NDN Technical Report NDN-0056. http://named-data.net/techreports.html
Revision 1: March 18, 2018

VectorSync: Distributed Dataset Synchronization over Named Data Networking

Wentao Shang*, Alexander Afanasyev† and Lixia Zhang*
*UCLA
{wentao,lixia}@cs.ucla.edu
†Florida International University
aa@cs.fiu.edu

***Abstract*—Distributed dataset synchronization (sync for short) provides an important abstraction for multi-party data-centric communication in the Named Data Networking (NDN) architecture. Several NDN Sync protocols have been developed so far, each made its own design choices to work well under specific network conditions. This paper presents VectorSync, a new realization of the sync protocol that is built upon the lessons learned so far. In VectorSync, the shared dataset state is represented by a version vector, allowing for efficient inconsistency detection and difference reconciliation. Each communicating party maintains a complete view of the active participants through a leader-based group management mechanism, effectively bounding the size of the sync state. Our simulation-based evaluation shows that the VectorSync design improves the efficiency of dataset synchronization compared toChronoSync protocol under a wide range of network conditions and provides efficient group membership management without affecting the synchronization speed.**

Note that this report describes VectorSync protocol developed in early 2017. Since then, we have been working on a revised design to make the protocol resilient in face of intermittent connectivity, such as the case in ad hoc mobile scenarios.

I. Introduction

Named Data Networking (NDN) [1], [2] is an information-centric network architecture designed to replace the host-oriented communication model in TCP/IP. At its network layer, NDN employs a basic *Interest-Data exchange* primitive to provide best-effort retrieval of individual data objects over the network. While this simple yet powerful primitive significantly narrows the semantic gap between the application layer and the network layer, it is still cumbersome to use directly to build large-scale distributed applications, such as Web services, content sharing, big data processing, etc.

Early on in the NDN research effort, the concept of *sync* was identified as an important abstraction to simply application design for *multi-party data-centric communication*. Distributed applications running on top of sync publish and consume messages in a shared dataset. The sync protocol disseminates the knowledge of the application data and maintains a consistent state of the shared dataset across all participants in the system. Similar to the earlier works on reliable multicast in IP networks [3]–[5], NDN sync plays a role of *transport* protocol in the NDN architecture that provides eventual delivery of all the data (knowledge about all data) to all the parties in the communication group.

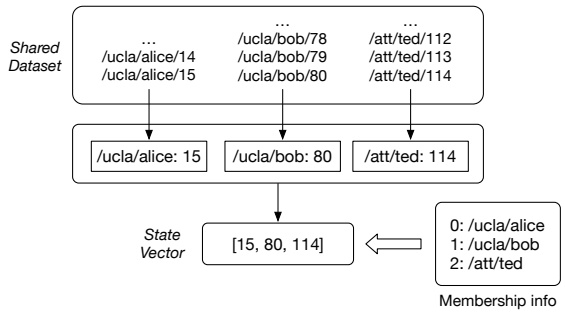

Fig. 1. Representing the dataset namespace using a state vector

In this paper we present *VectorSync*, a new sync protocol for the NDN architecture. The design of VectorSync benefited from the past experience in developing existing sync protocols such as ChronoSync [6], RoundSync [7], and PSync [8], which provides valuable insights into the tradeoffs between various design approaches. Similar to ChronoSync and several other protocols, VectorSync adopts the convention of each producer naming its data under a unique name prefix with continuous sequence numbers However, the key difference between VectorSync and its predecessors is the use of version vector [9] or state vector (hence the name of the protocol) to exchange the complete dataset state among the communicating parties (Figure 1). This enables a simple state encoding (by enumerating the latest data sequence number from each producer) and an efficient state reconciliation mechanism (through version vector comparison and *join*). Another important contribution of VectorSync protocol design is the integration of a *leader-based group membership management* (inspired by the previous works [10], [11]) that maintains a consistent view of the active participants in the application group. Managed group further enables VectorSync to support a dataset snapshot service that captures the dataset state changes over the history of the application session.

The rest of this paper is organized as follows. Section II gives an overview of NDN sync. Section III states the motivation of developing the VectorSync protocol. Section IV introduces the system model and assumptions we make in the VectorSync design. Section V describes the VectorSync protocol components in detail. Section VI presents the simulation study on the performance of VectorSync. Section VII

addresses the open issues related to the operation of the VectorSync protocol. Section VIII discusses the connection between VectorSync and the reliable multicast protocols in IP. Finally, Section IX concludes the paper and addresses future work.

II. Background

Several sync protocols have been developed for the NDN architecture to facilitate distributed applications. In this section we briefly review several sync protocols that became an inspiration for the current work (ChronoSync [6], RoundSync [7], and PSync [8]), referring to detailed description of these and other existing sync protocols to a comprehensive overview published recently [12].

