UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Dataset Synchronization in Named Data Networking

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Wentao Shang

2017

ABSTRACT OF THE DISSERTATION

Distributed Dataset Synchronization in Named Data Networking

by

Wentao Shang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017
Professor Lixia Zhang, Chair

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 takes a different design approach than the others. They all enable a group of distributed nodes to publish to, and consume data from, a shared dataset and maintain a consistent state about the dataset among the participants. However, each of them has its own issue in the protocol design that causes inefficiency in the dataset synchronization. In addition, none of them provides built-in membership management support, making it difficult to remove departed nodes from the sync protocol state (maintained by every node in the group). As a result, existing applications running on top of sync have to implement group management by themselves, either at the application layer or by extending the underlying sync protocol.

In this dissertation we first present a comparative study on the design of existing sync protocols. Our analysis focuses on the data naming convention, the representation of dataset namespace, and the state synchronization mechanism. Through the side-by-side comparison on those key design aspects, we identify common design patterns in the existing protocols and articulate their design issues.

Based on the lessons learned from the comparative study, we design a new sync protocol called VectorSync that addresses the issues in the existing works and enables new functions. In VectorSync, the state of the shared dataset namespace is concisely represented by version

vector, which allows the sync nodes to detect and reconcile inconsistencies efficiently. Every node maintains a consistent view of the current group members through a leader-based membership synchronization mechanism, which also provides the basis for data authentication and access control. Our simulation-based evaluation shows that the VectorSync design improves the efficiency of dataset synchronization compared to the widely used ChronoSync protocol under various network conditions and provides efficient group membership management without affecting the synchronization speed.

At the end of this dissertation we present the design of a few pilot applications in the Internet of Things (IoT) area that utilize different NDN sync protocols to enable important functions that are often difficult to achieve under the current TCP/IP-based IoT architecture. These new applications demonstrate the importance of NDN sync and illustrate the unique sync-based application design patterns that arise from NDN's data-centric communication model.

The dissertation of Wentao Shang is approved.

Jeffrey A. Burke

Yifang Zhu

Todd D Millstein

Mario Gerla

Lixia Zhang, Committee Chair

University of California, Los Angeles

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to gratefully and sincerely thank my academic advisor Prof. Lixia Zhang for providing invaluable support throughout my Ph.D. program. I also want to thank my colleagues from UCLA Internet Research Laboratory Alexander Afanasyev, Yingdi Yu, Ilya Moiseenko, and others for their support and technical discussions, which provides valuable insights into this work. I am also enormously grateful to our collaborators in the NDN team Prof. Jeffrey Burke, Prof. Beichuan Zhang (University of Arizona), and Prof. Lan Wang (University of Memphis) for their guidance and support for my research. Finally, I would like to gratefully thank my parents and my wife Yuhan Wu for their unconditional support throughout my Ph.D. study.

VITA

2009	B.S. (Electronic Engineering), Tsinghua University, Beijing, China.
2012	M.Eng. (Electronic Engineering), Tsinghua University, Beijing, China.
2014–2017	Teaching Assistant, Computer Science Department, UCLA.
2012–2017	Graduate Student Researcher, Computer Science Department, UCLA.
2014	PhD Student Intern, Cisco Systems, Cambridge, Massachusetts.
2015	Software Engineering Intern, Google, Mountain View, California.
2016	Software Engineering Intern, Google, Mountain View, California.

PUBLICATIONS

W. Shang, J. Thompson, M. Cherkaoui, J. Burke, L. Zhang, "NDN.JS: A JavaScript Client Library for Named Data Networking," *Proc.of INFOCOM Workshop*, 2013.

