prefix + seq#	Stream prefix + seq#
Sync state representation	Hash tree	IBF of hashes of names	Manifest storing names or digests of data	List of {prefix : seq#}	List of {prefix : seq#} + round log	IBF of hashes of names with highest seq#
State change notification	Data replying to *RootAdvice* Interest with local root hash	Interest carrying digest of IBF	Interest carrying hash of manifest	Data replying to *Sync Interest* with updates	*Sync Interest* carrying digest of current round	Data replying to *Sync Interest* with new IBF
State update retrieval	*NodeFetch* Interest retrieving child node hashes	Interest retrieving IBF content	Interest retrieving manifest	Data replying to *Sync Interest* with updates	Data replying to *Data Interest* with updates in current round	Data replying to *Sync Interest* with new IBF
Min RTT for data propagation	$\leq$ depth of sync tree	3.5	2.5	1.5 if no simultaneous data	1.5 if no simultaneous data	1.5

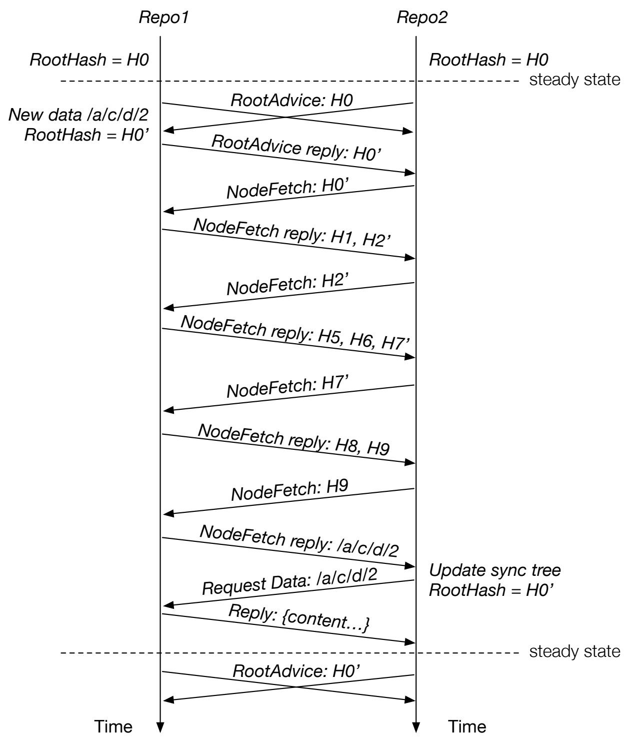

Fig. 2: Synchronization in CCNx (pre-1.0) sync

any data once it is added to the shared dataset. This is because the algorithm cannot distinguish the case where a repo intentionally removed a piece of received data from the case where the repo has never received the data before. As a result, the shared dataset must be monotonically growing, which creates usability issues with the applications who generate a large mount of data and need to perform garbage collection periodically to reclaim the storage. For example, when the NDNVideo application [9] was deployed on top of CCNx repo to publish live video streams, the operator had to cleanup the data and restart all repo instances every day at midnight in order to avoid overwhelming the storage of the test server.

B. *iSync*

iSync [4] is an optimized version of the CCNx Sync. Similar to CCNx Sync, it supports the synchronization of shared data with arbitrary names. However, instead of just representing the dataset as a sync tree, iSync adds the use of the Invertible Bloom Filters (IBF) [10] to store all the names from the shared dataset in compressed form. Since the IBF can only store fixed-length items, the data names must be first mapped to fixed-length IDs (generated from the hash of the names) before they are added to the IBF. For this purpose, each iSync node also maintains a bi-directional ID-name mapping.

Different from CCNx Sync,[2] iSync uses "digest broadcast" Interests (equivalent to the RootAdvise Interest in CCNx) as a notification mechanism for a repo to advertise its current state to other repos, rather than a solicitation for different sync states. The notification Interest carries the digest of the current IBF from the sending repo. When a repo receives a digest different from its own, it sends another Interest to request the corresponding IBF content. After it receives the IBF from a remote repo, the repo subtracts its own IBF from the remote IBF and extract individual IDs from the resulting "diff" IBF. Once the repo extracts all new IDs, it issues Interests to request the original NDN names corresponding to those IDs and then fetches the new data using the original names. Note that in iSync the repo does not expect any reply to the initial notification Interest it sends. It therefore gets around the issue with multiple replies generated for the same broadcast or multicast Interest.[3]

[2]The original iSync paper [4] describes the CCNx Sync protocol differently compared to the official specification [3] released in the CCNx source code package.

[3]Note that this causes imbalance in the Interest-Data packet flow in the network. Assuming the number of notification Interests is negligible compared to the total network traffic, the impact can be a tolerable amount of waste of PIT entries that eventually expire and get removed.