In ChronoSync [6], each producer in a sync group names its data under its unique data publishing prefix using continuous sequence numbers. This allows ChronoSync to effectively reduce the hierarchical namespace to a list of (producer prefix, latest sequence number) pairs, which is referred to as the *sync state*.[1] The producers maintain multicast *"long-lived" Sync Interests* that carry the digest of their local sync state to notify others about the sender's current state and to request future updates made on top of that state. In steady state, every producer issues Sync Interest with the same digest; updates are returned as a Data packet to all members in the group via the "multicast tree" created by the pending Sync Interests. However, if multiple producers generate updates simultaneously, ChronoSync needs to either retransmit the Sync Interests with exclude filters to retrieve additional updates or fall back to a recovery mechanism.

RoundSync [7] addresses the simultaneous data publishing issue in ChronoSync by dividing the synchronization process into *rounds*: each producer may publish at most one data item in a round and must move to a new round if some other producer has already published data in that round. RoundSync also decouples the overloaded functionality of Sync Interest in the ChronoSync design by introducing a new type of message called *Data Interest* for fetching the data published in each round; the Sync Interest serves solely as a notification message triggered by state changes. However, note that if multiple producers happen to publish data in the same round, they still need to issue Data Interest with exclude filters to retrieve the simultaneous data.

PSync [8] is initially developed for the consumers to synchronize a subset of data from a single producer. The whole dataset is organized into streams where the packets under the same stream prefix are ordered by sequence numbers. The producer's data publishing state is represented by the set of latest names from different streams, and further compressed by an Invertible Bloom Filter (IBF) [13]. The consumer's subscription list is encoded by a Bloom Filter that stores the prefixes of the subscribed streams. To request new data, a consumer sends "long-lived" Interest to the producer carrying the producer's previous IBF and the consumer's subscription list. When new data (not included in the old IBF) is published, the producer responds to the consumer with its latest IBF, and the latest data names from the streams that are subscribed the consumer.

[1] If the application requires a different naming convention, it can encapsulate the application-layer data name or the whole data object inside the sequentially named data.

III. Motivation

Inspired by the previous works, the design of VectorSync represents the latest evolution in the NDN sync research. As we have described in Section II, the existing sync protocols made a few important design decisions that may affect the efficiency of the sync process under various conditions. First, both ChronoSync and PSync utilize long-lived Interest to pre-establish the return path for the sync state updates, which causes the overhead of maintaining soft-state pending Interests in the network through periodic (re)transmission of the Sync Interests. Second, both ChronoSync and RoundSync require the use of exclude filters to retrieve simultaneous updates in multiple round-trips, which leads to bloated Sync Interest size and, more importantly, increased data synchronization time. Those issues motivate VectorSync to adopt an alternative state synchronization mechanism that does not require long-lived Interest or exclude filter with the goal of improving the efficiency of the protocol.

Another key motivation of VectorSync is based on the observation that existing sync protocols do not provide the support for group membership management, which causes difficulty in removing departed producers from the protocol state maintained by each communicating party. For example, due to the lack of group membership management, the NLSR [14] routing protocol deployed on the NDN testbed experienced an operational issue that the sync protocol state was bloated with stale information as the routers were replaced and restarted after software update or system failure. Different from its predecessors, VectorSync provides a leader-based group membership management mechanism that maintains a consistent view of the group among the active participants. Managing group membership at the sync layer is necessary not only for pruning the protocol state, but also for enabling efficient data loss detection within a "closed" space of the protocol state. The group membership information can also facilitate data authentication and access control which is important for securing the sync communication (see detailed description in Section V-E).

IV. System Model

In this section we describe the system model and assumptions about the communicating parties and the networks in the design of the VectorSync protocol. We consider an application group with a finite number of active participants (also called *sync nodes*), each of which may join and leave the group at any time.[2] Each participant is assigned a data publishing prefix that is aligned with the topological prefix of the underlying

[2] Although the synchronization algorithm in VectorSync can support a group of arbitrary size, in practise we limit the size of the group so that the protocol state can be transmitted on the wire in a single Data packet.

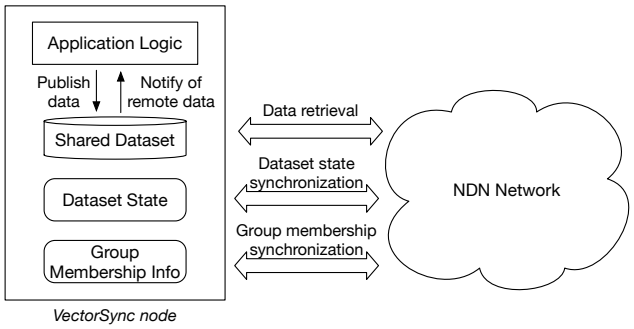

Fig. 2. VectorSync Protocol Components

(a) Application data name:
/[producer-name]/[app-name]/[seq#]

(b) Notification Interest name:
/[group-prefix]/vid/[view#]/[leader-name]/%DA/[producer-name]/[seq#]

(c) ViewInfo data name:
/[group-prefix]/vinfo/[view#]/[leader-name](/[segment#])

(d) Snapshot data name:
/[group-prefix]/snapshot/[view#]/[leader-name]

Fig. 3. VectorSync naming conventions

network and unique within the group. The underlying network is unreliable and may drop or delay packets arbitrarily during the transmission. However, we assume the participants in the same group have reachability to each other most of the time. The network may be partitioned temporarily, dividing the group into multiple subgroups, but will eventually reconnect.