W. Shang, Z. Wen, Q. Ding, A. Afanasyev, L Zhang, "NDNFS: An NDN-friendly File System," *NDN Technical Report NDN-0027*, 2014

W. Shang, Q. Ding, A. Marianantoni, J. Burke, L. Zhang, "Securing Building Management Systems Using Named Data Networking," *IEEE Network*, Vol. 28, Issue 3, 2014

W. Shang, Y. Yu, T. Liang, B. Zhang, L. Zhang, "NDN-ACE: Access Control for Constrained Environments over Named Data Networking," *NDN Technical Report NDN-0036*, 2015

W. Shang, Y. Yu, R. Droms, L. Zhang, "Challenges in IoT Networking via TCP/IP Architecture," *NDN Technical Report NDN-0038*, 2016

W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, et al., "Named Data Networking of Things," *Proc. of IoTDI'16*, 2016

W. Shang, A. Afanasyev, L. Zhang, "The Design and Implementation of the NDN Protocol Stack for RIOT-OS," *Proc. of GLOBECOM Workshop*, 2016

W. Shang, J. Thompson, J. Burke, "MicroForwarder.js: an NDN Forwarder Extension for Web Browsers," *Proc. of Sigcomm ICN'16*, 2016

W. Shang, Z. Wang, A. Afanasyev, J. Burke, L. Zhang, "Breaking out of the Cloud: Local Trust Management and Rendezvous in Named Data Networking of Things," *Proc. of IoTDI'17*, 2017

P. de-las-Heras-Quirós, E. M. Castro, W. Shang, Y. Yu, S. Mastorakis, A. Afanasyev, et al., "The Design of RoundSync Protocol," *NDN Technical Report NDN-0048*, 2017

W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "A Survey of Distributed Dataset Synchronization in Named Data Networking," *NDN Technical Report NDN–0053*, 2017

CHAPTER 1

Introduction

Named Data Networking (NDN) [JST09, ZAB14] is an *information-centric* network (ICN) architecture designed to replace the host-oriented communication model in TCP/IP with a data-centric one. At its network layer, NDN employs a basic *Interest-Data exchange* mechanism to provide best-effort name-based retrieval of individual data objects over the network. This simple yet powerful communication primitive enables the network layer to forward the requests for data (called *Interest* packets) based on the application-layer data names and provide pervasive in-network caching for the application data in the NDN forwarders. At the application layer, the NDN names express the trust relations between the data and the signing keys and allow consumers to authenticate the data packets according to the application-defined trust policy [YAC15] regardless of where the packets are retrieved from. While the Interest-Data exchange primitive has significantly narrowed the semantic gap between the application layer and the network layer, it is cumbersome to use directly to build large-scale distributed applications (e.g., Web services, content sharing, big data processing, etc.).

Early on in the NDN research effort, *sync* was identified as an important abstraction for multi-party communication, that can be built on top of NDN's network layer primitives, to simplify the development of distributed application in a data-centric network architecture. Applications running on top of sync can publish and consume messages in the local copy of a shared dataset that is synchronized across a group of distributed nodes. The sync protocol disseminates the knowledge of the application data and maintains a consistent state of the shared dataset across all nodes participating in the same group. This is typically achieved by synchronizing the *namespace* of the dataset, thanks to the unique and secured binding

between names and immutable data objects in NDN. After learning the new data names via sync, the applications can decide whether to fetch the new data according to their own semantics. When multiple nodes request the same data, the data can be delivered efficiently using NDN's built-in multicast data delivery and cached in the network to satisfy future requests. Sync is essentially playing a *transport* layer role in the NDN architecture that bridges the gap between the functionality required by the distributed applications and the semantics offered by the network-layer primitives, similar to the role TCP played in bridging the gap between applications' need for reliable data delivery and IP's datagram service.

In this dissertation, we first present a comparative study of existing sync protocols including CCNx 0.8 Sync [Pro12a], iSync [FBC15], CCNx 1.0 Sync [Mos14], ChronoSync [ZA13], RoundSync [HCS17], and PSync [ZLW17]. While they all implement distributed dataset synchronization service, those sync protocols differ in several key design aspects such as how to name the data in the shared dataset, how to represent the state of the dataset, and how to synchronize the dataset state via NDN's Interest-Data exchange mechanism. Our comparative study highlights those differences and the tradeoffs behind different design choices. Through the side-by-side comparison we also extract a few common design patterns shared by the existing works and identify the issues in the existing designs that affect the efficiency of the synchronization process in various aspects.

Many distributed applications require explicit group membership management so that the nodes can keep track of the current active participants in the group. However, existing sync protocols do not provide group membership management support. As a result, the existing sync protocols cannot easily remove departed nodes from the dataset state, causing the size of the state representation to grow unbounded over time. Applications that demand group membership management have to implement them either as part of the application protocol or by extending the underlying sync protocol.

