A Survey of Distributed Dataset Synchronization in Named Data Networking

Philipp Moll*, Wentao Shang*, Yingdi Yu*, Alexander Afanasyev†, and Lixia Zhang*
*UCLA – {phmoll,wentao,yingdi,lixia}@cs.ucla.edu
†Florida International University – aa@cs.fiu.edu

***Abstract*—Distributed data set synchronization protocols (sync protocols for brevity) provide an abstraction for connection-agnostic multiparty communication in NDN. A number of sync protocols have been proposed over the years, each featuring different design choices in data naming, dataset state representation, and state propagation mechanisms, which led to different design tradeoffs. In this report, we survey all the NDN sync protocols that have been developed, highlighting their commonalities as well as fundamental differences through detailed analysis and side-by-side comparisons. We also articulate the remaining issues to be addressed to make sync protocols available to a broad range of applications.**

Revision History

Revision 1 (May 2017): The initial version of the survey discusses six sync protocols that had been developed by the time: CCNx 0.8 Sync, CCNx 1.0 Sync, ChronoSync, RoundSync, iSync, and PSync.
Revision 2 (May 2021): This second vision adds the description and discussion of six new sync protocols which were developed after 2017: VectorSync, StateVectorSync, ICT-Sync, PLI-Sync, syncps, and QuadTreeSync. In addition, some parts of the initial report are also restructured and revised.

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 a simple communication primitive that lets a data consumer to send an Interest packet with a name or name prefix to retrieve a Data packet which is named under that prefix and can be verified. While the Interest-Data exchange primitive has significantly narrowed the semantic gap between the application layer and the network layer in today's TCP/IP network architecture, it is cumbersome to use when building distributed applications that often 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. With the TCP/IP architecture, whenever communication involves more than two parties, the applications have to either establish multiple TCP connections between the peers or rely on (at least logically) centralized infrastructure to support multiparty communication.

The data-centric nature of the NDN architecture provides a foundation for distributed dataset synchronization protocols (Sync protocols, or Sync, for brevity) as an important layer of abstraction for multiparty communication on top of the network layer Interest-Data exchange primitives. Distributed applications and services rely on sync protocols to keep each other informed about updates in the dataset and to learn about newly published data. This use of NDN Sync to provide reliable data-centric communication differs from data retrieval via a TCP connection in three fundamental ways. First, sync naturally 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 interconnected 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.

NDN can achieve distributed dataset synchronization by synchronizing the *namespace* of the shared dataset among a group of distributed entities (called *Sync entities*). To share a new data item, a producing (side of) applications injects its name into the dataset. After learning the new name, 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 a *transport-layer* protocol 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.

Different sync protocols have been developed over the years, each having distinct features and differentiating from protocols before. Observing the development of existing Sync protocols allows one to gain insights into the design decisions and their rationals. Different terminology, scarce documentation, and the long history make parsing these lessons from existing work cumbersome for researchers new to NDN. This report surveys existing developments and aims to shed light on the fundamental differences of available protocols. We introduce the existing sync protocols and identify the three most important questions in sync protocol design.

The remainder of the report is organized as follows. We provide a high-level overview of sync protocols, their components, and requirements in Section II. In Section III, we introduce existing sync protocols and critically reflect their design decisions. Section IV summarizes the commonalities and differences of the presented protocols and provides a qualitative comparison on performance metrics. Section V discusses ongoing challenges, and Section VI concludes the

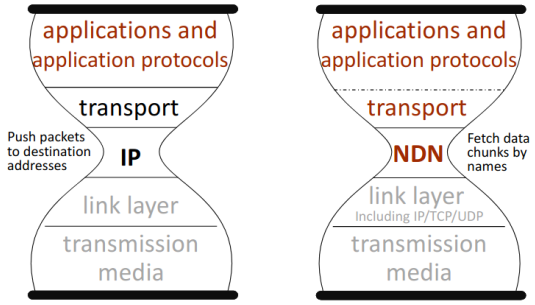

Fig. 1: Comparing the hourglass models of the IP and NDN architecture (Figure adopted from [3]).

report.

II. A High-Level Picture of Sync Protocols

Looking at network architectures, the lowest layer is the communication media that transports bits, and the highest layer, applications that exchange information. As visualized in Figure 1, this basic consideration is the same for the IP and NDN architecture. Differences, however, can be found in the narrow waist, the universal layer that is (indirectly) used by all applications and represents an abstraction for lower network layers. While IP forwards datagrams to destinations, NDN fetches named, secured data chunks. Although the two network services are fundamentally different, in both cases, application developers do not want to directly utilize the services provided by the network layer. Therefore we need transport protocols to bridge the gap between the services desired by applications and that provided by the network layer.

Engineers building applications on top of the matured IP architecture can choose among various transport layer protocols, such as UDP for unreliable packet delivery, or TCP and QUIC for reliable communication. The range of available possibilities, however, does not end at the transport layer. A broad variety of middleware (e.g. ZeroMQ[1], gRPC[2], MQTT[3]) allows choosing between higher-level communication concepts that are built on top of transport layer protocols and patch application semantics into IP datagrams, such as message topics for publish/subscribe.

Looking at the more recent NDN protocol stack still reveals a gap between the narrow waist and applications that want to use transport protocols. This is because NDN transport reaches beyond connecting endpoints and aims to directly offer higher-level communication concepts that use application semantics. While middleware in IP needs to augment datagrams with semantic information, NDN's named content chunks inherently provide such information. This brings the applications' need one step closer to the actual network and allows using functionality provided by the network, such as data-centric security, network-level multicast, and in-network caching.

One of such transport layer protocols in NDN is distributed dataset synchronization (sync for short). Sync protocols enable namespace synchronization in group communication setups. This namespace synchronization allows implementing reliable multi-producer/multi-consumer communication over NDN. Therefore, producers add the name of the data to distribute into a distributed dataset, which is synchronized among potential consumers. In this section, we examine the tasks necessary to build such a communication layer and provide the basic design of a sync protocol.

Essential concepts for sync protocols to work are a) the application namespace that is synchronized, referred to as *data naming*, b) the way the dataset's state is represented by the protocol, referred to as *dataset state representation*, and c) the sequence of network messages required to synchronize the dataset, referred to as *state sync mechanism*. In the following, the role of these concepts for sync is discussed and used later for comparing existing sync protocols.

*a) **Data naming**:* The unique binding between names and immutable data objects in NDN allows to uniquely identify a shared dataset by knowing the hierarchical names of all data packets in the dataset. Therefore, dataset synchronization in NDN is reduced to the synchronization of the corresponding namespace. Knowing the available data names allows retrieving data objects by Interest-Data exchange. Sync protocols may directly synchronize arbitrary data names, or leverage a sequential naming convention to simplify the dataset namespace.

*b) **dataset state representation**:* The data structure that internally represents the state of the shared dataset is essential for the protocol function and brings certain properties that can be used by the actual sync mechanism (eg. tree traversal with tree-based structures). Every sync participant keeps a local copy of the dataset state and uses the sync protocol to keep up with the changes generated by other participants in the sync group. This requires to encode the dataset without loss of information and allows sync participants to detect and reconcile the differences in the shared namespace between distinct states. Examples for such encodings found in existing sync protocols are tree-based structures, Invertible Bloom Filters (IBF), and state vectors.

*c) **State sync mechanism**:* Each node participating in a sync group may publish new data to the shared dataset at any time. The sync protocol ensures that the other nodes in the group receive the new data and reach an agreement on the state of the dataset. Existing sync protocols show similarities in the protocol function which allows sketching a basic sync protocol design, as visualized in Figure 2. In the basic design, a sync participant (Alice) publishing new data notifies the other participants about the existence of the new data (step 1). This is done by either sending a Sync Interest to the other members of the sync group, or by answering outstanding Sync Interests. Depending on the protocol, this first message carries exact information about the changed publication, or only a notification about a change, without further specifying the changed publication and eventually requiring additional communication for state synchronization. After having received the latest name and thereby synchronized the namespace, the actual publication can be retrieved by Interest-Data exchange (step 2). Besides, sync protocols may utilize periodic heartbeat

[1]https://zeromq.org/, last accessed: 2021-02-19
[2]https://grpc.io/, last accessed: 2021-02-19
[3]https://mqtt.org/, last accessed: 2021-05-20

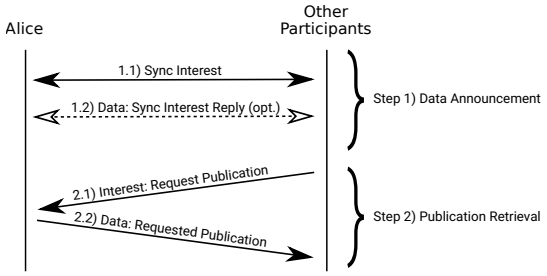

Fig. 2: Basic Sync Protocol Design

messages that can either be used to detect changes in the synchronized namespace, or to provide information about group members.

Figure 2 only shows a basic design, the exact messages sent by real sync protocols may differ. In the following, general requirements, as well as efficiency and security considerations applying to sync protocols are discussed. The commonalities of the basic design to existing sync protocols allow for transferring insights and discuss implications of design decisions later on.

A. Group Communication Requirement

One prerequisite that applies for most sync protocols is a group communication capability of the underlying network. To support dataset synchronization inside a group, sync protocols use a group communication namespace for broadcasting Sync Interests to all participants of the sync group. To achieve group communication, the protocol may rely on multicast capability, or explore other group rendezvous mechanisms. The design of the group communication mechanism is outside the scope of the sync protocol and not in the focus of this report. Further discussion on group communication considerations is provided in Section V-A.

B. Interest Aggregation and Security Implications

In NDN, Interests with the same name are aggregated on forwarding nodes in the network. This feature results in reduced traffic, and a reduced number of Interests to process on the receiving side. This observation indicates that one focus of protocol design should concern Interest aggregation.

In the basic sync protocol design, two types of Interests exist: i) Sync Interests that notify other participants about new Data, and ii) Interests for publication retrieval. Interests for publication retrieval carry the same name for all participants and, hence, are aggregated without further actions. For Sync Interests, aggregation becomes more tricky. Interest aggregation of Sync Interests requires that Interest names must not include identifiers of the sending participant. Besides, since the dataset's state is included in Interest names, these Interests are only aggregated when participants have the same knowledge of the dataset state.