The participants in an application group communicate with each other by publishing data to, and consuming data from, a shared dataset. The updates in the dataset made by one party are propagated to others over the network by the VectorSync protocol, which notifies the application about the new data using a pre-configured callback function. When multiple applications are running on the same physical node, each application creates its own group and operates independently.

Figure 2 illustrates the system components of the VectorSync protocol. Each VectorSync node maintains three important data structures:

- *Shared dataset*: the local storage of the data items published in the application group. Data published locally or received from the remote peers via VectorSync is stored in this data structure for easy access by the local application instance.
- *Dataset state*: a version vector representation of the shared dataset namespace containing all published data that the node is aware of. This data structure summaries the node's latest knowledge about the dataset state and supports efficient set difference reconciliation between two copies of the dataset.
- *Group membership list*: a list of active participants in the application group, which is referred to as the *view*, a terminology initially devised for Viewstamped Replication [10], [11] and adopted by many other distributed protocols. Each view is uniquely identified by a *view ID*, which is used for detecting inconsistent knowledge of the group membership among the participants.

V. Protocol Design

The VectorSync protocol consists of two interdependent components: the dataset state synchronization mechanism for maintaining a consistent state of the shared dataset, and the group membership synchronization mechanism for maintaining consistent knowledge about the current group membership. Both protocol components are non-blocking: a producer can publish new data at any time even if it has been disconnected to the group and/or has outdated knowledge about the group membership; the changes in the dataset and the group membership information are propagated asynchronously in the group to achieve eventual consistency.

A. Data Naming and Dataset State Representation

Fig. 3(a) shows the naming convention for the application data in the shared dataset. Each producer in the group publishes application messages under a unique data publishing prefix that also serves as the name of that producer. The "app-name" component indicates the name of the application, which is known to every party a priori. The last component in the data name is a monotonically increasing sequence number that uniquely identifies the data packets from the same producer. Applications that demand more complex data naming conventions may be supported through one level of indirection by encapsulating the application-named data object in the sequentially named data.

Since each producer's sequence number increments continuously, the state of the shared dataset is concisely summarized by a version vector [9] (called *state vector*) that contains the latest data sequence number from every producer in the group. The order of the producers in the version vector follows the group membership list, where the producers are ordered canonically based on their unique prefixes. Therefore, the producers' data prefixes can be omitted from the state vector, minimizing the cost of transmitting the full vector over the network. Figure 1 illustrates an example of the state vector representing the namespace of a dataset with three producers.

B. Dataset State Synchronization

When a producer publishes a new data packet, it sends out a notification Interest to announce the name of the new data so that others in the group can fetch the data immediately upon receiving the notification. Figure 3(b) shows the naming convention for the notification Interest name, which starts with a multicast prefix that uniquely identifies the group (and the application), and carries the producer name and the sequence number of the new data at the end. This provides enough information for the receiver to construct the new data name following the naming convention in Figure 3(a).

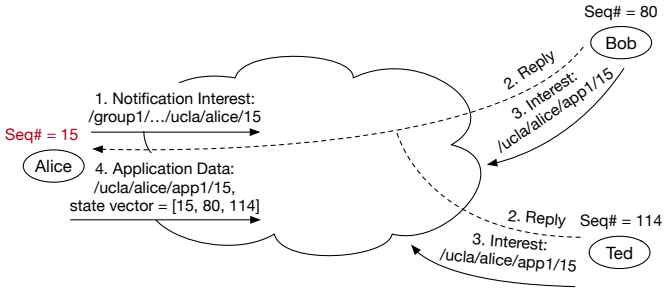

Fig. 4. VectorSync protocol message exchange in a group of three parties

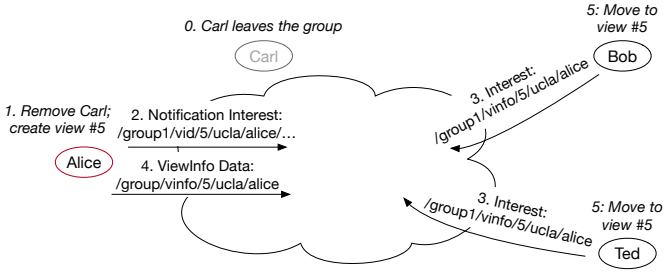

Fig. 5. View change process after removing a member

Besides the application message, the data also carries the producer's state vector at the time the data is published and the view ID representing the producer's knowledge of the group membership (which provides context for interpreting the state vector). If the receiver is in the same view as producer, it will perform a $Join$ operation that takes the entry-wise maximum of the received and local vectors:

Given $v_1 = (a_1, a_2, ..., a_n)$ and $v_2 = (b_1, b_2, ..., b_n)$, $Join(v_1, v_2) = (Max(a_1, b_1), Max(a_2, b_2), ..., Max(a_n, b_n))$

The receiver replaces its local state vector with the output of $Join$, which represents the union of the local and the remote dataset. If any entry in the new state vector contains a higher sequence number than before, the receiver will issue Interests to fetch the new data identified by the sequence numbers in between. Note that the data prefix of the producer represented by each entry is readily available in the membership list.