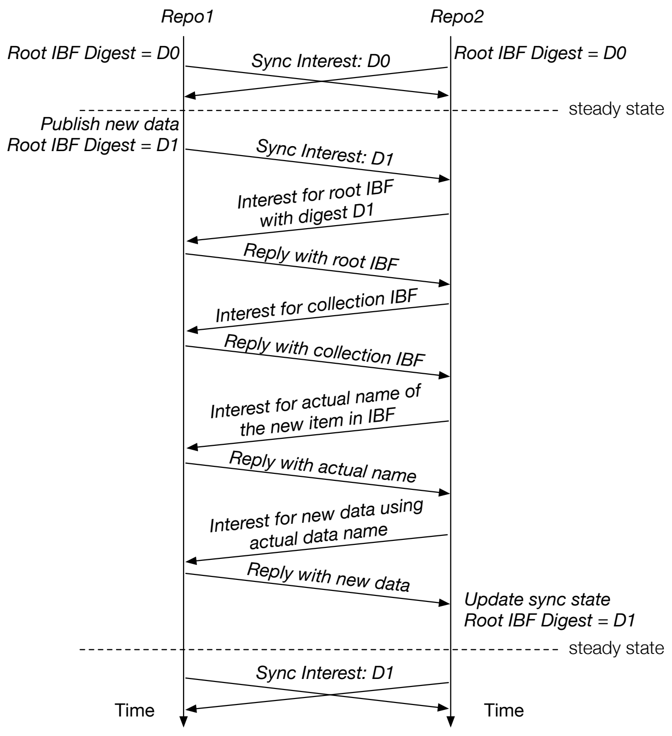

Fig. 3: Synchronization process in iSync

A major limitation in the IBF data structure is that it can losslessly encode up to a certain number of items, beyond which some of the stored items cannot be extracted. Unfortunately, it is not uncommon that during the synchronization process the set difference between the namespace of some repos may contain too many IDs that cannot be encoded in the IBF in a lossless fashion. iSync provides several ways to control the size of the set difference at multiple levels in the protocol design. First, the shared dataset is divided into multiple collections that host data for different applications; each collection maintains its own IBF independently from others. Second, iSync protocol enforces each repo to periodically advertise its local sync state and resolve the difference, which bounds the delay of the data propagation and the size of the set difference between any two repos. Third, iSync creates multiple *local IBFs* to record the small-step changes during each sync period; if the advertised IBF (called *global IBF*) contains too many changes, the repo can fetch the local IBFs instead and perform more fine-grained difference reconciliation.

An example of the synchronization process between two repos in iSync is shown in Fig. 3.

C. CCNx 1.0 Sync

The design proposal of CCNx 1.0 Sync [5] abandons the old-version CCNx Sync design and proposes a simple manifest-based solution.[4] The manifest packets are named under a routable *data collection prefix* announced by every sync node, followed by the hash of the manifest and segment numbers. The manifest contains the SHA256 hashes or the exact names of all data objects in the shared data collection. When the SHA256 hashes are used, the names of the data objects are constructed by appending the hash value to the same data collection prefix in the manifest name. The application-layer data (with real application names) may be encapsulated in those data objects.

[4]The authors are not aware of any real implementation of this new design.

Each sync node uses Interest packets to advertise the hash of its local catalog manifest when it generate new data. The advertisement Interests are also named under the data collection prefix and forwarded to all sync nodes announcing that prefix. They have short lifetime and do not retrieve any data. To increase the possibility that all nodes can receive the advertisement, the node repeats the advertisement Interest once or twice within a few seconds after the first advertisement is sent. Once a node receives a different hash, it should also advertise its own hash under the control of some gossip protocol (with random backoff and duplicate suppression). It then sends out Interests to retrieve the corresponding manifest packets (possibly segmented), compares the names listed in the manifest with its local namespace, and then retrieves the missing data over the network. This approach is similar to iSync but without the benefit of efficient encoding and differentiation provided by the IBF data structure.

D. ChronoSync

ChronoSync [6] attempts to improve efficiency of dataset synchronization by utilizing naming conventions. In particular, each ChronoSync node publishes data that contain application-layer messages under its own unique name prefix. This prefix also serves as an identifier for the node in the sync group and is aligned with the topological prefix of the access network for each node. The name of the Sync Data is constructed by concatenating the node prefix with a sequence number that starts from zero and gets incremented by one for each new data published by the sync node. Although in theory the sequence number could wrap around if represented by fixed-size integers, in practice it is usually not a big issue: for a sync node publishing data at the rate of 1000 packets per second, it takes more than half a billion years for a 64-bit sequence number to wrap around; even with 32-bit integer representation it still takes about 50 days for the sequence number to wrap around, which provides enough time for the sync group to garbage-collect the previous data.