Also, we want to discuss the relation of security and Interest aggregation. The receipt of Sync Interests may influence the dataset state. Without security considerations, this fact allows adversaries to easily manipulate the sync process by injecting Sync Interests holding incorrect dataset state. Overcoming this issue is possible by authenticating Interests. When Interest senders cryptographically sign the Interest, receivers can verify if the Interest was sent by an authorized sync participant, otherwise, the Interest is dropped. Thereby, adversaries are effectively prevented from injecting incorrect state.

Implementation-wise, signed Interests hold the cryptographic signature as an additional name component. With all participants using their secret key for signing, the signatures of individual participants, and thereby the final Interest names, differ for all participants. As a result, Interest aggregation stops working.

One way to allow for Interest aggregation with authentication in place are group key schemes. When all authorized sync participants use the same group key for Interest signing, signing results in the same Interest name for all participants – Interest aggregation works as expected. However, a concept for group key management is required, yet not the focus of this report. A concept likely to be usable for group key management was introduced by Zhu et al. [4] in the context of a secure audio conferencing tool.

C. Scalability vs. Overhead Trade-off

Sync Interests are used to synchronize the changes among all participants of a sync group. With a growing dataset size, the amount of data to synchronize increases. As discussed earlier, existing protocols can be classified into two classes, which traces back to a scalability vs. overhead trade-off. One class directly encodes the dataset state in Sync Interests or their response Data, accepting a larger packet size. The other class includes a digest in Sync Interests that allows inferring whether the dataset changed or not. When changes occur, the actual dataset state is communicated in follow-up communication.

While communicating changes directly in the Interest name reduces communication overhead and improves on dissemination delay, this approach reduces scalability. Unlike Data that can be segmented into multiple packets when exceeding the MTU, this is not possible for Interests. Hence, the dataset size is capped by the Interest capacity. Keeping in mind that authenticated Interests carry a cryptographic signature, the Interest capacity becomes even more constrained.

III. Existing Sync Protocols

In this section, we examine the set of existing sync protocols that have been developed for the NDN architecture. A coarse overview of existing protocols visualizing different branches of development is shown in Figure 3. Considering the historical development of sync protocols, early protocols provide valuable lessons learned used to optimize the design of state-of-the-art protocols. PSync, syncps, StateVectorSync, and ICT-Sync are four protocols that represent the latest state of protocol design. Although QuadTreeSync and PLI-Sync are later developments, they are developed for special use-cases and might not be seen as general-purpose protocols. Based on the basic sync protocol design presented in Section II, we

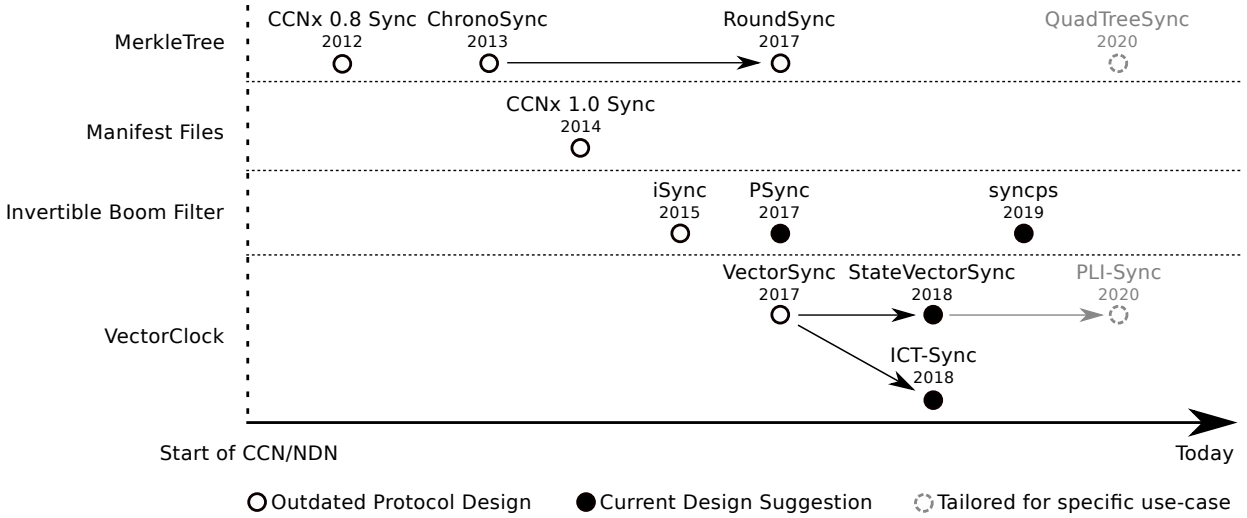

Fig. 3: The evolution of existing sync protocols. Arrows between two protocols denote the reuse of existing concepts. Included years indicate the time when the protocol was first published.

will investigate the function of existing protocols and identify different design choices and trade-offs.

Table I summarizes the commonalities and differences among existing sync protocols. In the rest of this section, we group protocols by their way of dataset representation and present their operation. For each protocol, we discuss the three design aspects described in Section II and critically reflect design decisions and trade-offs. Thereafter, Section IV provides a brief summary including a preliminary comparison of efficiency measures, such as synchronization delay and protocol message size.

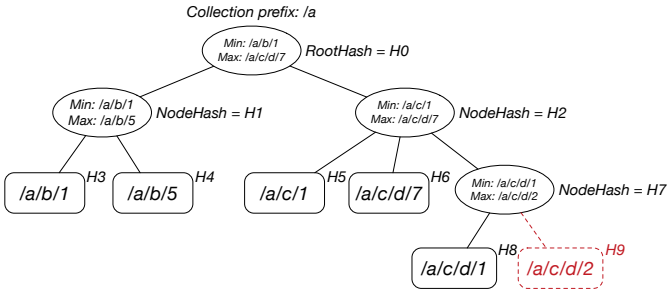

Fig. 4: Example of a sync tree in CCNx 0.8 Sync

A. Merkle Tree-based Sync Protocols

Naming schemes in NDN are often seen as trees, where the tree's root is represented by the leftmost name component, and with every name component, a new tree level is added. This hierarchical structure, often referred to as *Name Tree*, already suggests using a tree-based data structure for representation. A subset of sync protocols utilize tree-based structures and Merkle trees [5] for dataset state representation. A Merkle tree assigns every non-leaf node a cryptographic hash value representing all child nodes. When the value of a child node changes, all hash values on the path from the changed node to the tree's change. This reduces the task of comparing changes among multiple copies of a tree to comparing hash values of tree nodes.

1) CCNx 0.8 Sync: The CCNx 0.8 Sync protocol [6] is the earliest synchronization solution proposed for the NDN/CCN architecture. CCNx 0.8 Sync allows a set of repos to synchronize a shared data collection that contains data with arbitrary application names. The set of data names under a common *collection* prefix is organized into a tree structure called the *sync tree* (see Fig. 4). Sync tree nodes store a single data name (ie., a *leaf*) or summarize other nodes in lower tree levels. The structure of the sync tree is determined by the order in which the data names are added to the collection, which is independent from the canonical ordering of the data names.

Each node in the sync tree is associated with a hash value: the value of the leaf node is simply the hash of the name stored in that node; the value of the non-leaf node is recursively computed as the arithmetic sum of the hashes of all its children. In other words, the hash value of a node is the sum of the hashes of all data names contained in the sub-tree under that node. For example, in Fig. 4, $H_3 = Hash(/a/b/1)$, $H_2 = H_5 + H_6 + H_7$, and $H_0 = H_1 + H_2$. The root hash (H_0 in Fig. 4) then provides a summary of the entire namespace (i.e., sum of all data name hashes).

Any producer connected to a repo can publish new data into the data collection 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 hash values along the path from the new leaf node to the root. For example, in Fig. 4 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 node hashes H_7 and H_2, eventually propagating the change up to the root hash H_0.

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[4]. The RootAdvice Interest name starts

[4]The periodic RootAdvice is CCNx 0.8 Sync's equivalent to periodic Sync Interests of the basic sync protocol design discussed in Section II.

TABLE I: Comparison of existing sync protocols in NDN

	CCNx 0.8 Sync	**iSync**	**CCNx 1.0 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	Digest tree (List of {prefix : seq#})	Digest tree (List of {prefix : seq#}) + round log	IBF of hashes of names with highest seq#
State change detection	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	*Included in state change detection*	Data replying to *Data Interest* with updates in current round	*Included in state change detection*
	syncps	**VectorSync**	**SVS**	**PLI-Sync**	**ICT-Sync**	**Quadtree Sync**
Synchronized Namespace	Stream prefix + timestamp	Stream prefix + seq#	Sync prefix + node prefix + seq#	Sync prefix + node prefix + stream prefix + seq#	Stream prefix + seq#	Map chunk prefix + seq#
Sync state representation	IBF of name digests	State vector (List of {seq#})	State vector (List of {prefix : seq#})	State vector (List of {prefix : seq#})	State vector (List of {UID : seq#})	Quadtree with seq# for leaf nodes
State change detection	Sync Interest carrying IBF	Seq# in Sync Interests and state vector in Heartbeat Interest	State vector in Sync Interest	State vector in Sync Interest + opportunistic prefetching	Sync Interest and response Data	Digests replied to Sync Interest
State update retrieval	*Included in state change detection*	*Included in state change detection*	*Included in state change detection*	*Included in state change detection*	*Included in state change detection*	Map chunk Data replied to Sync Interests when traversing the tree

with a multicast prefix which is shared by all repos, followed by the current root hash of the sync tree. An incoming remote root hash that is different from its own indicates an update to the data collection. The repo who received the changed RootAdvice sends a *NodeFetch* Interest, named under the multicast prefix, and thereby retrieves the list of hashes for all the children under the root node of the sync tree. The NodeFetch process is recursively applied to all the nodes in the sync tree, skipping those with the same hash value between local and remote, 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 via Interest-Data exchange and insert the data in its local copy of the data collection. An example of the synchronization process in CCNx 0.8 Sync is illustrated in Fig. 5. Note that Fig. 5 shows the sync protocol between two repos for clarity. The RootAdvise and NodeFetch Interests are sent to the multicast prefix and received by all repos storing the data collection.