Upon receiving a notification Interest, the receiver also sends a reply packet to satisfy the pending notification Interests in the network and provide an acknowledgement to the data producer. The reply carries the receiver's state vector and view ID, which provides an opportunistic channel for propagating sync states to the producer.[3] Figure 4 shows an example of a new data production and dissemination process in a group of three parties. When there is no packet loss and no cached data in the network, the minimum delay for propagating the data from the producer to another party in the group is $1.5 \times RTT$ (plus processing delay).

C. Group Membership Synchronization

VectorSync utilizes a leader-based protocol to maintain a consistent *view* of the application group among the communicating parties. To synchronize the view, the sync node who has the highest-ordered prefix among the active participants declares itself as the leader and publishes its knowledge about the current view in an NDN Data packet (called *ViewInfo* packet) which is followed by others in the group. Each view is identified by a tuple called *View ID* that contains a monotonically increasing view number and the leader's data prefix (to disambiguate multiple views with the same number during group partition).

To maintain its membership, a sync node publishes periodic heartbeat packets in the shared dataset in addition to the application data in order to assert its existence. If its heartbeat is missed for more than M times (where M is an application-specified parameter), the node is considered to have left the group.[4] Like application data, the heartbeat contains the node's state vector and view ID, which enables periodic synchronization of the dataset state inside the group.

When a sync node joins or leaves the group, the leader initiates a *view change* process by incrementing the view number and publishing its latest knowledge of the group membership in a ViewInfo packet following the naming convention in Fig. 3(c). Note that the ViewInfo is named under the multicast group prefix rather than the leader's own prefix, which allows any sync node in the group to store and serve the ViewInfo packet. The ViewInfo contains the list of active participants with their data publishing prefixes and public key certificates, which essentially provides a certificate bundle signed by the leader for all the members in the view.[5] If the current leader leaves the group, the node with the second highest-ordered name immediately becomes the new leader and initiates the view change to remove the previous leader from the view.

All notification Interests carry the sync node's current view ID in the Interest name, as is shown in Figure 3(b). This allows the group participants to discover new views created in the group. Upon receiving a notification Interest with a higher view number (than its own), a sync node tries to fetch the ViewInfo corresponding to the view ID carried in the Interest name. Figure 5 illustrates the process of removing a departed member and moving the group to a new view. VectorSync essentially turns the membership management problem into a data synchronization problem by publishing the membership information as data and synchronizing the latest membership data in the group.

Before receiving the new ViewInfo, the sync node keeps publishing data in its current view. After moving to the new view, the node creates a state vector for the new view and

[3]Note that the producer will receive at most one of the replies generated by the group members, which may not reflect the latest dataset state in the group.

[4]Note that the heartbeat mechanism assumes the group participants' clocks advance at roughly the same speed, but does not require the clocks to be synchronized.

[5]The ViewInfo may be segmented if it is too large to fit into a single Data packet.

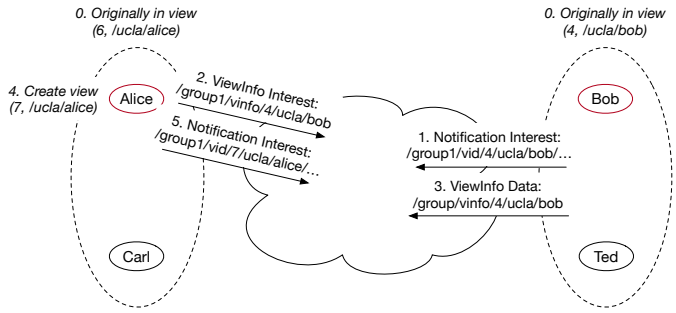

Fig. 6. Merging two sub-groups after network partition heals

fills the entries with the latest sequence numbers for the members the node already knew.[6] After that the node can start synchronizing the dataset state in the new view.

When network partition happens, each partition may select its own leader that creates a new view including a subset of group members. VectorSync allows multiple leaders, each leading a different sub-group, to co-exist in the same distributed system, and provides a simple reconciliation mechanism whereby the leader with the highest-ordered prefix among all active members takes the responsibility of merging the sub-groups into a new view after the network partition heals. Figure 6 shows an example of the view change process that merges two sub-groups. Note that VectorSync does not provide a separate group-joining mechanism. To join an existing group, a sync node first creates a single-node view with itself being the leader. Then the leader of an existing view will follow the view-merging process to add this new member into the group.

D. Handling Packet Loss

Packet loss and link failure in the network may cause the group participants to miss one or more data packets published by each other. VectorSync provides several measures for protecting the state synchronization process against packet loss. First, VectorSync requires all parties to generate reply packets when they receive notification Interests. If the data producer does not receive any reply before the notification Interest expires, it will retransmit the notification up to a pre-configured number of times. However note that the receipt of a single reply does not imply that all the other parties in the group have received the notification. This is inherently due to the multicast nature of the notification mechanism and the "one-Interest-one-Data" requirement in NDN.