The sync node maintains a 2-level "flat" sync tree, as is shown in Fig. 4, with each leaf containing the data prefix and the latest sequence number of each member in the sync group. Each leaf is associated with the digest calculated over node's prefix and the latest sequence numbers. The root of the tree maintains the digest of concatenation of leaf digests canonically ordered by the corresponding prefix names. Since the naming convention is to publish Sync Data with continuously increasing sequence numbers (starting from zero), this sync tree is essentially a condensed representation of the namespace containing all Sync Data ever published in the group, and root digest is a short summary of the dataset.

ChronoSync nodes periodically send out *Sync Interests* containing the root digest of the local sync tree to all members

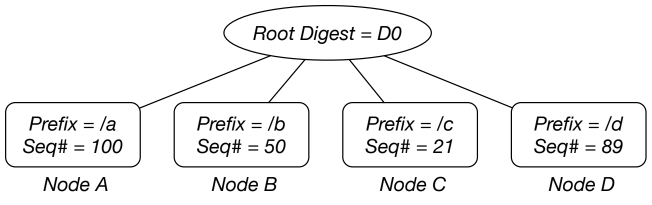

Fig. 4: Example of a sync tree in ChronoSync

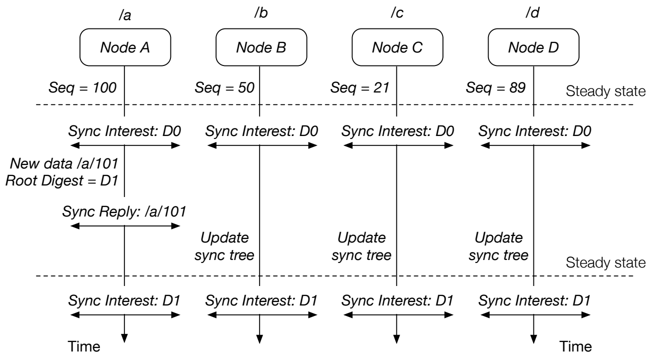

Fig. 5: Synchronization process in ChronoSync

in the sync group. When some node publishes new data and increments its sequence number, instead of replying to the Sync Interest with its new root digest as in CCNx Sync, the node replies with the name of its newly published data (i.e., the node prefix and the sequence number).[5] This *Sync Reply* is efficiently delivered to all the other nodes in the group, following the multicast tree built by the previous Sync Interest. After they receive the reply, the sync nodes update their local sync tree, recompute the root digest, and then send out Sync Interest that carries the new digest. An example of the synchronization process in ChronoSync is shown in Fig. 5.

To allow efficient state reconciliation, each ChronoSync node maintains a limited log of historical digests and the corresponding dataset states. If some node is lagging behind in the synchronization process and sends out a Sync Interest with a digest that has been observed by other nodes, these sync nodes can respond with all the data published in the group since that digest is announced. Note that when multiple sync nodes reply to the Sync Interest carrying a previous digest (potentially with different sets of updates if they are not synchronized), at most one of those relies will be received by the sender of that Interest. Nevertheless, the reply helps speed up the synchronization process of the Interest sender who is trying to catch up with the rest of the group.

There are several cases where a node may receive Sync Interests with unrecognized digests. In the first case, a node may receive a Sync Interest with an updated digest before receiving the Sync Reply that triggered the update. To handle that situation, ChronoSync injects a random delay to process the Sync Interest with unknown digest at a later time, expecting to receive the corresponding Sync Reply while waiting.

[5]If multiple data packets are generated, the Sync Reply carries only the largest sequence number of all new data.

In the second case, multiple Sync Replies can be generated in response to the same Sync Interest, if multiple nodes publish new data at the same time. However, because of NDN's flow balance property, nodes will receive no more than one reply to the Sync Interest. As a result, nodes may receive different data items, compute multiple different state digests, and start announcing them in the sync group.

The third and a more complicated case arises if the network is partitioned for a long period of time and then reconnected. The sync nodes in different partitions have cumulated multiple updates to the sync tree, leading to a sequence of digests that are unrecognizable to the nodes in other partitions.

ChronoSync can handle simple cases when the nodes diverge by at most one Sync Reply by resending the previous Sync Interest with exclude filters that contain the implicit digests of the received Sync Replies. However, if multiple changes have been applied to the sync state at some node, the mechanism using exclude filters will not be able to retrieve the diverging sync replies generated by every node (see II-E for detail). In such cases, ChronoSync falls back to a *recovery* mechanism: when a node observes an unknown digest, it will send out a special *Recovery Interest* containing the unknown digest; the nodes who recognize that digest will reply with the complete information about its sync tree, rather than the specific changes that lead to that digest; when the requesting node gets the reply, it will merge the received sync tree into its local sync tree by taking the higher sequence number from both trees for each sync node.