One issue in the update propagation mechanism of CCNx 0.8 Sync arises when multiple repos publish new data simultaneously. Simultaneous publications result in more than one reply to a RootAdvice Interest, where only one will be returned to the Interest issuer. In such a case, the sync agent who sends the initial RootAdvice Interest needs to issue additional Interests to fetch other replies.

A side-effect of the CCNx 0.8 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 any data once it is added to the data collection. 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 data collection maintained by CCNx 0.8 Sync is monotonically growing. This might create usability issues when applications generate a large amount of data and need to perform garbage collection periodically to reclaim storage. For example, the NDNVideo application [7] was deployed on top of the CCNx repo to publish live video streams. This system had to be cleaned up and restarted every day at midnight to avoid exceeding the storage capacity of the repo servers.

2) ChronoSync: ChronoSync [8] attempts to improve efficiency of dataset synchronization by utilizing naming conventions. In particular, each ChronoSync node publishes data 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 node's topological prefix of the access network. Data names are constructed by concatenating the node's prefix with a sequence number that starts from zero and gets incremented for each new data published by the sync node.

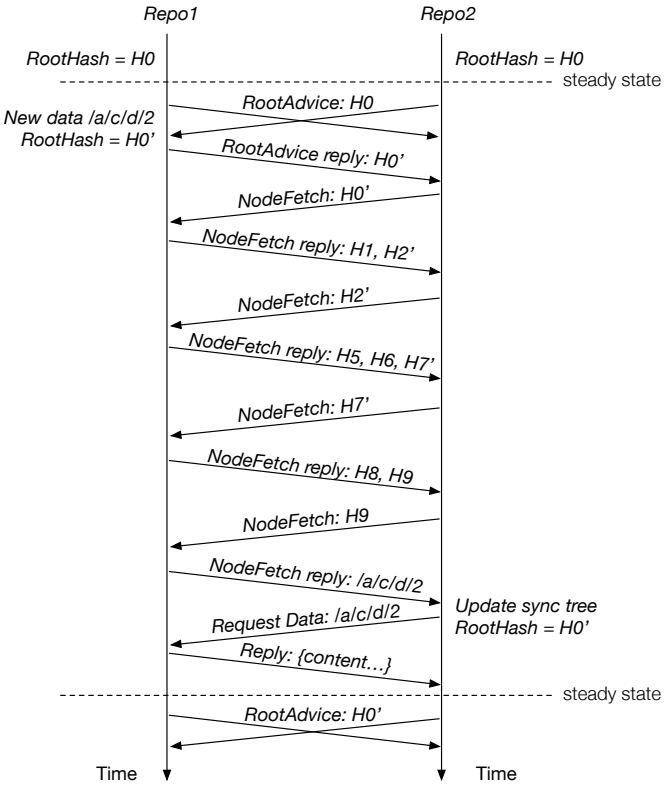

Fig. 5: Synchronization in CCNx 0.8 sync

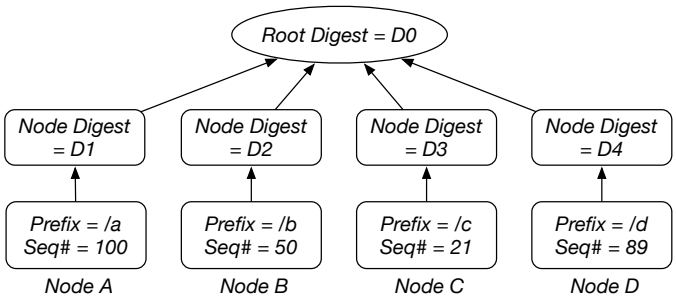

Fig. 6: Example of ChronoSync's sync tree

Sync nodes maintain a two-level sync tree, as shown in Fig. 6. Each leaf contains the data prefix and the latest sequence number of each producer in the sync group. Besides, leafs are associated with the digest calculated over node's prefix and the latest sequence number. The tree's root maintains a digest containing all leaf nodes in canonical order. Since the naming convention is to publish data with continuously increasing sequence numbers (starting from zero), this sync tree is a condensed representation of the namespace containing all Data published in the group.

ChronoSync nodes maintain *long-lived Sync Interests*[5] in the network. This is realized by emitting a new Sync Interest immediately after the previous one expires or is satisfied. The Sync Interest stays in the Pending Interest Table (PIT) of the forwarders in the network so that any reply to the Sync Interest can be returned to every node in the group as soon as it is generated. The Sync Interest name starts with the multicast group prefix and carries the current root digest of the sender's local sync tree. The Sync Interest serves two important purposes: first, it advertises the sender's digest in the group so that other nodes can detect inconsistency in the sync state; second, it requests the next state changes generated on top of the state identified by the carried digest.

[5]Long-lived Interests are Interests with lifetime set to a value close to the generation delay of the data they are requesting. The term is employed when that delay likely exceeds several RTTs – usually longer than a few seconds.

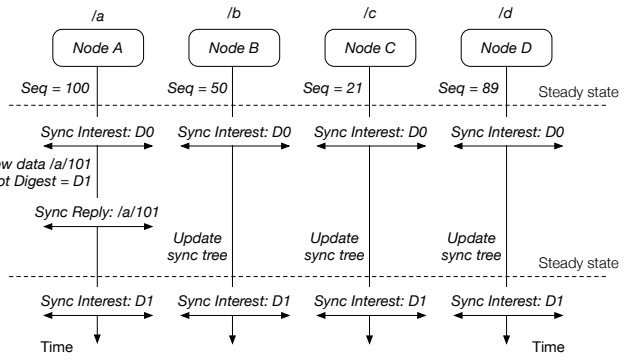

Fig. 7: Synchronization process in ChronoSync

When all nodes have up-to-date information about the dataset, all nodes generate identical state digests and send out the same periodic Sync Interests that are aggregated by the NDN forwarders. When a node publishes new data, its sequence number is incremented. The node replies to the outstanding long-lived Sync Interest with the name of its newly published data (i.e., the node prefix and the sequence number)[6]. This *Sync Reply* is delivered to all other nodes in the group, following the multicast tree built by the pending Sync Interests. After they receive the reply, the nodes update their local sync tree, recompute the root digest, and send out Sync Interests carrying the new digest. An example of the synchronization process in ChronoSync is shown in Fig. 7.

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 replies will be delivered to the nodes lagging behind.

In general, sync participants are supposed to recognize digests in received Sync Interests. However, certain situations lead to the receipt of unknown 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

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

time, expecting to receive the corresponding Sync Reply while waiting.

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 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[7]. 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 RoundSync's improvements in Section III-A3 for detail). In such cases, ChronoSync falls back to a *recovery* mechanism: when a node observes an unknown digest, it triggers a special *Recovery Interest* containing the unknown digest; the nodes who recognize the digest reply with the complete information about the sync tree, rather than the specific changes that led to the digest. When the requesting node receives the reply, it merges the received sync tree into its local sync tree by selecting the higher sequence number for every node in the sync tree.

3) RoundSync: ChronoSync's inability to efficiently handle simultaneous data generation led to development of the RoundSync [9] protocol. In ChronoSync, the function of Sync Interests is overloaded to (1) detect different states among the sync nodes, and (2) to retrieve the updates from other nodes. As a result, Sync Replies carrying the updates to the shared dataset are 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 cannot derive the name for those Sync Replies and therefore cannot send Interests to retrieve them. Merging the diverged sync states only creates new set of sync states, potentially contributing in further divergence of the states. To re-synchronize, ChronoSync relies on a recovery mechanism to receive the entire sync state.

RoundSync removes the need for a recovery mechanism by dividing the synchronization process into *rounds*, updating the semantics of the Sync Interest, and introducing a new type of Interest packet called *Data Interest*. Sync Interests in RoundSync, augmented with round number information, serve only as a notification mechanism to inform other sync nodes about the state in the round. When a divergence is detected, nodes can request the change in the round using a Data Interest[8]. Therefore, published data within a specific round can be retrieved even if the states are not fully synchronized. The replies to a Data Interest have the same functionality as the Sync Reply in the original ChronoSync design, i.e., they carry the prefix and sequence number of the newly published 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 reducing the chances of state divergence caused by simultaneous data production.

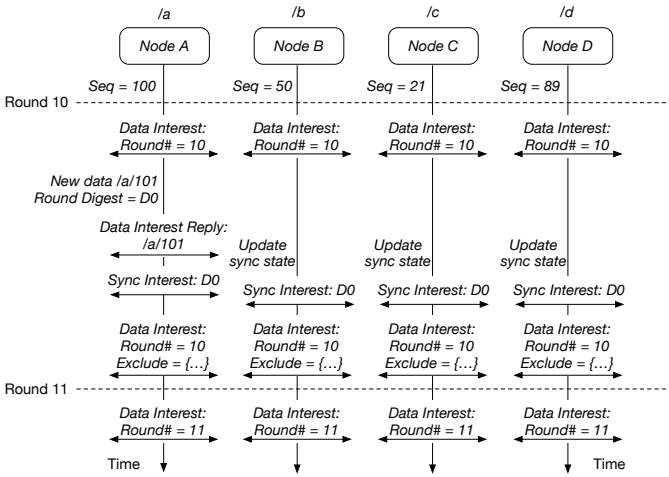

Fig. 8: Synchronization process in RoundSync

In the example in Fig. 8, 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 *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 for the rounds before S and resumes normal RoundSync operation for the rounds after S.

4) Quadtree Sync: The Quadtree Sync Protocol (QSP) [10] is tailored for the synchronization of data with a geographic context, such as found in online games or GIS applications.

[7]Please note that exclude filters were removed in the NDN Packet Format Specification version 0.3. Hence, the latest ChronoSync implementation enters the recovery mode without using exclude filters.

[8]Names of Data Interests do not include a state digest but only the round number.