Second, VectorSync requires each group participant to periodically publish heartbeat packets in the shared dataset. As long as there is no permanent failure in the network, the nodes will eventually receive some new data from each other and detect any missing data using the latest sequence number. The state vector carried in the data and the notification reply also enables periodic state reconciliation.

[6]New members are assumed to start with sequence number zero.

In addition to the built-in state synchronization mechanism, VectorSync can also benefit from link-layer loss detection and fast retransmission (if available) to achieve faster recovery from packet loss in the network, rather than waiting for new data to be published in the group.

E. Securing VectorSync Communication

VectorSync requires that the participants in the same application group share a common trust anchor and have obtained public key certificates from that trust anchor before participating in the application. After authenticating the ViewInfo published by the leader, a node can directly use the public keys in the ViewInfo packet to authenticate the data published by other parties, including application data, heartbeat, notification reply, etc. This prevents malicious nodes from publishing invalid ViewInfo (which may contain unauthorized nodes as members) or application data with invalid state vector under a legitimate group member's name. Note that an attacker can still send notification Interests containing arbitrary sequence numbers, in which case the legitimate nodes will ignore those sequence numbers since no corresponding data can be retrieved. If necessary, such attack can be prevented by requiring the producer to sign the notification Interest so that others can authenticate the notification before fetching the new data.

Access control can also be achieved by leveraging the group membership information, similar to the solution in NDN-ACT [15]. The leader may periodically generate a symmetric data encryption key and distribute the key to every party on the current membership list [16]. The producers encrypt the application messages using the latest encryption key, which can be decrypted only by the members in the same view.

F. Dataset Snapshot

An important feature in the VectorSync design is that the departed members are removed from the state vector after the view change, preventing the size of the vector from growing unbounded as the sync nodes come and leave in a long-running application session. As a result, the new members that join the group late will not learn about the previous data published by the departed members (unless they rejoin the group later). However, in some application scenario it may be useful to preserve the historical data, including that published by the departed nodes, so that new members joining the application can still discover and retrieve the old data they need. This can be supported by providing a distributed *dataset snapshot* service on top of VectorSync to capture the dataset state changes over the history of the group.

The snapshot process starts after each view change in the group. After joining a new view, each group member publishes its local state vector of the previous view (together with the view ID) in a data packet called *local snapshot*. The local snapshot packets are published in the shared dataset and propagated to the leader by VectorSync. The leader is responsible for collecting local snapshots from all members and aggregating the reported state vectors via the *Join* operation to summarize

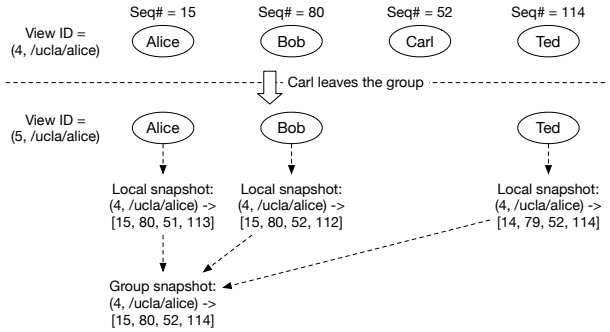

Fig. 7. Example of generating a group snapshot after a view change

the group-wide knowledge of the dataset state at the end of the previous view. Finally, the leader publishes the "joined" state vector and the corresponding view ID in a data packet called *group snapshot* and notifies the members in the current view, who may use that information to detect missing data in the previous view. Figure 7 illustrates the snapshot process with a simple example. To permanently store the data, the dataset snapshot service requires a stable storage node (e.g., a repo) to collect and store the group snapshots and the corresponding data packets.

There are several important details worth discussing in the design of the snapshot mechanism. First, the group snapshot data is named under the current view ID following the naming convention in Fig. 3(d), and the content carries the previous view ID. Therefore the group snapshot packets published over time essentially form a chain of successive view IDs, which can be used to trace the history of view changes. Second, when group partition happens, each sub-group will generate a (partial) snapshot for the previous view. To fully recover the dataset state for that view, one would need to trace the branches in the view change history to obtain all group snapshots. Third, after the group partition heals, the members that belonged to different subgroups will publish local snapshots for different views. In that case, the group snapshot will contain multiple state vectors, one for each unique view reported by the current members. Finally, calculating the group snapshot requires collecting state vectors from all members in the current view. If one member in the group crashes before it publishes any local snapshot, the leader will not be able to generate a group snapshot until it removes the departed member and creates a new view. As a result, the group members need to retain the state vectors from all previous views not covered by any group snapshot and report all of them in their local snapshots.