To support more complex naming conventions with richer embedded semantics (e.g., for trust management), application can use ChronoSync together with "one level of indirection." In other words, application-defined data names (and the data itself if the size is small) can be encapsulated in the data packets managed by the sync layer. This can be realized either directly or with the help of sync-managed *actions* that describe objects added or removed from the application data collection, such as updated files a distributed file system or messages in a chat room.

E. RoundSync

The recognition of the ChronoSync problem in scenarios with many simultaneous data generation led to development of the RoundSync [7] protocol. Specifically, one of the causes of the problem is the overloaded function of Sync Interests: (1) to detect different states among the sync nodes and (2) to retrieve the updates from other nodes. As a result, the Sync Replies carrying the updates to the shared dataset will be named after the previous Sync Interest name which contains the digest of the corresponding sync state. If a node generates Sync Replies on top of a diverged state (e.g., in the scenario with partitioned sync group), nodes with different states will not be able to derive the name for those Sync Replies and therefore cannot send Interests to retrieve them. Merging the diverged sync states will only create new set of sync states, potentially contributing in further divergence of the states. To re-synchronize in this case, ChronoSync must rely on a recovery mechanism to receive the entire sync state.

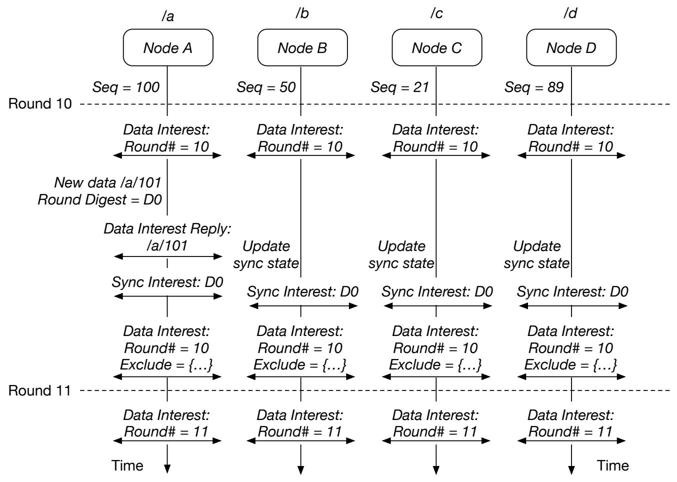

Fig. 6: Synchronization process in RoundSync

To address this problem, RoundSync divides the synchronization process into *rounds*, updates semantics of the Sync Interest, and introduces the new type of Interest packet called *Data Interest*. The RoundSync's Sync Interest, augmented with the round number information, serves only as a notification mechanism (similar to iSync) to inform other sync nodes about the state in the round. When the divergence is detected, the nodes can request the data in the round using the predictably named Data Interests, i.e., names of Data Interests do not include state digest but only the round number. Therefore, published data within a specific round can be retrieved even if the states are not fully synchronized. The replies to the Data Interest have the same functionality as the Sync Reply in the original ChronoSync design, i.e., they carry the node's prefix and sequence number of the new Sync Data. In addition, RoundSync mandates that a sync node can publish at most one data packet in each round and must move to a new round when it receives new data published by others in the current round. This helps reduce the chances of state divergence caused by simultaneous data production.

For example in Fig. 6, a sync node may start publishing data at round 11 even though it is still trying to synchronize with other nodes at round 10 or earlier. If multiple nodes publish data in the same round simultaneously, they will detect the inconsistency through Sync Interest and then send Data Interests with exclude filters to retrieve those Data Interest replies. Since there will be at most one reply from each node in a single round, the exclude filter mechanism will allow the nodes to eventually retrieve all updates.

RoundSync maintains digest for each round in a *rounds log* table. To allow nodes who missed the Sync Interests in earlier rounds to detect and recover the missing data, RoundSync also computes a *cumulative digests* that covers the entire dataset as observed in a round and is piggybacked in the Data Interest replies of future rounds. Upon receiving a different cumulative digest for some round that is long before the node's current round, the sync node sends out a Recovery Interest to fetch the full sync state and the current round number S from the node who generated that cumulative digest, instead of retrieving missing data round-by-round (which may take a long time). After receiving the reply, the node merges the received dataset with its own, discards the rounds log entries in the rounds before S and resume normal RoundSync operation for the rounds after S.