Motivated by the above observation, we design and implement a new NDN sync protocol called *VectorSync* to enable new functions not offered by the existing works. The design of VectorSync is informed by the past experience in developing the existing sync protocols, which provides valuable insights into the tradeoffs between various design choices. Similar to

ChronoSync, RoundSync, and PSync, VectorSync adopts the naming convention that each sync node name its data under its data publishing prefix with continuous sequence numbers. This enables VectorSync to represent the state of the shared dataset namespace efficiently using a *version vector* [PPR83][1] that contains the latest data sequence number from each producer in the group. When new data is published, VectorSync notifies the group members about the name of the new data so that others can fetch the data immediately upon receiving the notification. The data carries the state of the producer at the time it is published, which allows receiving nodes to detect and reconcile inconsistency in the dataset using version vector operations.

The key difference between VectorSync and its predecessors is the integration of the *leader-based group membership management* mechanism that synchronizes the view of the current group membership among the active participants while the nodes join and leave the group over time. By maintaining a consistent group membership list at each node, VectorSync is able to compact the version vector in to an array of sequence numbers where the nodes are ordered by their data publishing prefixes. The group membership information also includes the members' security credentials, which can facilitate data authentication and access control. Synchronizing with managed group also allows VectorSync to support useful services such as dataset state snapshot and data total ordering.

At the end of this dissertation we describe a few new applications in the Internet of Things (IoT) area that utilize NDN sync to achieve important functions. IoT networks often comprise constrained sensor and actuator devices communicating over intermittent wireless channels, which creates significant challenges for network communication [SYD16]. The NDN architecture enables an efficient, secure, and robust solution to IoT networking by leveraging name-based data retrieval, data multicast delivery, and in-network data caching that are all built into NDN's network layer primitives [SBL16]. On top of the data-centric architecture, NDN sync can support distributed IoT applications and services that are either infeasible or inefficient to achieve in the TCP/IP-based IoT systems. Through the exercise

[1]Hence the name of the protocol.

of developing those applications, we further demonstrate the usefulness of NDN sync in simplifying application design and improving data communication efficiency.

The rest of this dissertation is organized as follows. Chapter 2 gives an overview of the NDN architecture and the distributed dataset synchronization problem in NDN. In Chapter 3 we present our comparative study of the existing sync protocols proposed for the NDN/ICN architecture. In Chapter 4 we describe the design of the VectorSync protocol and present the simulation study on the performance of this new protocol in various network scenarios. Chapter 5 describes the design of a few sync-related services that can be implemented on top of VectorSync to provide useful functions for the distributed applications. Chapter 6 presents the design of three new applications that leverage NDN sync in the IoT environments. Finally, Chapter 7 concludes this dissertation.

CHAPTER 2

Background

In this chapter we give an overview of the Named Data Networking (NDN) architecture and a general introduction to the distributed dataset synchronization (or sync) problem in NDN.

2.1 Named Data Networking Overview

Named Data Networking (NDN) [JST09, ZAB14] is a recently proposed future Internet architecture that shifts the network communication model from *host-centric* as in today's TCP/IP to *data-centric*. NDN replaces host-addressed IP packets with named and secured data objects as the new narrow waist of the "hourglass" protocol stack. At the network layer, NDN defines two types of packets: *Interest* and *Data*. Each Data packet is uniquely identified by a hierarchical name which is also used by the applications to express application-layer semantics of the data. To retrieve a piece of data, a consumer sends an Interest packet that carries the name or name prefix of the desired data. It is the underlying NDN network's responsibility to find the data whose name matches the Interest name (i.e., the Interest name is a prefix of the Data name) and return it to the consumer. This is fundamentally different from today's TCP/IP architecture where the network is responsible only for setting up connections between two end-hosts and the application data is carried over the established connections.