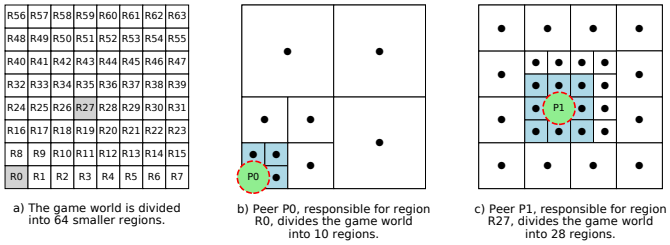

Fig. 9: Hierarchic structure of geographic regions in Quadtree Sync. (Figure adopted from [10])

While this specialization on geographic data renders QSP not being usable as general purpose sync protocol, the protocol shows similarities to CCNx 0.8 Sync and improves on the data dissemination delay.

QSP structures a geograhpic area by covering it with a quadtree[9]. The root of the tree represents the whole area to synchronize; every tree level divides an area into quadrants, making the represented area smaller with every tree level. The tree's leaf nodes represent the smallest regions handled by QSP and are referred to as map chunks. Every map chunk has assigned a sequence number that is increased whenever the region changes. In addition, the quadtree is built as Merkle tree, allowing to efficiently find changed map chunks when comparing multiple copies of a tree. By including the quadtree structure in the naming scheme, a name in QSP directly refers to a region of the synchronized area. These semantically enriched names allow applications constructing names and requesting information based on the application needs, instead of worrying about network-related addressing issues.

QSP assumes that a sync participant is responsible for managing a certain region of the overall geographic area and publishes changes of that region. Besides, changes outside the participants region (referred to as *remote regions*) can be subscribed. Therefore, every sync participant maintains a copy of the quadtree covering the whole geographic region. For subscribing to a remote region, participants issue a Sync Interest having the name of the remote region as name. In addition, the local hash value of the corresponding tree node is appended to the Sync Interests name. The sync participant managing the requested region (referred to as *region owner*) receives the Sync Interest and compares the received hash value to its own hash value. In case the hash values match, the region did not change and no further synchronization is required. If the hash values do not match, a change occurred; synchronization is required. In this case, the region owner can decide how to proceed. It can either i) lookup changed chunks in a local history and return the changes, ii) send back hash values of the requested regions child nodes (or even of child nodes several levels lower in the tree to speed up synchronization), or iii) enumerate the sequence numbers of all leaf nodes belonging to the requested region in the Sync Interests answer. The selected option depends on the current situation. In case no lookup of changed map chunks is possible, lower level hash values, or sequence numbers of leaf nodes need are returned. In case the requested region is large in size, enumerating sequence numbers might not be feasible. Lower level hash values are returned and changes are retrieved by followup Sync Interests for smaller regions. Two examples for structuring an area in smaller regions using QSP are visualized in Figure 9.

QSP differs from the basic sync protocol design in two aspects. First, Sync Interests are neither used to notify about new data in a push-style nor by using long-lived Interests. Instead, changes are requested periodic Sync Interests. Second, the protocol works without relying on a multicast prefix, since Interests do not target all participants, but a specific region of the area to synchronize.

Besides, one unique feature of QSP is the clear separation of sync participants from the application namespace. While other sync protocols assume participant-specific data prefixes, QSP uses the application's quadtree-based namespace for Sync and Data Interests and thereby prevents binding the data to specific participants. This eventually allows a change of region responsibilities during the sync process, without requiring to communicate this change of responsibilities to other participants.

B. Manifest-based Sync Protocols

Manifests are usually used to store metadata of applications. The idea of manifest-based sync protocols is to encode the dataset's sync state in manifests. Exchanging these manifests informs sync participants about the current sync state. Currently, only one manifest-based sync protocol is available, which is discussed in the following.

1) CCNx 1.0 Sync: The design proposal of CCNx 1.0 Sync [11] abandons the CCNx 0.8 Sync design and adopts a simple manifest-based solution. 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.

Each sync node uses Interest packets to advertise the hash of its local catalog manifest when it generates new data (cf. Sync Interest in Figure 2). These advertisement Interests are named under the data collection prefix and forwarded to all sync nodes announcing that prefix. These Interests have a short lifetime and do not retrieve any data. To increase the possibility that all nodes can receive the advertisement, the advertisement is repeated once or twice within a few seconds after the first advertisement is sent. Once a node receives a different hash, it advertises its own hash under the control of a gossip protocol (with random backoff and duplicate suppression). It then sends out Interests to retrieve the corresponding (possibly segmented) manifest packets, compares the names listed in the manifest with its local namespace, and then retrieves the missing data over the network.

[9]Quadtrees are tree-based data structures, where every non-leaf node has exactly four children.

Compared to the basic sync protocol design from Section II, CCNx 1.0 Sync requires one additional step between data announcement and publication retrieval. While the Sync Interest only communicates a digest, the actual dataset changes need to be retrieved as Manifest file using Interest-Data exchange. Only thereafter, the actual publications can be retrieved.

C. Invertible Bloom Filter-based Sync Protocols

Bloom filters (BF) [12] are a probabilistic data structure that allow for efficient membership testing by using a bit array. This allows BFs to be used for testing whether an element is part of an existing dataset or not. However, the foundation of BFs is probabilistic, resulting in membership responses can be false-positive (but not false negative). The false-positive rate can be controlled by the BF size.

Invertible Bloom Filters (IBF) [13] are an extension of BFs and allow for operations beyond membership testing only. With IBF's set operations, such as difference calculation, become possible, which might be well-suited for sync protocols. Current representatives of IBF-based sync protocols are *iSync*, *PSync*, and *syncps*.

1) iSync: iSync [14] supports the synchronization of shared data with arbitrary application names. To efficiently represent the sync state, iSync uses IBFs to store all the names from the shared dataset in a compressed form. Since the IBF can only store fixed-length items, data names are first mapped to fixed-length IDs (generated by hashing the names) before they are added to the IBF. Additionally, a bi-directional mapping table is maintained by every sync participant so that it can recover the original NDN names from IDs.

iSync uses "digest broadcast" Interests to advertise its current state to other nodes periodically (cf. Sync Interests). Since the encoded size of the IBF is typically large, the advertisement Interest only carries a digest of the current IBF from the sending node. When a node receives a digest different from its own, another Interest to request the corresponding IBF content is issued. After receiving the IBF, the node subtracts its own IBF from the received IBF and extracts the individual IDs from the resulting "difference" IBF. Once the new IDs are extracted, the original NDN names corresponding to those IDs are requested, and finally the new publication can be retrieved by using the original names. An example of iSync's synchronization process is shown in Fig. 10.

One limitation in the IBF data structure is that it can only losslessly encode up to a certain number of items. Beyond that, some of the stored items cannot be extracted. To prevent having too many stored items, 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 enforces each node 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 nodes. Third, iSync creates multiple *local IBFs* to record the small-step changes during each sync period. When the advertised IBF (called *global IBF*) contains too many changes, the sync node can fetch the local IBFs instead and perform more fine-grained difference reconciliation.

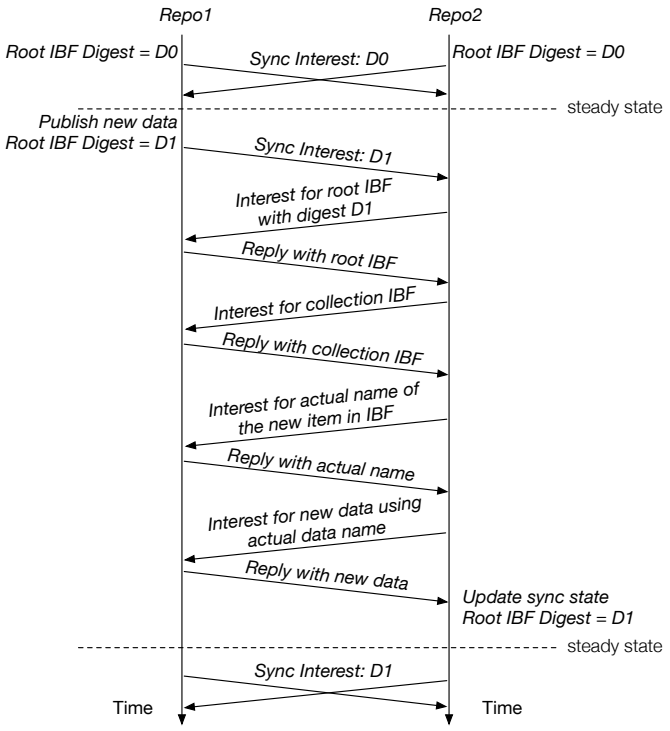

Fig. 10: Synchronization process in iSync

2) PSync: The PSync protocol [15] is designed to support two use-cases. The *partial sync mode* allows synchronizing multiple consumers with a subset of a large data collection maintained by a single producer. The *full-sync mode* offers multi-producer/multi-consumer communication, as provided by other sync protocols. Publications published by producers are organized into *data streams*, where each stream is identified by a unique data stream prefix. Individual publications of a data stream are enumerated by appending a sequence number to the data stream prefix. PSync employs IBFs to represent the dataset state by storing the hash values of the data names in fixed-length slots of the IBF. By exploiting the stream-based namespace structure, the IBF only needs to store the latest data name of each data stream. This reduces the amount of information stored by the IBF and hence, allows reducing the amount of data transmitted over the network.

To support the synchronization of a subset of the dataset (*partial sync mode*), PSync introduces a subscription list allowing consumers to specify the data streams that it is interested in[10]. The subscription list is a BF that stores the data stream prefixes the consumer is interested in. The size of the BF is determined by the total number of streams a consumer may want to subscribe to and the false positive rate the consumer is willing to accept[11].

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

[11]Special cases like full subscription may be encoded more efficiently with special markers.