F. PSync

PSync [8] (a.k.a. PartialSync) was originally designed for the consumers to synchronize a subset of a large data collection with a single producer. The data published by the producer are organized into *data streams* which are identified by the unique stream prefixes. Like in iSync, PSync also employs IBF to represent the namespace by storing the hash of the names (called *KeyID*) in the fixed-length slot of the IBF. However, PSync also adopts the naming convention in ChronoSync and RoundSync that data packets from the same stream are ordered by the continuous sequence numbers. Therefore the IBF only needs to store the latest data name from each stream. This further reduces the amount of information stored by the IBF and allows the applications to choose a smaller IBF size that can be transmitted more efficiently over the network.

To support the synchronization of a subset of the producer's data (a.k.a., *partial sync*), PSync introduces the subscription list to encode the prefixes of the data streams that the consumer is interested in.[6] The subscription list is a Bloom Filter (BF) that stores the hashes of those stream prefixes. The size of the Bloom Filter is determined by the total number of streams a consumer may subscribe and the false positive rate the consumer is willing to accept. Special cases like empty and full subscription may be encoded more efficiently with special markers.

During the sync process, the consumer keeps a local copy of the producer's IBF which indicates the data it has received so far. To sync up with the producer and retrieve new data, the consumer sends a *Sync Interest* whose name contains the local IBF copy and the consumer's subscription list. When the producer receives the Sync Interest, it first subtracts the IBF in the Sync Interest from its current IBF, and extracts the KeyIDs of the new data packets that have not been received by the consumer yet. Then the producer checks whether the stream prefixes of those new data packets are included in the consumer's subscription list (subject to certain false positive rate). Finally the producer generates a sync reply containing the original names of the new data packets in the subscribed streams and also its latest IBF. Upon receiving the sync reply, the consumer updates its local IBF copy with the received IBF, and sends out Interests to fetch the new data. An example of the synchronization process in PSync is shown in Fig. 7.

An important feature in the PSync design is that each consumer maintains its own data consumption and subscription status. The producer, on the other hand, does not maintain per-consumer state, which significantly reduces the amount of data stored by the producer. However, this *stateless* producer design introduces two additional costs: first, the Sync Interest and Sync Reply need to carry the IBF and the subscription list (BF)

[6]PSync allows the consumers to specify their subscription only at the granularity of data streams.

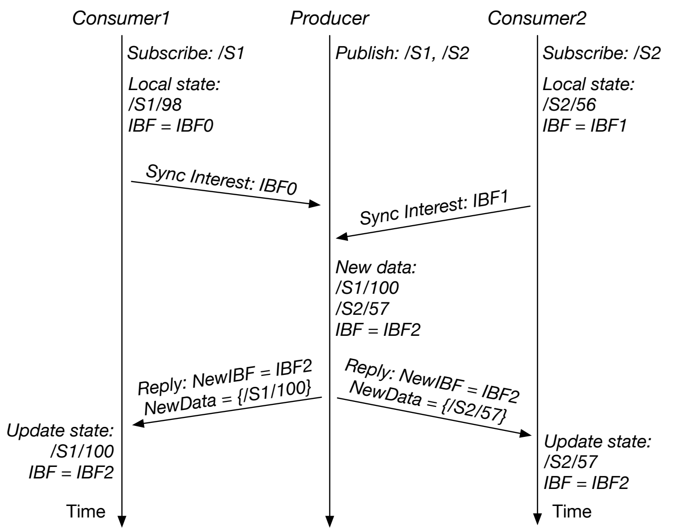

Fig. 7: Synchronization process in PSync

which will bloat the size of the Interest name up to hundreds of bytes; second, the producer needs to generate Sync Replies in real-time for each Sync Interest since it does not remember the previous consumption status of each consumer and cannot pre-generate the next Sync Reply.

Although it was initially designed for producer-consumer synchronization, PSync can support multi-producer distributed dataset synchronization (like other sync protocols discussed in this paper) where each sync node is both producer and consumer at the same time. This is achieved by having every sync node subscribe to all data streams published by every other node. However, in this "full synchronization" mode each node only needs to maintain a single IBF which represents the state of the whole dataset, rather than keeping a separate IBF for each node in the group. In addition, the Sync Interests need to be forwarded via *multicast* to the entire group so that any node who has produced new data can respond with a reply that carries the updates.

G. Summary

Table I summarizes the design choices made by different sync protocols. As we can see from the side-by-side comparison, a few common design patterns have arisen in each of the key design aspects among the existing sync protocols. ChronoSync, RoundSync, and PSync have adopted similar data naming conventions, i.e., naming the data packets using sequence numbers under a common name prefix for each producer or data stream, to simplify the representation of the shared dataset namespace. Having continuous and monotonically increasing sequence numbers in the data name allows the cumulative data collection generated by the same producer or in the same data stream to be summarized by the highest sequence number. This helps reduce the amount of information that needs to be encoded in the sync state representation.