VI. Evaluation

We implemented a prototype VectorSync module and conducted simulation-based performance evaluation using the latest version of the ndnSIM [17] simulator. The network topology we use for the simulation is generated from the measurement result [18] of a real-world ISP network (shown in Fig. 8) with 176 nodes and 289 links, among which 10

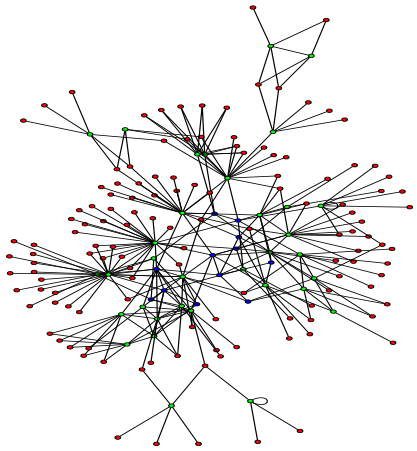

Fig. 8. The simulation network topology

nodes are randomly selected to participate in a VectorSync group. Each selected node runs an application that publishes a total of 100 data packets in a shared dataset following the Poisson process with average inter-arrival time of 10 seconds. To simulate packet loss, we configure *all* nodes in the network to randomly drop the received packets at a pre-configured error rate.

We focus on two important performance metrics in the evaluation:

- Data synchronization delay: the time needed for published data to be received by *all* parties in the group;
- Network traffic volume: the amount of Interest and Data packets transmitted in the network during the experiment.

A. Comparison with ChronoSync

In this subsection we compare the performance of VectorSync and ChronoSync under different packet loss rates using a stable group (i.e., with no membership change throughout the experiment). The main reason of choosing ChronoSync for comparison is that it is currently the only sync protocol with mature published implementation[7] used by many NDN applications. As we have described in Section II, ChronoSync propagates new data name in reply to the pending Sync Interests and notifies the other nodes to fetch the new data, achieving a minimum propagation time of $1.5\times RTT$. However, if multiple nodes publish data simultaneously, ChronoSync may experience longer synchronization delay because the nodes have to spend extra round trips to retrieve additional Sync Replies using exclude filter. In addition, when the state of the group has already diverged, ChronoSync relies on the recovery mechanism to retrieve the full sync state that corresponds to some "unrecognized" state digest. This will cause additional delay and higher traffic in the network (because the recovery Interest is forwarded to all nodes in the group via multicast).

[7]https://github.com/named-data/ChronoSync

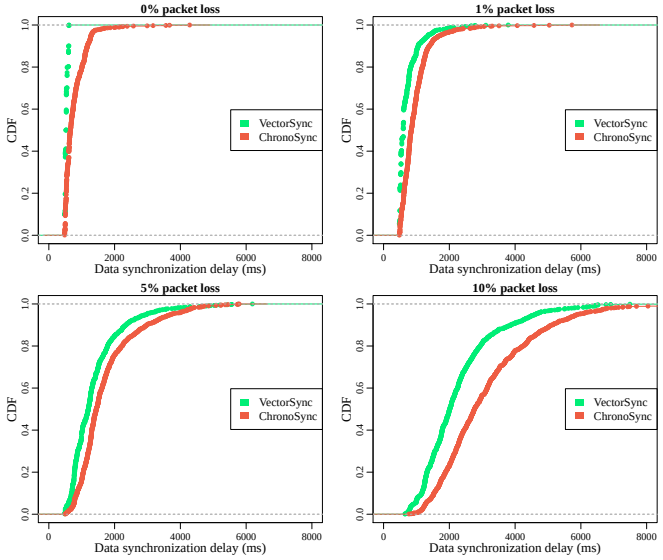

Fig. 9. Data synchronization delay: VectorSync vs. ChronoSync

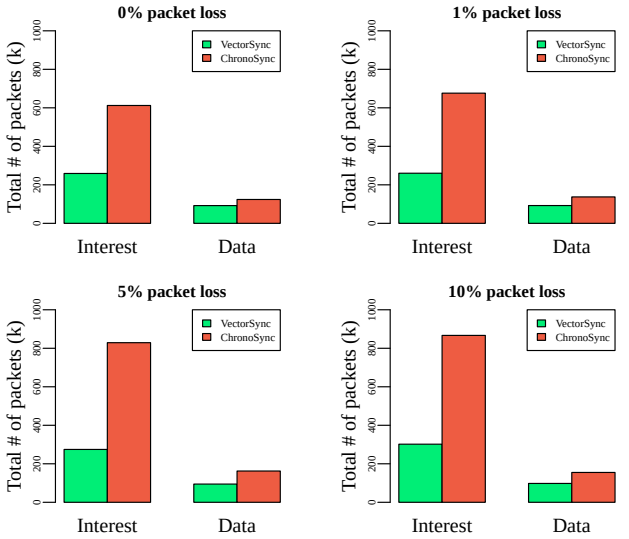

Fig. 10. Total number of packets transmitted in the network: VectorSync vs. ChronoSync

Figure 9 shows the CDF plots of the data synchronization delay of VectorSync and ChronoSync under different packet loss rates. We can see that even when there is no packet loss in the network, ChronoSync nodes still experience significantly longer synchronization delay for about 40% of the published data. This is due to simultaneous data publishing and state divergence that cause ChronoSync nodes to spend extra round-trips to fetch simultaneous sync replies or even invoke the recovery mechanism. On the other hand, VectorSync is resilient to simultaneous data publishing because the notification Interest carries explicit information about the new data name (instead of a state digest), which allows receiving nodes to fetch the new data immediately after receiving the notification. When the packet loss rate increases, both sync protocols experience longer synchronization delay but VectorSync still performs better than ChronoSync.