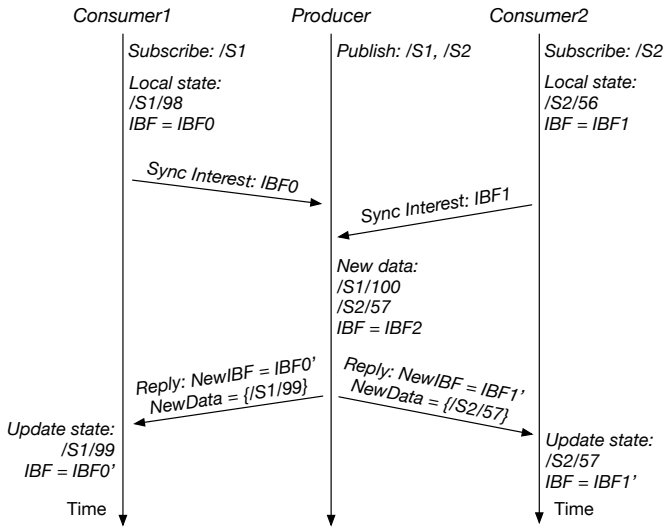

Fig. 11: Partial Sync Process of PSync

Consumers need to learn which data stream prefixes are contained in the dataset before they can join the sync process. Therefore, PSync provides an initialization phase, in which consumers request the available data stream prefixes by sending a *Hello Interest* to the producer. As a response, all data stream prefixes including the corresponding latest sequence numbers are returned. During the sync process, a 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 to retrieve the latest dataset state, a consumer maintains a *long-lived Sync Interest* whose name contains the consumer's local IBF copy and the subscription list. When the producer publishes new data, it first subtracts the IBF of the pending Sync Interest from the new IBF and thereby extracts the names of publications that have not been received by the consumer yet. In the next step, the producer checks whether the data stream prefixes of those new publications are included in the consumer's subscription list. If they are in the subscription list, the producer generates a Sync Reply containing the original names of the new publications and the difference IBF that only contains the new data names. Upon receiving the Sync Reply, the consumer updates its local IBF copy with the received IBF and retrieves the new publications by Interest-Data exchange. Fig. 11 visualizes an example of PSync's synchronization process.

In the *full-data synchronization mode*, all sync participants are producers and consumers at the same time. Also, every participant subscribes to the full dataset, which allows dropping the subscription list in Sync Interests. Long-lived Sync Interests are sent to the multicast prefix and kept pending until a new publication is produced. When a new publication is generated, it is added to the producer's IBF, which is sent as a reply to the pending Sync Interests. This reply notifies all sync participants about the newly produced publication.

Comparing PSync's full sync mode to the basic sync protocol design from Section II, we would like to highlight that PSync employs a pull-based approach for the initial Sync Interest. Participants request changes via long-lived Interests sent to the multicast prefix. The use of long-lived Interests, however, is seen as controversial. A long-lived Interest has to be kept pending on all forwarding nodes for a substantial time and thereby consumes resources on all forwarding nodes. On the other hand, in PSync Interests are used for their intended purpose – requesting Data – instead of for pushing notifications without awaiting a response, as done by other protocols.

Focusing on Interest aggregation, we see that the partial sync mode allows every consumer to define the data they are interested in. This design results in different names for every consumer and prevents Interest aggregation. Besides affecting traffic volume, Interest aggregation potentially reduces the load on the Interest-receiving side. Producers in PSync do not simply answer requests with ready-to-use data but need to calculate the response for every request on the fly by subtracting the received IBF from the producers IBF. This might result in higher processing overhead for producers. With the full data synchronization mode, however, Interest aggregation works as expected. Besides, the receipt of Interests in PSync does not affect the dataset state, which removes the need for authenticated Interests.

For discussing scalability, we revisit the exchanged information sent during the sync process. In partial sync mode, a Sync Interest carries the subscription list as BF and dataset state as IBF. The size of the subscription list depends on the number of subscribed data streams, the size of the IBF, on the number of expected differences. The limited capacity of Interest packets, however, does not allow growing these data structures beyond a certain size limit. This ultimately limits the maximum number of data streams of PSync and thereby negatively affects scalability.

3) syncps: The main idea of the *syncps* protocol [16] (also known as publish/subscribe sync) is to synchronize lifetime bounded publications by using IBFs for dataset state representation. *syncps* behaves fundamentally different from other protocols since it abandons the separate Interest-Data exchange for publication retrieval (cf. step 2 in the basic sync protocol design of Fig. 2). Instead, the actual publications are transmitted in the response to Sync Interests, leading to low protocol overhead and data dissemination delay.

The protocol design described in this report was reverse engineered from *syncps*'s open-source codebase[12]. The publications' arbitrary data names are hashed to fixed-sized digests and inserted into an IBF. Publications are assumed to be valid only for a predefined time after which they are removed from the IBF again. Sync Interests in *syncps* carry the sending participants IBF and are realized as long-lived Interests that are immediately sent when new Data is published, or periodically before the previous Sync Interest times out. On receipt of a Sync Interest, participants subtract the received IBF from the local IBF and thereby find new publications produced by other participants, as well as publications that are known locally but not by the remote participant. For the latter case, up to multiple publications missing by remote participants are packaged in

[12]https://github.com/pollere/DNMP-v2/blob/main/syncps/syncps.hpp, last accessed: 2021-02-19

the Sync Interests response Data (piggybacking of multiple Data packets in a single one).

Publications produced by other participants are retrieved by sending a Sync Interest including the local IBF (indicating the missing publications). Remote participants receiving the Interest detect the absence of the latest publications and transmit the missing publications in the response Data.

While *syncps* reduces overhead and data dissemination delay, the piggybacking approach reduces the network's ability to make use of semantic name information. Multiple publications, each having a unique name, are summarized in one Data packet having a rather generic sync group-specific name. The name of individual publications could be relevant for multiple participants of the sync group. Besides, the publications name is required for network-level multicast and in-network caching to work. The Sync Interest response Data, however, might be created for a single participant only, possibly reducing communication to a host-to-host pattern.

Focusing on scalability, we can identify the IBF in Sync Interests and the piggybacking approach as limiting factors. The required IBF size depends on the number of data streams in the dataset. With many data streams, the IBF size needs to grow to prevent false positives. Since the IBF is encoded in Sync Interests, this is only possible up to a certain degree. However, the lifetime-bound of publication lowers this issue but brings other issues, such as the requirement for time-synchronized sync participants.

The piggybacking approach assumes that publications are small in size and not created simultaneously. Adding NDN Data packet fields to a large publication might already exceed the MTU and leads to becoming unable of piggybacking. The same holds for many simultaneous publications. In this case, only a subset of publications can be added to one Data and the transmission of the remaining is postponed to the next Sync Interest.

D. State Vector-based Sync Protocols

The class of state vector-based sync protocols is distinguished by the use of state vectors for representing the latest dataset state. This idea originates from Vector Clock [17], a data structure used to provide a partial order to events occurring in distributed systems. A notable difference between state vector-based protocols compared to previous developments is denoted by the form of the dataset representation in the initial Sync Interest. When sending a digest in the Sync Interest, receivers of the Interest can only infer whether the dataset changed or not, but no information about the exact change can be inferred. This makes consecutive communication to retrieve details about the changes necessary. The state vector-approach can communicate not only information about the existence of a change, but concrete dataset changes in the same message. This potentially leads to lower communication overhead and improved performance.

A number of state vector-based protocols were proposed. The basic concept of these protocols is very similar in respect to protocol design and dataset representation and only differ in details. The VectorSync protocol [18], [19] laid the foundation for later developments. Later protocols mainly optimize on different aspects of VectorSync. This report focuses on providing a detailed introduction to VectorSync in Section III-D1 and summarizing the differences of later protocols to VectorSync.

1) VectorSync: The VectorSync protocol by Shang et al. [18], [19] was the first sync protocol that represents the dataset's state by using a state vector. In VectorSync, each producer publishes data under its unique name prefix with a continuous sequence number allowing to keep track of individual publications. The latest publication of a producer is denoted by the latest sequence number; consequently the latest dataset state is known when knowing the latest sequence number of every producer. This information is contained in the state vector.

As indicated in Figure 12a, VectorSync's state vector encodes sequence numbers for every producer prefix. Receiving a state vector allows comparing the received sequence numbers to the local copy of the dataset. If any of the received sequence numbers is higher than in the local dataset, the local dataset is updated and the actual Data can be retrieved from the corresponding producer prefix.

To make use of the received sequence numbers, a mapping from state vector entries to producer prefixes is required. This mapping is referred to as group membership list. The group leader – a designated sync participant responsible for keeping track of the sync group's participants – is responsible for maintaining an up-to-date group membership list.

Protocol Design: Focusing on the protocol design, we see that VectorSync is required to handle two types of data: i) the current dataset state is synchronized by distributing state vectors in Sync and Heartbeat Interests. ii) the sync group's membership list is distributed by the group leader that keeps track of the active participants. In the following, both parts of VectorSync's protocol are discussed.

The protocol for distributing dataset changes via state vectors is visualized in Figure 13. Whenever a producer (eg. Alice) publishes new data, a *Sync Interest* is broadcasted to the group prefix (message 1). The Sync Interest contains the latest sequence number of Alice's data stream and allows the other participants (Bob and Ted) to update their local dataset with the received sequence number. As a response, a Data containing the updated state vector is sent (message 2). After having learned about Alice's new publication, Bob and Ted request the newly generated data by Interest-Data exchange (messages 3 and 4).

The second task – maintaining the group's membership list – is implemented by a leader-based approach. Every sync participant broadcasts periodic Heartbeat Interests including its producer prefix and state vector to the sync group. By receiving heartbeats, sync participants detect other participants (and additionally get informed about changes in the dataset). Also, by convention, the participant with the highest order producer prefix is chosen as the group leader. Based on the information received from Heartbeat Interests, the group leader creates and distributes versioned *membership info* objects containing information about all sync participants (eg. data publishing prefixes, and public key certificates). If participants

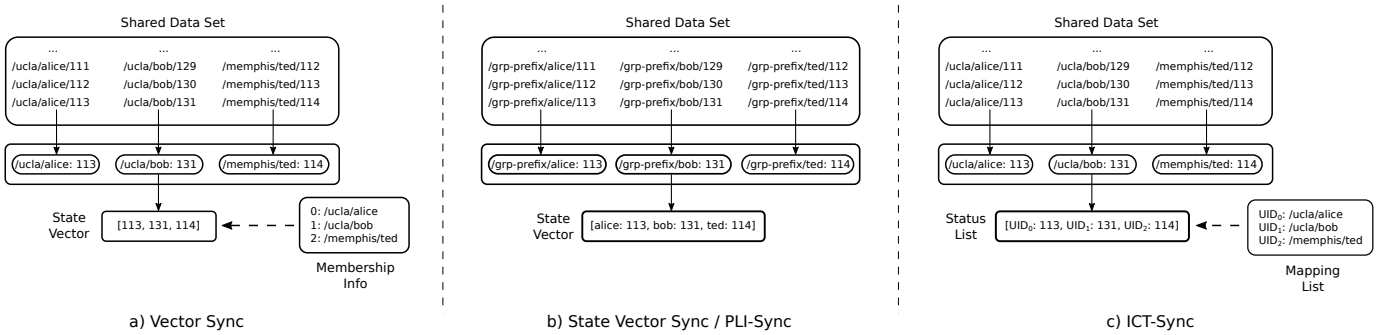

Fig. 12: Relation between application namespace and state vector structures in current state vector-based sync protocols.