The existing sync protocols have used a variety of data structures to represent the sync state. All of those data structures provide lossless encoding of the data names (or the hashes of the names) in the shared dataset. Both CCNx Sync and CCNx 1.0 Sync enumerate the dataset namespace in the hierarchical sync tree and the manifest, respectively. To reconcile the set difference, the sync nodes simply compare the content in the sync tree and the manifest and then retrieve the missing data from the remote nodes. ChronoSync and RoundSync also enumerate the dataset namespace by listing the latest sequence numbers from all data producers in the sync state. State reconciliation is achieved by comparing the sequence numbers from the local and remote sync state for each producer and taking the maximum as the latest sequence number. iSync and PSync use IBF to compress the dataset namespace and perform set reconciliation efficiently using IBF subtractions. However, due to the limitation of IBF capacity, both iSync and PSync have to provide means for controlling the size of the set difference between the IBFs maintained by different nodes.

The surveyed sync protocols use one of the two communication models for propagating the new data published in the sync group. The first model is to use multicast Interests to carry the summary of the sync state changes (e.g., digest of the updated sync state), which serves as a notification to prompt other nodes in the sync group to retrieve detailed information about the changes. The second model is to have the sync nodes send "long-lived" Interests to each other (typically using multicast) to pre-establish the return path for the data packet that carries the complete information about the sync state changes. The "long-lived" Interests essentially become a "one-packet" subscription to the sync state update information generated in the future.

It is difficult to compare the efficiency of different sync protocols because it is highly dependent on the application scenarios and the implementation choices. A typical metrics is the synchronization delay, i.e., the number of round-trips necessary for bringing the group in sync after an update happens, which is shown in the last row of Table I. In iSync, PSync, and CCNx 1.0 Sync, the synchronization delay is guaranteed to finish within certain number of RTTs. For CCNx Sync, the number of round-trips required to retrieve all updates from a remote node depends on the depth of the sync tree. ChronoSync and RoundSync achieves optimal synchronization delay when there is no simultaneous data publishing. If multiple nodes generate sync replies at the same time, the protocols need additional round-trips to retrieve all sync replies using Interests with exclude filters. Therefore, the worst-case RTT will be proportional to the number of simultaneous updates in the group, which is bounded by the number of data publishing nodes in the group.

Another metrics is the packet size of the sync protocol messages, which reflects the network bandwidth requirement of the sync communication. Here we mainly focus on the encoding size of the sync state (or state updates) carried in the Interest and/or Data packets. For iSync and PSync, the size of the IBF that summarizes the dataset namespace depends on the size of the hash function output and the data publishing rate of the applications. In a typical implementation that uses 64-bit hash functions and 32-bit counter values, the size of each slot in the IBF is 20 bytes. For an IBF with capacity of 20 items (i.e., allowing at most 20 items to be extracted successfully), the encoded size of the IBF [10] is around $1.5 \times 20 \times 20 = 600$ bytes.[7] In contrast, ChronoSync and RoundSync require only the updates to be propagated in reply to the Sync Interests and Data Interests, respectively. Those update data packets contain the prefix and the latest sequence number from each producer who has published new data. Assuming the average size of the data name (i.e., prefix + sequence number) is 40 bytes, the maximum content size of the update is $40 \times N$, where N is the number of producers in the sync group. In practice, not all producers will be publishing at the same time and the size of the update packets are typically smaller than in the IBF-based approaches. In CCNx Sync, the size of the NodeFetch reply packets is proportional to the number of children under the request node in the sync tree; also, the protocol requires multiple NodeFetch packets to resolve all the differences. In CCNx 1.0 Sync, the size of the manifest is proportional to the number of the included data names, representing either a complete shared dataset or serving as (hierarchical) links to manifests (or manifests of manifests) of sub-datasets.

III. Open Issues

In this section we discuss open research issues in distributed data synchronization in NDN that have not been addressed by the existing sync protocols. By inspecting the range of design choices in those open areas, we hope to shed light on the directions for future work.

A. Interest Multicast

The group communication model in the sync protocols has created several challenges for the routing scalability in the NDN network. First of all, the sync protocol relies on Interest multicast to deliver the state change notification to every node in the group. Supporting Interest multicast via routing would require the per-application multicast prefixes be announced by all networks where the sync nodes reside, which is usually not feasible for large networks hosting many sync groups. One solution to the scalable multicast problem is to establish some communication topology in the sync group, e.g., using *Distributed Hash Table* (DHT) [11], so that the Sync Interests can be propagated via that topology. Another solution currently under our investigation is to utilize a *multicast overlay* that contains a number of dedicated rendezvous points in the network. Those rendezvous points are responsible for collecting and delivering the Sync Interests via the overlay to every sync node.