We also measure the total number of Interest and Data packets transmitted over the network during the experiment, which is shown in Figure 10. Compared to VectorSync, ChronoSync generates much higher volume of Interest packets because ChronoSync nodes need to send additional multicast Sync Interests with exclude filter to detect simultaneous updates every time they receive a Sync Reply. The recovery Interests for repairing diverged states also contribute to the high number of Interests. On the other hand, in VectorSync the Interest traffic volume grows only slightly as the packet loss rate increases, due to the retransmission of expired Interests.

B. Dynamic Membership Changes

In this subsection, we study the impact of dynamic group membership changes on the performance of VectorSync. We set up the simulation scenario where each node leaves the group (by shutting down the data publishing application and the VectorSync module) at a randomly selected time point,

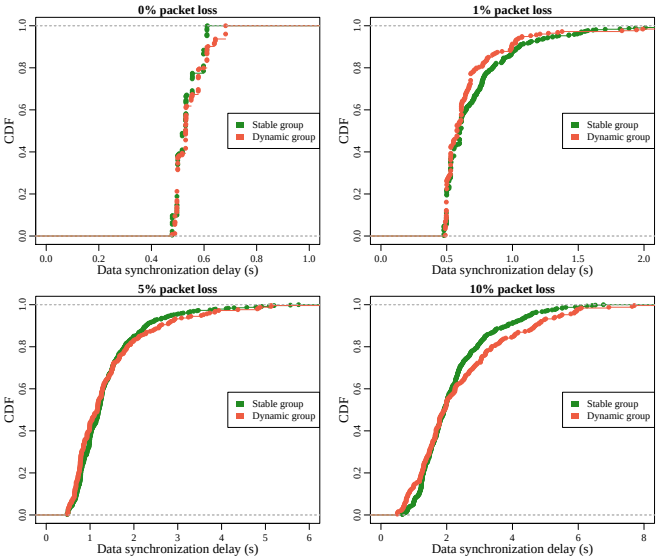

Fig. 11. Data synchronization delay with and without dynamic membership changes

and compare the data synchronization delay to the previous results under the scenario with no membership changes.

Figure 11 shows the CDF plots of the data synchronization delay under different packet loss rates with and without dynamic membership changes. As we can see, the data synchronization delay is mostly unaffected by the view change process when the packet loss rate is below 10%. This is mainly due to three important design properties of the VectorSync protocol. First, VectorSync decouples view change from dataset synchronization: nodes can always fetch the new data based on the explicit information carried in the notification Interest even if their membership knowledge is not synchronized with the data producer yet. Second, VectorSync

synchronizes the membership information by publishing the ViewInfo and announcing the view ID in every notification Interest, which enables the nodes to synchronize their views quickly after the membership changes. Third, VectorSync utilizes a deterministic leader selection algorithm that allows the group to pick a new leader as soon as the current one leaves, which also improves the view synchronization speed.

The main reason for the noticeable increase in the data synchronization delay under 10% packet loss is because the overall data rate in the group gradually decreases as the nodes leave the group over time; consequently, the remaining nodes have to wait longer for the next application data or heartbeat packet to provide updated state vector in order to detect missing packets. Note that in the scenario with no packet loss, the maximum data synchronization delay under dynamic membership changes is longer than that without membership changes. This is because the benefit of in-network caching diminishes when there are less number of nodes in the network (near the end of the experiment) to fetch the published data.

VII. Discussion

In this section we discuss a few open issues that are usually outside the scope of sync protocol design but closely related to the sync protocol operations. While our discussion focuses on VectorSync, the proposed solutions can be applied to other sync protocols as well.

A. Group Rendezvous

The group communication in NDN sync requires an efficient rendezvous mechanism through which the distributed nodes can reach each other. All existing sync protocols, including VectorSync, assume the availability of network-layer multicast so that the participants in a distributed application can easily send multicast Interest packets to all the others at once. However, deploying network-layer multicast over wide-area Internet is proven to be difficult in practice. Moreover, there are a number of application scenarios where the network-layer multicast is either infeasible or prohibitively expensive (e.g., ad hoc environments). One solution to the scalable multicast communication problem is to establish some (virtual) topology among the communicating parties, e.g., using *Distributed Hash Table* (DHT) [19], over which the multicast Interests can be propagated. Another solution currently under our investigation is to utilize a *multicast overlay* that contains a number of dedicated rendezvous points in the network. Those rendezvous points are responsible for collecting and delivering the multicast Interests via the overlay to every participant. A third alternative is to adopt the viral propagation (also called epidemic dissemination) [20] model, where a node disseminates a message to a subset of its neighbors and those neighbors further propagate the message until all nodes in the group have received it. This communication model is particularly suitable for infrastructure-less and ad hoc network environments.