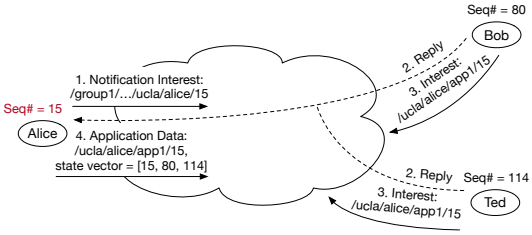

Fig. 13: VectorSync's message exchange for dataset synchronization in a group of three parties. (Figure adopted from [19])

enter or leave the sync group, a new version of the membership info object is created and distributed.

Combining the membership management and dataset state synchronization, VectorSync becomes able to synchronize datasets across multiple producers and consumers without the need for long-lived Interests, such as required by other protocols. However, VectorSync's design allows for optimizations, which are discussed in the following.

Intermittent connectivity: The separate group membership management part of the protocol is a limitation in scenarios with intermittent connectivity. Without having the latest version of the membership info object, the individual entries of a state vector can not be mapped to producer prefixes. Whenever new producers join or leave, the group leader publishes a new version of the membership info object, requiring all sync participants to retrieve it before continuing with dataset synchronization.

Focusing on scenarios with intermittent connectivity, such as wireless ad-hoc networks, sync participants can frequently get disconnected from and reconnected to the rest of the sync group. This churn requires the group leader to regularly publishing new versions of the membership info object. Without up-to-date membership info, sync updates can not be interpreted, which adds additional complexity to the protocol. With network fragmentation, this becomes even worse. Network fragmentation may result in the sync participants getting split into multiple sub-groups. Each sub-group needs to agree on a group leader that publishes membership info objects without containing the participants located in other sub-groups.

Reviewing VectorSync's design, it becomes apparent that handling memberships complicates the sync process and shows potential for improvement. However, VectorSync's membership management allows exchanging other relevant information, such as data prefixes, sync participants' certificates, or group keys. This partially makes-up for the complexity added by membership management.

Delay Tolerant Networking: A concept related to intermittent connectivity scenarios is Delay-Tolerant Networking (DTN). With intermittent connectivity, participants get occasionally disconnected from others. To allow temporarily disconnected nodes to receive data, asynchronous concepts for communication are required. DTN overcomes this absence of end-to-end communication paths by storing packets and forwarding them opportunistically to neighboring nodes. This need for storing packets in the network indicates DTN's suitability for content-centric concepts. Epidemic routing (ER) [20] is one of the most prominent concepts for data distribution in DTN and already moves towards a data-centric direction. In ER messages are stored on nodes and continuously replicated to newly discovered contacts, which employs a flooding-based behavior. While ER assigns messages unique identifiers that are comparable to flat names, ER is still based on a connection-oriented architecture. This requires handling message caching and replication on the application layer. Besides, ER does not employ concepts for end-to-end security. In contrast, VectorSync as an NDN-based sync protocol allows using both: in-network caching, and data-centric security concepts. This allows for improvements in security and application complexity when used in DTN. However, VectorSync's membership management approach does not work in DTN. Hence, unveiling the full potential of data-centric architectures requires improvements concerning membership management.

2) StateVectorSync: The *StateVectorSync* protocol (SVS)[13] [21], [22] is the successor of VectorSync and aims to support dataset synchronization in disruptive networks. In such networks, network fragmentation occasionally happens and challenges VectorSync's group membership management.

The logical consequence to improve on VectorSync's limitation is removing the leader-based membership management. This is implemented by slight modifications of the state vector, as depicted in Figure 12b. Instead of encoding sequence numbers only, SVS encodes tuples consisting of the producers' prefixes and their latest sequence numbers. Otherwise, the protocol design for dataset synchronization is similar to

[13]The StateVectorSync protocol is also referred to as *Distributed dataset Synchronization over disruptive Networks* (DDSN)

VectorSync. Differences in the protocol are described in the following. SVS's Sync Interests contain the full state vector of the sending sync participant, instead of the sequence number of a single data stream only. Thereby, other sync participants can not only update the state of one data stream but of the complete dataset. As in VectorSync, the response to the Sync Interest contains the updated state vector. The actual publication retrieval stays unchanged.

By removing the necessity for a separate group membership list, SVS becomes capable of working in environments with intermittent connectivity. This is shown by a comparison of SVS and ER [23]. Simulation results show superior performance in regards to retrieval delay and resilience to packet loss.

One limitation of the SVS design is concerning limited flexibility in regards to the producer's namespace. While arbitrary producer prefixes can be encoded in VectorSync's group membership list, the encoding of the producer's name in SVS only allows a single name component to identify the producer. This is one result of removing VectorSync's membership management approach. However, we want to highlight the possibility of allowing arbitrary prefixes by conventions on application-level, or by adding one level of indirection.

3) PLI-Sync: The *Prefetch Loss-Insensitive Sync* protocol (PLI-Sync) [24], [25] is built on top of SVS and aims to support robust group communication in disadvantaged wireless networks. The key novelty of PLI-Sync is to combine a slightly modified version of SVS with opportunistic content prefetching to increase synchronization efficiency. While earlier sync protocols allow consumers to decide which Data to retrieve (messages 3 and 4 in Figure 13), participants in PLI-Sync retrieve all published Data. This design choice was made to strengthen resilience in networks with intermittent connectivity by distributing data as often as possible, and thereby increasing redundancy.

Concerning namespace design, every sync participant is assigned a `nodeId`. Under this identifier, multiple data streams can be published, each identifiable by a `streamNo`, leading to a naming scheme as follows:

`/<grpPrefix>/n/<nodeId>/stream/<streamNo>/<seqNo>`

Opportunistic prefetching is realized by sync participants issuing Interests for sequence numbers not yet produced. These Interests stay pending in the network until the next data item is produced and result in instant delivery of thereof. In addition to prefetching, SVS acts as a fallback solution to counteract packet loss. Besides, SVS is used for detecting newly generated data streams, which is not possible with prefetching only.

Evaluations of PLI-Sync show promising results in heavy loss scenarios. However, PLI-Sync's requirement that all participants need to fetch all data might reduce PLI-Sync's suitability to being used as a general-purpose sync protocol.

4) ICT-Sync: The ICT-Sync protocol [26], [27] is based on a state vector and aims to support asynchronous communication. As depicted in Figure 12c, ICT-Sync maintains two separate data structures for representing the producers of the sync group and the dataset state. The *mapping list* (`ml`) maps data prefixes to unique identifiers (UID). These UIDs are used in the *status list* (`sl`) – a state vector-like structure – for assigning the latest sequence number to all UIDs.

The protocol design is similar to VectorSync, but terminology changed. VectorSync's state vector is referred to as *status list* (`sl`), and the *mapping list* (`ml`) is a structure similar to VectorSync's membership info. Dissecting ICT-Sync's synchronization process, the protocol uses a two-step approach for dataset synchronization. In the *join*-phase, sync participants join the sync group by sending a Sync Interest containing their (initially empty) `sl` to the group prefix. Thereby, every participant starts as a consumer and is informed about the current dataset state. In the second phase – the *publish*-phase – producers register data prefixes by adding them with corresponding UIDs to their local `ml`. The new UIDs are also added to the `sl` and are appended as state vector to the Sync Interest's name. Other participants are informed about new data by receiving this Sync Interest. Incoming Sync Interests that not yet contain the newly added UIDs are answered with Data containing triples in the form of `<UID; data-prefix, latest-sequence-no>`. Receiving this Data allows other participants adding information about the new data in their `ml` and `sl`.

A breaking change introduced by ICT-Sync is the optional use of intermediate nodes that are utilized for enabling asynchronous communication and to provide data persistence. Intermediate nodes may be deployed in the network and act like standard sync participants, with the difference that produced data is immediately fetched, validated, and persisted. This allows intermediate nodes to provide a copy of the published data when original data producers become unavailable, and hence, provide means for asynchronous communication.

While ICT's intermediate nodes increase data availability, they might require to revisit security considerations. By using intermediate nodes, network components are legitimized to act as sync participants. They build their own view of the dataset and create Sync Interests based on their knowledge. Thereby, intermediate nodes not only forward Interests but create and broadcast their own messages to the sync group. This elevates the responsibility of network components from performing transport-level operations to application-level tasks and requires other participants to trust intermediate nodes. For security-critical use-cases, it might not be feasible to extend trust to network nodes that might be in control of third parties, such as network operators.

Focusing on Interest aggregation, we suggest using group keys for Interest authentication in Section II-B. With intermediate nodes being part of the sync group, a group key approach means that intermediate nodes become able to create signatures that are indistinguishable from signatures of regular participants.

While intermediate nodes might open trust-related issues, ICT-Sync improves on VectorSync by removing the leader-based membership management approach. As a replacement for VectorSync's membership info objects, ICT-Sync's `ml` keeps data of individual producers, such as producer prefixes, and thereby allows arbitrary data names.

IV. Discussion on Protocol Design Decisions

In the previous section, we introduced existing NDN sync protocols. We now explicitly focus on specific design aspects and the range of possible solutions. We reflect how individual protocols differ and reflect positive and negative influences of concrete decisions. Table II provides a preliminary comparison of performance metrics[14] of the existing NDN sync protocols.