NDN forwarders in the network forward the Interest packets based on the Interest name. When a forwarder receives an Interest, it first looks up the Interest name using longest prefix matching in the *Content Store* (CS), a.k.a., the data cache, to see if a matching Data packet can be found in the local cache. If a matching Data packet is found, it is

returned immediately to the incoming interface of the Interest. Otherwise, the forwarder checks the local *Pending Interes Table* (PIT) to see if the Interest with the same name has been forwarded before. If a PIT entry with exactly the same name is found, the forwarder records the incoming interface in the existing PIT entry and drops the received Interest, aggregating multiple pending requests for the same data. Otherwise, the forwarder creates a new PIT entry for that Interest and records the incoming interface. Then it looks up the Interest name in the *Forwarding Information Base* (FIB) using longest prefix matching to find a set of next-hop interfaces that guide the Interest towards the potential locations where the data can found. Finally, the forwarder executes the *forwarding strategy* defined for the namespace that covers the Interest name, which may forward the Interest to all, or a subset, of the next-hop interfaces, or delay the Interest for certain amount of time based on the current network condition (e.g., congestion level and packet loss rate). This *stateful* Interest forwarding plane [YAM13] enables the NDN network to make smarter forwarding decisions compared to the *stateless* IP forwarders.

When a matching Data packet is encountered, either in the forwarder's cache or in the local storage of the data producer, the Data packet is returned to the consumer by following the reverse forwarding path of the Interest packet as recorded by the PIT entries in the forwarders. The forwarders along the path also keep a local copy of the returned Data packet in their own cache, which can be used to satisfy future requests for the same data. Each forwarder maintains its local cache independently based on some cache management policy (e.g., Least Recently Used (LRU), or popularity-based). This opportunistic in-network data caching mechanism significantly improves the efficiency of disseminating popular contents: when multiple consumers in a distributed system request the same content, only the first request needs to be forwarded towards the producer, while later requests can be satisfied immediately from the forwarder cache.

The NDN architecture provides inherent data-centric security support. The binding between the data and its name is secured by a cryptographic signature created by the data producer. Each Data packet carries the *KeyLocator* field that identifies the data signing key, which is another piece of content and can be retrieved over the NDN network in the form of

a key certificate. The consumer verifies the signature in the Data packet according to a pre-defined application-layer trust policy that specifies the relationship between the data name and the signing key name [YAC15]. To bootstrap the trust relationship, the trust policy also includes one or more *trust anchors* whose public keys are trusted by the consumers a priori. To authenticate the received data, the consumer follows the application trust policy to verify the signatures carried in the Data packet and, if necessary, walk up the certificate chain of the signing keys recursively until it reaches one of the trust anchors. To enforce access control, the producer can encrypt the content of the Data packets and distribute the decryption keys only to the authorized consumers [YAZ16]. The encrypted data can also be cached in the network to satisfy the Interests from the consumers who share the data decryption key. Compared to the channel-based security model in TCP/IP (e.g., using TLS or IPsec), the data-centric security model in NDN protects the content both in transit and at rest without requiring pair-wise secured channels between the communicating parties, which is particularly beneficial for improving the communication scalability in distributed systems.

2.2 Sync: Efficient Multi-party Communication in NDN

Today's Internet applications are typically built around large-scale distributed systems that require efficient support for multi-party communication. Popular examples of distributed applications today include file sharing, collaborative editing, group messaging and conferencing, and the fast-growing Internet of Things (IoT) which has become increasingly popular over the past few years. In NDN's data-centric communication paradigm, the multi-party communication problem becomes a distributed data synchronization (sync for short) problem where multiple participants in an application publish data in a shared dataset and consume data published by others from that dataset. NDN sync provides an important layer of abstraction to support distributed applications and services on top of NDN's Interest-Data exchange primitives. All participants in a distributed application join a *sync group* that operates on a shared dataset. When a node in the group publishes new data in the dataset, the data will be propagated to all the other nodes in the group via the sync protocol so

that the application instances running on those nodes will be notified of the new data. NDN sync greatly simplifies the design and development of distributed applications which can publish and consume data in the local copy of the shared dataset that is kept up-to-date by the underlying sync protocol without worrying about how to get the latest data from other participants in the group.