B. Data management

A sync node usually stores two types of information locally: the data generated by the applications, and the internal state (or metadata) created by the protocol itself. There are several practical issues related to the management of the dataset and protocol state that may have significant impact on the operations of the applications running on top of sync.

1) Data sharing: To improve the data availability, every sync node should be able to serve the entire dataset, including the data published by other nodes, to the consumers. For example, in a group chat application like ChronoChat [12], a user who join the chat room late may want to retrieve some of the earlier chat messages published by other users, even if those users have long left the chat room. In that scenario it is beneficial if the current users in the chat room could store and serve the data published by the past users.

The challenge here is that the application data generated by different peers can be named under the unicast prefixes of different nodes.[8] To fetch the shared data from non-authoritative peers, a sync node can utilize the *Forwarding Hint* [13] mechanism. When a node wants to retrieve some historical data published by a node that is no longer reachable, it may issue Node Data Interests with the forwarding hint field carrying one or more unicast network prefixes of the current group members. The NDN forwarders will use the forwarding hint to direct the Interest toward one of those group members.

2) Data archiving: Long-running applications often accumulate a large amount of data. When a new sync node joins the application, it may take a long time to bootstrap the node by fetching the data objects in the shared dataset piece-by-piece. To improve the efficiency of application data transfer, it may be needed to consolidate the data into a single archive file. For example, if the application publishes “actions” through the sync protocol to modify its data (e.g., editing files in a shared folder), the archive file may contain only the latest version of the application data that reflects all the changes made by the users. This mechanism shares a similar spirit with the compaction process in Log-Structured Merge (LSM) databases [14] that coalesces multiple disjoint DB tables into a larger sorted table in order to speed up the lookup operations.

The archive file could be published under the application namespace and segmented into multiple large data packets that can fit into the Maximum Transmission Unit (MTU) of the underlying network. If necessary, the archive data may be stored in dedicated repos without incurring additional storage cost for the sync nodes. Consumers who are interested in the historical data may send pipelined Interests (with the Forwarding Hint as described earlier) to fetch the entire archive file from the sync nodes or the application repos.

3) Data sharding: As the application continues to generate new data, the size of the whole dataset may exceed the storage size of each individual node. If the application requires permanent storage of all data ever published by the users, the shared dataset needs to be *sharded* across the sync nodes for storage scalability. One way to achieve that is to build a data sharding service on top of the sync protocol using consistent hashing [15] or Distributed Hash Table (DHT) techniques to divide the data namespace among the sync nodes. When a sync node receives notification of a new data packet, it consults the data sharding service to decide whether it is responsible for storing that data. If the data falls into its local shard, the node will subsequently retrieve the data and store a local

[7] Note that once the size of the IBF is chosen, all Interest and Data packets carrying the IBF will have the same size even if the number of updates is lower than the maximum capacity.

[8] Having all sync nodes publish data in a shared application namespace (identified by a common multicast application prefix) would require support for Interest anycast in the network.

copy; otherwise, it simply updates the local sync state without fetching the data. Note that the sync nodes are still able to sync up with each other even if each of them maintains only a subset of the shared data because the sync state is solely based on the namespace of the dataset.[9]

4) Garbage collection: Some sync protocols generate internal state that grows infinitely as the application continues to operate (e.g., the local and global IBFs in iSync and the round digest log in RoundSync). Since the storage of each sync node is finite, the sync protocols need to provide a garbage collection mechanism to clean up the protocol state. For example, in ChronoSync a dead node may be removed from the sync state if every other node has received all data published by the dead node. Depending on the application semantics, the historical data in the shared dataset may also be garbage-collected to avoid overwhelming the storage. The sync protocol can generate periodic "checkpoints" that inform every node in the group to delete the data generated before the checkpoints. Note that supporting *safe* garbage collection requires collecting the data consumption state from all the participants in the distributed application, which further requires a group membership management mechanism (discussed in the next subsection).

C. Group membership management

Having a consistent view of the group membership among the nodes in the distributed system is a prerequisite for a lot of useful functions such as system snapshot, garbage collection, access control, and strong consistency guarantee. For example, to generate a group-wide snapshot of the published data, the sync protocol needs to collect the latest data publishing state from every node in the group; to safely garbage-collect some historical data from the shared dataset, the sync protocol needs to make sure that all nodes in the group have received that data.