A. Sequential vs. Arbitrary Data Naming

One of those design aspects is data naming. ChronoSync, RoundSync, Quadtree Sync, PSync, and state vector-based protocols adopt a sequential data naming convention. This means that data packets are named using sequence numbers under a common name prefix for each producer (resp. data stream). This design choice simplifies the representation of the shared dataset's namespace. Having continuous and monotonically increasing sequence numbers in the data name allows producers to summarize their data collection by the highest sequence number. This reduces the amount of information that needs to be encoded in the sync state and simplifies the protocol design since the sync protocol only needs to focus on synchronizing the latest sequence numbers rather than the whole namespace.

Looking at protocols with sequential data naming, we identify varying freedom in choosing data prefixes. ChronoSync, RoundSync, PSync, VectorSync, and ICT-Sync allow producers to specify their own data prefix. SVS, PLI-Sync, and Quadtree Sync, however, force sync participants to use the group prefix as a common prefix and use the producer's identifier as an intermediate name component only. We want to highlight that the restriction of being bound to a common data prefix often comes with simplified protocol design. Eg. to allow using producer-specific prefixes, VectorSync and ICT-Sync require additional mappings from producer identifiers to actual prefixes. Maintaining these mappings is integrated into protocol design and bloats the actual sync process. SVS, in contrast, does not need a separate mapping and the receipt of a single Sync Interest already allows inferring the dataset state. More complex naming schemes may be supported by adding one-level of indirection, or by application-level conventions.

In contrast to sequential naming, CCNx 0.8 Sync, CCNx 1.0 Sync, iSync, and syncps allow for arbitrary application names. These protocols allow synchronizing datasets where individual publications are not related to each other and provide the most flexibility. This flexibility, however, comes with higher overhead during the sync process or similar limitations. So for instance, CCNx 0.8 Sync requires a request-reply iteration for every level of the protocols name tree, CCNx 1.0 Sync maintains a (potentially large) manifest file that needs to be retrieved for every update, iSync handles up to multiple levels of indirection to provide scalability, and in syncps the number of producers is limited by the maximum size of IBFs that can be encoded in Interest names.

[14]The basis for comparison in Table II is a qualitative evaluation of protocol design aspects rather than a quantitative experiment.

B. Push Notifications vs. Pull Subscriptions

The existing sync protocols typically use one of the two communication models for propagating the information about the new data published in the sync group. The first model – referred to as *push notifications* – is to piggyback information about dataset changes in multicast Interests. Participants that receive these Interests can directly change their dataset state and are informed about the latest change. The second model – referred to as *pull subscriptions* – is that sync nodes send "long-lived" Interests to each other (typically using multicast) to request data packets that carry information about updates to the sync state. Without immediate changes to report, these "long-lived" Interests essentially become temporary "one-packet" subscriptions to sync state updates generated in the future.

Pull subscriptions using long-lived Interests require the network to maintain a soft state (implemented by the Interests being kept pending on all forwarding nodes) and consume resources on forwarders in the network. Also, dropped long-lived Interests (eg. due to congestion) are not detected by the Interest-sender until timeout, potentially decreasing the protocol's performance.

Ideally, to reduce overhead long-lived Interests are aggregated in the network, resulting in a multicast of Data upon generation. NDN's flow balance principle, however, leads to a lack of efficient support of simultaneous data generation. A single produced Data satisfies all outstanding long-lived Interests and is efficiently delivered to all other sync participants. The almost simultaneous generation of a second update on a different participant can not be delivered until the previously satisfied long-lived Interests are re-issued. This potentially leads to outdated Data packets being cached in in-network caches and requires multiple rounds of Interest-Data exchange until a consistent state is reached.

Push notifications are implemented as broadcast Interests that piggyback a notification. This approach misuses Interests for pushing data, while intended for data retrieval only. Concerning reliability, lost notification Interests can be mitigated by adding redundancy. Acknowledgments in the protocol design may be used by the notification producer to ensure others received the message by repeating the notification until acknowledged.

One potential issue of Interest notifications is that Interests are not designed to be authenticated and thereby allow abusive use if not properly secured. Whenever an Interest changes the sync state of recipients, the Interest needs to be authenticated (eg. by using a cryptographic signature). Such authentication, however, can limit the ability of Interest aggregation.

Example sync protocols utilizing pull-based subscriptions using long-lived Interests are ChronoSync, PSync and syncps. The family of state vector-based protocols employs push notifications by sending authenticated Sync Interests on Data generation.

C. Data Dissemination Delay

One metric relevant for sync is the data dissemination delay, i.e., the number of round-trips (RTTs) necessary to propagate

TABLE II: Comparison of existing NDN sync protocols on common performance metrics

	CCNx 0.8 Sync	**iSync**	**CCNx 1.0 Sync**	**ChronoSync**	**RoundSync**	**PSync**
Data dissemination delay	Interest period + tree walk	Interest period + 3.5 RTT (+ RTT to retrieve local IBFs)	$\geq$ 1.5 RTT (influenced by manifest size)	min. 0.5 RTT for notification + 1 RTT for update retrieval	min. 1.5 RTT for notification + 1 RTT for update retrieval	0.5 RTT for notification + 1 RTT for update retrieval
Interest overhead	Periodic	Periodic	One per update	Long-lived Interest	Long-lived Interest + Data Interest	Long-lived Interest
Factors affecting Sync Interest size	Node hash	IBF digest	Manifest digest	State digest (+ exclude filter)	Round digest (+ exclude filter)	IBF + subscription list
Factors affecting Sync Data size	Number of children under the requested node	IBF size (depending on the number of new data)	Number of dataset entries	Number of names with new seq#	Number of names with new seq# in a round	IBF size + number of names with new seq#
	syncps	**VectorSync**	**SVS**	**PLI-Sync**	**ICT-Sync**	**Quadtree Sync**
Data dissemination delay	0.5 RTT (Data as response to long-living Sync Interest)	0.5 RTT for notification + 1 RTT for update retrieval	0.5 RTT for notification + 1 RTT for update retrieval	0.5 RTT with opportunistic prefetching	0.5 RTT for notification + 1 RTT for update retrieval	Min is 1.5 RTT, longer with high amount of changes
Interest overhead	Long-lived Interest	Periodic	Periodic	Periodic (low frequency, used as fallback only)	Periodic	Periodic per remote region + possible tree traversal
Factors affecting Sync Interest size	IBF size	Number of sync group members	Number of publishers	Number of data streams	Number of data streams	Constant size
Factors affecting Sync Data size	Number and size of new publications	Number of sync group members	No response to Sync Interests	No response to Sync Interests	No response to Sync Interests	Number of changes

new publications to the other members of the sync group. This delay is substantially influenced by the protocol design. Protocols that allow inferring the datasets state from a single message (eg. state vector, or IBF in Sync Interests) employ a lower data dissemination delay than protocols that only communicate a digest representing the datasets state. In the latter case, a digest only allows inferring that the dataset changed, but requires additional communication to locate the actual change. These additional messages increase the overall data dissemination delay.

In CCNx 0.8 Sync, Sync Interests carry a digest representing the current state of the datasets name tree, or of a subtree. The synchronization process is triggered periodically based on an internal sync timer. If the digest indicates a change, the change is obtained by traversing the name tree. The number of RTTs required for tree traversal and hence, to retrieve all updates, depends on the depth of the sync tree. In iSync, the process usually finishes within 3.5 RTT, unless the number of changes exceeds the capacity of the global IBF in which case the nodes need to retrieve additional "local IBFs". CCNx 1.0 Sync triggers the synchronization process when new data is published by pushing a digest of the datasets manifest to remote participants. After receiving a changed manifest digest (0.5 RTT) participants start retrieving the updated manifest. The dissemination delay depends on the manifest size.

ChronoSync achieves best synchronization delay without simultaneous data publishing. Notification about new publications is delivered as response to the long-lived Sync Interest in 0.5 RTT, Interest-Data exchange for publication retrieval adds 1 RTT and results in a data dissemination delay of 1.5 RTT. If multiple nodes generate Sync Replies at the same time, the protocol needs additional round-trips to retrieve all Sync Replies using Interests with exclude filters. The dissemination delay grows proportional to the number of simultaneous updates in the group and is bounded by the number of data sync participants.

In PSync, long-lived Sync Interests sent by the consumer carry the consumer's state. This allows the producer to reply with information about specific changes as soon as they occur, leading to 0.5 RTT for the notification of a change. The consequent Interest-Data exchange required to fetch the new publications results in a data dissemination delay of 1.5 RTT.

Similar to PSync, syncps assumes sync participants to maintain long-lived Sync Interests carrying the local state in an IBF. In contrast to PSync, syncps' replies carry actual publications instead of notifications about generated publications only. This immediate response leads to a data dissemination delay of 0.5 RTT, potentially getting larger with simultaneous publications. Simultaneous publications on a single participant are packed in a single response data and does not necessarily increase the data dissemination delay. If publications are generated by multiple participants simultaneously, all publishing participants reply to the same Sync Interest. This procedure results in only one response getting delivered. Outstanding

publications are retrieved by consequent Sync Interests, increasing the data dissemination delay by 1 RTT for every simultaneously publishing participant.

The function of existing state vector-based protocols is similar and hence, results in similar data dissemination delays. The notification of new publications via Interest notification takes 0.5 RTT; the consecutive data retrieval takes 1 RTT for Interest-Data exchange. The only vector-based protocol with fundamentally different behavior is PLI-Sync, where opportunistic prefetching may reduce the dissemination delay to a minimum of 0.5 RTT.

The data dissemination delay for the Quadtree Sync protocol depends on various factors, such as the tree size, and the number of changes. The basic concept of Quadtree Sync is similar to CCNx 0.8 Sync but improves the tree traversal process. Instead of retrieving changes for every single tree level, sync participants in Quadtree Sync request changes for subtrees. This leads to a lower-bound of 1.5 RTT for the data dissemination delay (0.5 RTT for notifying changes, and 1 RTT for the publication retrieval). In the case of a subtree having a high number of changes, Quadtree Sync prevents large response Data packets by delegating the delivery of changes to lower level subtrees. This delegation, however, requires subsequent requests and increases the data dissemination delay.