A number of sync protocols have been developed since the start of the NDN project. In Chapter 3 we will give a thorough analysis on the design of the existing sync protocols. There are also a number of applications developed by the NDN team that depend on sync to achieve critical functionality. Below is a non-exhaustive list of the existing sync-based applications:

- **CCNx repo** [Pro12b]: the earliest NDN/CCN application that uses sync to replicate the data across the repos that manage the same data collection;
- **ChronoShare** [AZY15]: a distributed file sharing application that uses sync to propagate the changes in the local file system to other peers managing the same shared folder;
- **ChronoChat** [ZBA12]: a server-less group chat application that uses sync to distribute chat messages among the users in a chat room;
- **NLSR** [VYW16]: a link-state routing protocol for NDN that uses sync to distribute routing announcements among the NDN routers;
- **NDN-RTC** [NDN17]: a group conferencing software that uses the sync channel to discover peers in the conference automatically and deliver chat messages (like in ChronoChat);
- **Federated catalog** [FSD15]: a distributed catalog for scientific data that uses sync to maintain consistency among the distributed catalog instances.

CHAPTER 3

Comparative Study of Existing NDN Sync Protocols

In this chapter, we examine the set of existing sync protocols that have been developed for the NDN/CCN architecture, including CCNx 0.8 Sync [Pro12a], iSync [FBC15], CCNx 1.0 Sync [Mos14], ChronoSync [ZA13], RoundSync [HCS17], and PSync [ZLW17]. Our goal is to extract common design patterns for NDN sync protocols and identify different design choices and tradeoffs made in different protocols. Through this analysis we hope to offer insights for the design of new sync protocols in the future.

3.1 Overview

The NDN sync protocols typically follow a common design framework. To synchronize a shared dataset, the sync protocol needs to first generate a concise summary of the dataset namespace that can be communicated efficiently between the sync nodes. The sync nodes then exchange this summary periodically via the sync protocol in order to detect and reconcile any inconsistency in the state of the dataset. To speed up the synchronization process, the sync protocol may optionally provide a quick notification of changes when a node publishes new data in the shared dataset. Note that the periodic summary exchange and the event-driven notification are not exclusive to each other and may be employed together in the sync protocol.

Our analysis focuses on the following key design aspects:

Data naming Thanks to the unique and secured binding between names and immutable data object in NDN, a shared dataset can be uniquely identified by the namespace con-

taining the hierarchical names of all data packets in the dataset.[1] Therefore the dataset synchronization problem in NDN is conveniently reduced to the synchronization of the corresponding namespace. The sync protocol may directly synchronize the application data names with arbitrary structure, or leverage naming convention to simplify the dataset namespace (and encapsulate the application data names if necessary).

Namespace representation The data structure that represents the state of the shared dataset namespace is often referred to as the *sync state*. Every sync node keeps a local copy of the sync state and uses the sync protocol to keep up with the changes generated by other nodes in the sync group. This requires the sync state to encode the namespace without loss of information and allow sync nodes to detect and reconcile the differences in the shared namespace between distinct states. The sync protocol may simply enumerate all names in the namespace (which is inefficient to be communicated over the network), or apply various compression techniques such as one-way hashing and Invertible Bloom Filter (IBF) [EGU11].

State sync mechanism Each node participating in a sync group may publish new data at any time. The sync protocol needs to ensure that other nodes in the group can eventually receive the new data and reach agreement on the state of the dataset. The state synchronization mechanism therefore should enable the nodes to (1) learn about the updates as soon as possible and (2) detect and reconcile inconsistency in the sync state caused by other factors such as packet loss and/or network partition. The synchronization process can be either notification-driven (i.e., nodes inform others when they publish new data), or based on periodic exchange of the state summary. The notification and the state summary can be carried in the Interest packets sent by the publishing node, or retrieved as Data packets.

To support dataset synchronization inside a group, the sync protocol also requires a group communication namespace for the sync nodes to publish and exchange protocol messages.

[1]The NDN sync protocols generally assume well-behaved applications that do not reuse the same name for different data objects.