Existing sync protocols do not manage the membership information for the sync group. CCNx Sync and iSync do not have the concept of "sync group" at all: the sync state (sync tree and IBF, respectively) does not reflect the identities of the repos that are maintaining the shared dataset. ChronoSync maintains the list of current participants in the sync group via the sync tree, but does not require the sync nodes to have a consistent view of the group while synchronizing with each other. If a sync node receives a new data object produced by a node that is not in its current sync tree, the receiving node will extend its sync tree by adding the producer node into it. Consequently, existing NDN applications running on top of the sync protocols (e.g., ChronoChat) currently have to implement custom group membership management solutions at the application layer.

D. Consistency and data ordering

Consistency in distributed systems has been extensively studied for decades. Strong consistency models such as *linearizability* [16] enforce a global total ordering of events observed by every node in the system. Weaker consistency models relax on the ordering requirements in different ways. In particular, a system with *eventual consistency* is allowed to diverge and expose inconsistent states during the system execution, as long as it eventually resolves the inconsistency.

The main benefit of strong consistency models is that they can simplify the application logic since the applications do not need to handle inconsistent states in the distributed system. That simplicity comes at the cost of longer delay and higher coordination overhead: to implement strong consistency, distributed systems typically use 2-phase commit or consensus protocols like Paxos [17], [18] to replicate the operations across different replicas and use 2-phase locking to ensuring correct ordering among the operations. Moreover, strongly consistent systems often suffer from availability problems in the presence of network and/or node failures. The famous CAP theorem [19] states that distributed systems can simultaneously achieve at most two out of the three properties: consistency (C), availability (A), and partition tolerance (P). Weaker consistency models are able to achieve higher performance and better tolerance to failures, but leave the burden of consistency checking and conflict resolution up to the application protocols.

We must keep in mind two important factors in all consistency discussions: a) the definition of consistency varies among different applications, and b) there exists a tradeoff between consistency and availability. Existing NDN sync protocols favor availability over consistency: they allow sync nodes to publish new data at any time and propagate the changes to other nodes asynchronously. In the absence of new data generation and permanent network failure, all sync nodes will eventually receive all data packets in the shared dataset. This model is sufficient for replicating *unordered* data collections (e.g., the repos running CCNx Sync). Applications with strong consistency requirements can always establish a consistent ordering on the data generated in the sync group.

IV. Conclusion

This paper presents an overview of the distributed dataset synchronization problem in NDN and a survey on the existing sync protocols. By summarizing and comparing their protocol design, we articulate the different design choices made in the existing sync protocols together with their advantages and limitations. We also discusses open issues that have not been fully addressed in the previous works. By writing this survey paper, we hope that new sync protocols developed in the future can benefit from the past experience and address the open problems with innovative ideas.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *Proceedings of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009, pp. 1–12.

[2] 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.

[9]Note that this requires the implementation of the sync protocol to decouple namespace synchronization from data fetching.

[3] CCNx Project, "CCNx Synchronization Protocol." [Online]. Available: https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt

[4] W. Fu, H. Ben Abraham, and P. Crowley, "Synchronizing Namespaces with Invertible Bloom Filters," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2015, pp. 123–134.

[5] M. Mosko, "CCNx 1.0 Collection Synchronization," Apr 2014. [Online]. Available: http://www.ccnx.org/pubs/hhg/4.7%20CCNx%201.0%20Collection%20Synchronization.pdf

[6] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*, Oct 2013, pp. 1–10.

[7] P. de-las Heras-Quirós, E. M. Castro, W. Shang, Y. Yu, S. Mastorakis, A. Afanasyev, and L. Zhang, "The Design of RoundSync Protocol," NDN Project, Technical Report NDN-0048, April 2017.

[8] M. Zhang, V. Lehman, and L. Wang, "Scalable Name-based Data Synchronization for Named Data Networking," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, May 2017.

[9] D. Kulinski and J. Burke, "NDN Video: Live and Prerecorded Streaming over NDN," NDN Project, Technical Report NDN-0007, September 2012.

[10] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the Difference?: Efficient Set Reconciliation Without Prior Context," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, pp. 218–229.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the 2001 SIGCOMM Conference*, 2001, pp. 149–160.

[12] Z. Zhu, C. Bian, A. Afanasyev, V. Jacobson, and L. Zhang, "Chronos: Serverless Multi-User Chat Over NDN," NDN Project, Technical Report NDN-0008, October 2012.

[13] A. Afanasyev, C. Yi, L. Wang, B. Zhang, and L. Zhang, "SNAMP: Secure Namespace Mapping to Scale NDN Forwarding," in *Proceedings of 18th IEEE Global Internet Symposium (GI 2015)*, April 2015.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205–218.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 654–663.

[16] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[17] L. Lamport, "The Part-time Parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, May 1998.

[18] ——, "Paxos Made Simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[19] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)," in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00, New York, NY, USA, 2000, p. 7.