B. Storage Scalability

As the participants in the sync group continue to generate new data, the size of the shared dataset may exceed the storage size of any individual node. If the application requires permanent storage of all historical data published in the group, the whole dataset has to be *sharded* across multiple nodes for storage scalability. One way to achieve that is to build a data sharding service on top of VectorSync using consistent hashing [21] or DHT techniques to distribute the data objects among the current members in the group. When a sync node receives notification of a new data packet, it consults the data sharding service to decide whether it is responsible for storing that data. If it is, the node will subsequently fetch the data and store a local copy of it; otherwise, it simply updates the local sync state but does not fetch the data. Note that the nodes are still able to sync up with each other even if each of them maintains only a subset of the shared data because the sync state is solely based on the namespace of the dataset.

VIII. Related Works

As an abstraction for multi-party communication, NDN sync is closely related to the reliable multicast protocols in the TCP/IP architecture such as RMTP [5], PGM [4], and SRM [3], with a lot of commonalities in the protocol design approaches. Similar to VectorSync (and its predecessors), those reliable multicast protocols also utilize sequence numbers for naming individual data pieces from the senders and detecting packet losses at the receivers. They also provide periodic messages (driven by either senders or receivers) to ensure eventual delivery of all data in the multicast group. In particular, the SRM protocol resembles VectorSync on several important design aspects. First, each member in SRM multicasts periodic *session messages* that carry the highest sequence number from the active data sources for loss detection, and allow the participants to determine the group membership information. The session message essentially provides the equivalent functionality as the heartbeat message does in VectorSync. Second, SRM introduces the concept of *pages* to partition the state of a large group. Each page is identified by the page initiator's ID and the page number. Each member may participate on any page and report only the state of its current page in the session messages. This is analogous to how VectorSync manages the group membership through views and synchronizes the dataset state among the active members in each view.

The fundamental difference between NDN sync and the reliable multicast protocols in IP is that the NDN sync protocols can benefit from the data-centric network architecture to improve the efficiency in the group communication. For example, NDN sync decouples the synchronization of the dataset namespace from the retrieval of actual data, thanks to the unique and secured binding between names and immutable data objects in NDN. When disseminating the data to multiple nodes in the group, the sync protocols can utilize NDN's built-in data multicast delivery capability that automatically suppresses duplicate Interests and Data with flow balancing,

maintains the multicast data delivery path via the Pending Interest Tables, and caches the data in the network to satisfy future requests. Those built-in data-centric communication features greatly simplify the design of the NDN sync protocols. On the other hand, the IP-based reliable multicast protocols have to introduce complex mechanisms for duplicate suppression, packet aggregation, multicast tree maintenance, etc., which essentially implements many of the NDN features on top of IP networks. Besides, none of them considered the data security issue, leaving it to the application layer to address.

IX. Conclusion

This paper presents VectorSync, a new sync protocol for the NDN architecture that provides reliable multi-party communication with built-in group membership management. Managing the group membership at the sync layer enables VectorSync to utilize a concise vector representation for the dataset state and eventually prune the departed members from the state. It also facilitates data authentication and access control in the sync communication. The simulation study shows that VectorSync is able to improve the efficiency of sync across different network environments compared to ChronoSync and achieve efficient group membership management without affecting the dataset synchronization performance.

Our work on the design and implementation of VectorSync is still preliminary and there are many opportunities for future research. First, the protocol may benefit from different sync state representations, such as compressing the version vector with Invertible Bloom Filter for large groups. Second, it is interesting to investigate how VectorSync can adapt to the application scenarios where it may not be necessary or feasible to closely manage the group membership (e.g., wireless ad hoc network environments). Finally, as discussed in Section VII, group rendezvous and storage scalability are pratical issues that need to be addressed. Given the importance of sync in the NDN architecture, we would like to call on the ICN community to join the effort of advancing this exciting research area.

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] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 6, Dec. 1997.

[4] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. L. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano, "PGM Reliable Transport Protocol Specification," RFC 3208, Dec. 2001.

[5] J. C. Lin and S. Paul, "RMTP: a Reliable Multicast Transport Protocol," in *Proceedings of IEEE INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation.*, vol. 3, Mar 1996.

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

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

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

[9] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 240–247, May 1983.

[10] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," in *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1988, pp. 8–17.

[11] B. Liskov and J. Cowling, "Viewstamped Replication Revisited," http://pmg.csail.mit.edu/papers/vr-revisited.pdf, 2012.

[12] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "Overview of Distributed Dataset Synchronization in Named-Data Networking," NDN Project, Technical Report NDN-0053, April 2017.

[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] A. K. M. M. H. Vince Lehman, Y. Yu, L. Wang, B. Zhang, and L. Zhang, "A Secure Link State Routing Protocol for NDN," NDN Project, Technical Report NDN-0037, January 2016.

[15] Z. Zhu, S. Wang, X. Yang, V. Jacobson, and L. Zhang, "ACT: Audio Conference Tool over Named Data Networking," in *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking (ICN)*, 2011, pp. 68–73.

[16] Y. Yu, A. Afanasyev, and L. Zhang, "Name-Based Access Control," NDN Project, Tech. Rep. NDN-0034, Revision 2, Jan. 2016.

[17] S. Mastorakis, A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM 2: An updated NDN simulator for NS-3," NDN Project, Technical Report NDN-0028, Revision 2, November 2016.

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

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

[20] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1987, pp. 1–12.

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