Table 3.1: Design comparison of existing NDN sync protocols

	CCNx 0.8 Sync	iSync	CCNx 1.0 Sync	ChronoSync	RoundSync	PSync
Sync namespace	Application data names	Application data names	Application data names	Node prefix + seq#	Node prefix + seq#	Stream prefix + seq#
Namespace representation	Name tree	IBF of hashes of names	Manifest storing names or digests of data	List of {prefix : seq#}	List of {prefix : seq#} + round log	IBF of hashes of names with highest seq#
State change detection	Periodic *RootAdvice* Interest	Periodic Interest carrying IBF digest	Notification Interest carrying hash of manifest	Data replying to long-lived *Sync Interest*	*Sync Interest* carrying round digest	Data replying to long-lived *Sync Interest*
State update retrieval	*NodeFetch* Interest retrieving child node hashes	Interest retrieving IBF content	Interest retrieving manifest	Data replying to long-lived *Sync Interest*	Data replying to *Data Interest*	Data replying to long-lived *Sync Interest*

To achieve group communication, the protocol may rely on the underlying network to provide multicast capability (which is the case in all existing sync protocols), or explore other group rendezvous mechanisms such as structured communication (e.g., Distributed Hash Table [SMK01]) and epidemic dissemination [DGH87]. Note that the design of the group communication mechanism is outside the scope of the sync protocol (and therefore not the focus of our analysis), but may have a significant impact on the protocol design choices.

Table 3.1 summarizes the design choices made by different sync protocols. In the rest of this chapter, we give a high-level survey of each sync protocol by focusing on the key design aspects mentioned above.

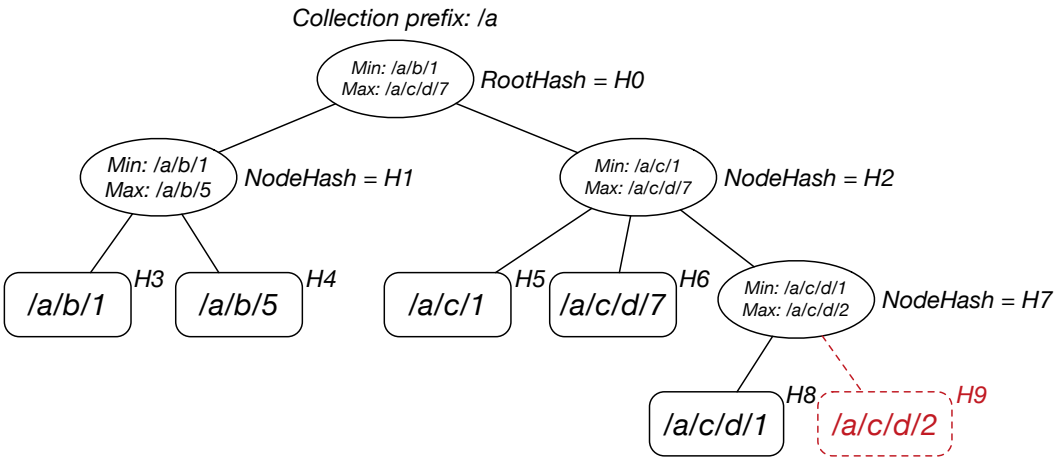

Figure 3.1: Example of a sync tree in CCNx 0.8 Sync

3.2 CCNx 0.8 Sync

The CCNx 0.8 Sync protocol [Pro12a] is the earliest synchronization solution originally proposed for the NDN/CCN architecture as a service module of the *ccnr* repo daemon. 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 Figure 3.1), where a node in that tree may store a single data name (i.e., a *leaf*) or a list of (leaf and non-leaf) nodes. 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 Figure 3.1, $H_3 = Hash(/a/b/1)$, $H_2 = H_5 + H_6 + H_7$, and $H_0 = H_1 + H_2$. The root hash (H_0 in Figure 3.1) then provides a summary of the entire namespace (i.e., sum of all data name hashes). Note that the sum of hashes is not a cryptographically strong summary: in certain cases two sync trees may store different sets of names but happen to have the same root hash.