D. Protocol Overhead

The packet size of sync protocol messages impacts the bandwidth utilization caused by sync communication and thereby becomes a relevant performance metric. In the following analysis, we focus on the size of dataset state updates carried in Interest and Data packets.

In IBF-based protocols, the packet size depends on the configuration of the carried IBF. The size of an IBF is configured in advance and is based on the number of expected data streams (number of expected changes to the dataset in case of PSync) as well as on the acceptable false-positive rate. Once configured, the IBF's size is constant and not dependent on the number of entries carried by the IBF. PSync and syncps carry the IBF in the Sync Interest and thereby add the IBF's size to periodically sent messages. By encoding the IBF in Interest packets, the maximum Interest packet size implicitly limits the IBF's size. This size limitation impacts scalability since it effectively caps number of possible data streams. The iSync protocol carries the IBF in Data packets and broadcasts a digest representing the IBF in Interests only. The IBFs are delivered in Data packets only when the digest changed. This reduces the protocol overhead compared to PSync and syncps.

In ChronoSync and RoundSync, a digest is appended to the Sync Interest's name, and the response Data contains updates to the dataset only. Those updates contain the prefix and the latest sequence number of producers who published new data. In case of simultaneous publishing, sync participants send additional Interests with exclude filters that enumerate implicit digests of all the previously received replies. This may cause the size of the Interest packets to grow linearly with the number of simultaneous replies.

In CCNx 0.8 Sync, the size of the NodeFetch reply packets is proportional to the number of children under the requested node in the sync tree. Depending on the name tree structure, the protocol requires multiple NodeFetch packets to resolve all differences. In CCNx 1.0 Sync, the size of the manifest is proportional to the number of the included data names, representing the state of the complete shared data.

State vector-based protocols append the state vector to the periodic Sync Interest's name. The state vector's size depends on the number of sync participants (resp. number of data streams) and growing the vector beyond the maximum Interest size is not possible. In terms of the Sync Interest that notifies about new changes, different design choices are observed. In Vector Sync, a publishing node only includes the highest sequence number of its own data stream. SVS and most other designs include the whole state vector in the Sync Interest, which increases the packet size but possibly contributes to resolving inconsistencies faster.

V. Open Issues

In this section we discuss open research issues in distributed dataset synchronization that have not been addressed by existing sync protocols discussed in this report.

A. Routing Scalability

All NDN sync protocols rely on Sync Interest multicast for dataset state update exchanges with all the participants in a sync group. Interest multicast on the Internet with the individual participants being distributed over a continent might lead to challenges concerning multicast routing scalability. Supporting Interest multicast on the forwarding-plane requires the per-application multicast prefixes being announced everywhere where sync nodes reside. On a global scale with potentially many different sync groups, routing per-application multicast prefixes might not be feasible.

In the context of NDN, routing scalability is a well-discussed topic. Zhang et al. [28] indicate several proposed solutions to overcome scalability issues. One potential solution is using *forwarding hints* [29]. With forwarding hints, a differentiation between Data *identifiers* and *locators* is made. An identifier is a name that uniquely identifies a piece of information and represents names that are relevant for applications. Locators are names that indicate the possible location where Data can be found and is thereby relevant for the forwarding plane only. Forwarding hints are appended to Interests for routing it towards a location; when reaching the location, the forwarding hint is removed for identifier-based forwarding in the local network. The concept of forwarding hints removes the need for routing application-level names on the Internet and thereby allow for higher scalability.

Besides, other approaches for overcoming routing scalability issues are discussed. Among them, utilizing *Data rendezvous*, or *self-learning*. For more information, we kindly refer the interested reader to [28], which provides an overview of existing solutions.

B. Group membership management

Some functionality, such as snapshot creation, requires a consistent view on the sync group's members. So for instance, VectorSync generates group-wide snapshots of the published data. To do do, VectorSync protocol needs to maintain a consistent view among all participant in a sync group.

Only a subset of existing sync protocols maintain group membership information. Some protocols, such as CCNx 0.8 Sync, or iSync do not use the concept of a sync group at all. In these protocols, the sync state does not reflect the participant's identities that are maintaining the shared dataset. When applications need this information, the group concept has to be built by the application on top of the sync protocol. Especially when focusing on security a sync group concept might yield advantages. Having the group concept in sync could allow eg. to prevent data generation by unauthorized users on the network-level.

C. Consistency and data ordering

Consistency in distributed systems has been studied for decades. Strong consistency models such as *linearizability* [30] enforce a global total ordering of events observed by every node in the system. Weaker consistency models relax the ordering requirements in different ways. In particular, a system with *eventual consistency* allows to diverge and expose inconsistent states during the system execution, as long as it eventually resolves the inconsistency.

For consistency discussions, two factors need to be kept in mind: a) the consistency requirement varies among different applications, and b) existing tradeoffs between consistency and availability might be considered. The primary goal of sync protocols is to facilitate multi-party data-centric communication in distributed systems. Therefore existing NDN sync protocols support weak consistency that favors availability over consistency. Sync protocols allow participants to publish and propagate new data at any time. Without permanent network failure, all sync nodes will eventually receive all data packets published by others. Different consistency models can be implemented on application-level on top of sync protocols to meet the application's requirements.

VI. Conclusion

This report presents an overview of the distributed dataset synchronization problem in NDN and a survey of twelve existing sync protocols. By summarizing and comparing their protocol design, we articulate the different design choices made in existing sync protocols together with their advantages and limitations. We hope that new sync protocols developed in the future can benefit from past experience, as described in this paper, and address the open issues with innovative ideas.

Acknowledgment

This work is partially supported by the National Science Foundation under awards CNS-1345318, CNS-1629922, and CNS-1719403. Lan Wang and John Dehart provided valuable inputs to this second versions of the report.

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.

[3] A. Afanasyev, J. Burke, T. Refaei, L. Wang, B. Zhang, and L. Zhang, "A Brief Introduction to Named Data Networking," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 2018, pp. 1–6.

[4] Z. Zhu, J. Burke, L. Zhang, P. Gasti, Y. Lu, and V. Jacobson, "A new approach to securing audio conference tools," in *Proceedings of the 7th Asian Internet Engineering Conference*, ser. AINTEC '11. ACM, 2011, p. 120–123.

[5] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[6] ProjectCCNx, "CCNx Synchronization Protocol," CCNx 0.8.2 documentation, 2012. [Online]. Available: https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt

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

[8] 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.

[9] 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.

[10] P. Moll, S. Isak, H. Hellwagner, and J. Burke, "A Quadtree-based synchronization protocol for inter-server game state synchronization," *Computer Networks*, vol. 185, p. 107723, 2021.

[11] M. Mosko, "CCNx 1.0 Collection Synchronization," Palo Alto Research Center, Tech. Rep., Apr 2014. [Online]. Available: https://doi.org/10.13140/RG.2.1.4510.1925

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[13] 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.

[14] 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.

[15] 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.

[16] K. Nichols, "Lessons learned building a secure network measurement framework using basic ndn," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ser. ICN '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 112–122. [Online]. Available: https://doi.org/10.1145/3357150.3357397

[17] B. Liskov and R. Ladin, "Highly available distributed services and fault-tolerant distributed garbage collection," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '86. ACM, 1986, p. 29–39. [Online]. Available: https://doi.org/10.1145/10590.10593

[18] W. Shang, "Distributed Dataset Synchronization in Named Data Networking," Ph.D. dissertation, UCLA, 2017, https://escholarship.org/uc/item/956023bx. [Online]. Available: https://escholarship.org/uc/item/956023bx

[19] W. Shang, A. Afanasyev, and L. Zhang, "VectorSync: Distributed Dataset Synchronization over Named Data Networking - Named Data Networking (NDN)," Named Data Networking, Tech. Rep., 2018. [Online]. Available: https://named-data.net/publications/techreports/ndn-0056-1-vectorsync/

[20] A. Vahdat and D. Becker, "Epidemic Routing for Partially-Connected Ad Hoc Networks," Duke University, Tech. Rep., 2000. [Online]. Available: http://issg.cs.duke.edu/epidemic/epidemic.pdf

[21] T. Li, Z. Kong, S. Mastorakis, and L. Zhang, "Distributed Dataset Synchronization in Disruptive Networks," in *16th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (IEEE MASS)*. IEEE, November 2019, p. 10.

[22] X. Xu, H. Zhang, T. Li, and L. Zhang, "Achieving resilient data availability in wireless sensor networks," in *2018 IEEE International Conference on Communications Workshops, ICC Workshops 2018 - Proceedings*, 2018, pp. 1–6.

[23] T. Li, Z. Kong, and L. Zhang, "Supporting delay tolerant networking: A comparative study of epidemic routing and NDN," in *2020 IEEE International Conference on Communications Workshops, ICC Workshops 2020 - Proceedings*, 2020, p. 6.

[24] Y. Hu, C. Serban, L. Wan, A. Afanasyev, and L. Zhang, "PLI-Sync: Prefetch Loss-Insensitive Sync for NDN Group Streaming," in *2021 IEEE International Conference on Communications*, 2021.

[25] ——, "PLI-Sync: Prefetch Loss-Insensitive Sync for NDN Group Streaming," 2020, Named Data Networking Community Meeting 2020 (NDNComm'20). [Online]. Available: https://www.nist.gov/news-events/events/2020/09/ndn-community-meeting

[26] H. B. Abraham, J. Parwatikar, J. DeHart, A. Drescher, and P. Crowley, "Decoupling information and connectivity via information-centric transport," in *ICN 2018 - Proceedings of the 5th ACM Conference on Information-Centric Networking*, 2018, pp. 54–66.

[27] H. B. Abraham, "Decoupling Information and Connectivity via Information-Centric Transport," Ph.D. dissertation, Washington University in St. Louis, 2019. [Online]. Available: https://openscholarship.wustl.edu/cgi/viewcontent.cgi?article=1513&context=eng_etds

[28] Y. Zhang, Z. Xia, A. Afanasyev, and L. Zhang, "A Note on Routing Scalability in Named Data Networking," in *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2019, pp. 1–6.

[29] 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.

[30] 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.