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 Figure 3.1 the insertion of a new data "/a/c/d/2" (marked as the red dashed square at the bottom right) will cause the sync agent to update the 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. The RootAdvice Interest name starts with a multicast prefix for the sync tree, which is shared by all repos and different from the collection prefix, followed by the current root hash of the sync tree. When a sync agent receives a remote root hash that is different from its own, it replies to this RootAdvice with its own root hash. The sync agent who receives a RootAdvice reply will send a *NodeFetch* Interest, which is also named under the multicast prefix of the sync tree, to the replying repo to retrieve the list of hashes for all the children under the root node of the remote sync tree. The NodeFetch process is recursively applied to all the nodes in the sync tree, skipping those with the same hash value 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 normal Interest-Data exchange and insert that data to its local copy of the data collection. An example of the synchronization process in CCNx 0.8 Sync (triggered by the update to the sync tree shown in Figure 3.1) is illustrated in Figure 3.2. Note that while we show the sync protocol messages only between two repos for clarity, the RootAdvise and NodeFetch Interests actually carry multicast prefix and will be received by all repos storing the same data collection.

One problem in the update propagation mechanism in CCNx 0.8 Sync is that when multiple repos publish new data simultaneously, there will be more than one reply to a RootAdvice Interest and only one of them will be returned to the Interest issuer. In such case, the sync agent who sends the initial RootAdvice Interest need to issue additional

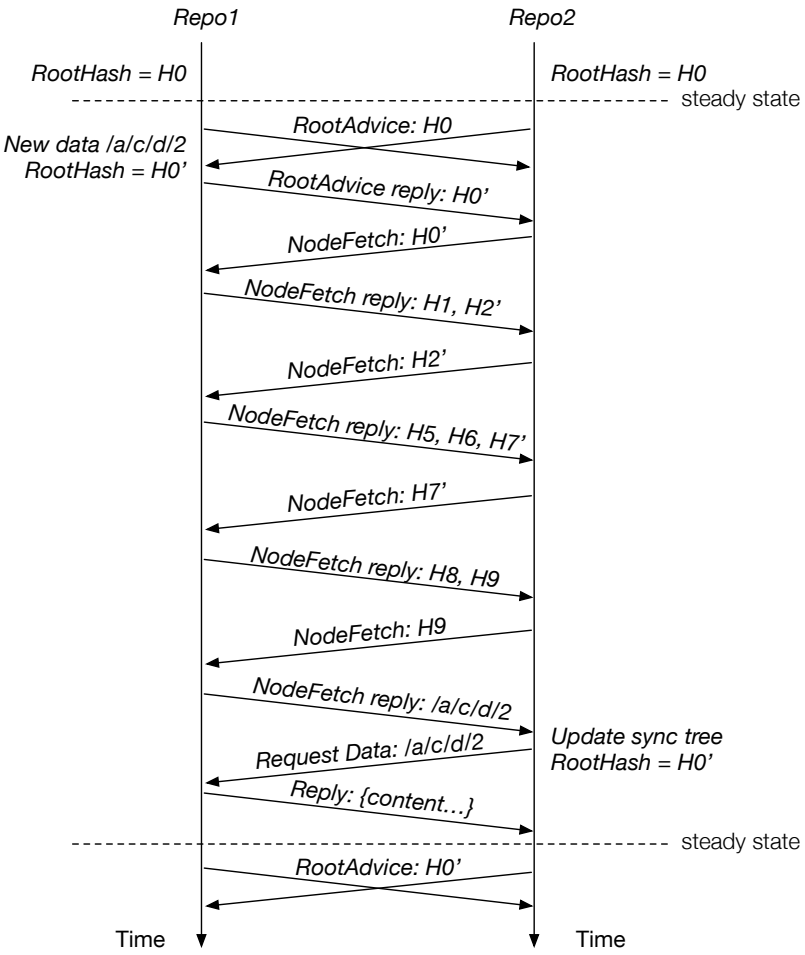

Figure 3.2: Synchronization in CCNx 0.8 sync

Interests to fetch other replies. The proposed solution is to attach *exclude filters* to the Interest to list the root hashes of the remote sync trees that have already been received. This ensures that each unique remote sync tree is examined only once for missing data.

A side-effect of the CCNx 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 must be monotonically growing, which creates usability issues with the applications who generate a large mount of data and need to perform garbage collection

periodically to reclaim the storage. For example, when the NDNVideo application [KB12] was deployed on top of CCNx repo to publish live video streams, the system administrator had to cleanup the data and restart all repo instances every day at midnight in order to avoid overwhelming the storage of the repo server.

3.3 iSync

iSync [FBC15] is a direct optimization on top of the CCNx 0.8 Sync design. Like in CCNx 0.8 Sync, it supports the synchronization of shared data with application names. To represent the sync state more efficiently, iSync uses Invertible Bloom Filter (IBF) [EGU11] to store all the names from the shared dataset in compressed form. Since the IBF can only store fixed-length items, the data names must be first mapped to fixed-length IDs (generated from the hash of the names) before they are added to the IBF. A bi-directional mapping table is maintained by every sync node so that it can recover the original NDN names from the IDs.

Different from CCNx 0.8 Sync,[2] iSync uses "digest broadcast" Interests (equivalent to the RootAdvise Interest in CCNx) to advertise its current state to other nodes periodically, rather than a solicitation for different sync states. Since the encoded size of the IBF is typically very big, the advertisement Interest carries only the digest of the current IBF from the sending node. When a node receives a digest different from its own, it sends another Interest to request the corresponding IBF content. After it receives the IBF from a remote node, the node subtracts its own IBF from the remote IBF and extracts individual IDs from the resulting "diff" IBF. Once the sync node extracts all new IDs, it issues Interests to request the original NDN names corresponding to those IDs and then fetches the new data using the original names. An example of the synchronization process in iSync is shown in Figure 3.3.

Note that the iSync node does not expect any reply to the initial advertisement Interest it sends. It therefore gets around the issue with multiple replies generated for the same

[2]The original iSync paper [FBC15] describes the CCNx 0.8 Sync protocol differently compared to the official specification [Pro12a] released in the CCNx source code package.

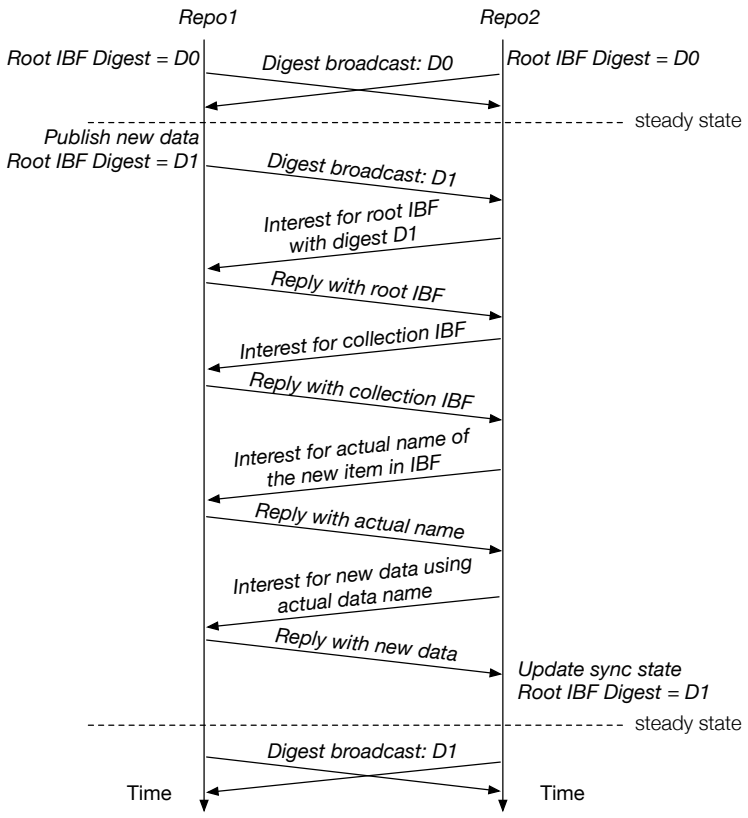

Figure 3.3: Synchronization process in iSync

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

A major limitation in the IBF data structure is that it can losslessly encode up to a certain number of items, beyond which some of the stored items cannot be extracted. Unfortunately, it is not uncommon that during the synchronization process the set difference between the namespace of some sync nodes may contain too many IDs that cannot be encoded in the IBF in a lossless fashion. iSync provides several ways to control the size of the set difference at multiple levels in the protocol design. First, the shared dataset is divided into multiple collections that host data for different applications; each collection maintains its own IBF independently from others. Second, iSync protocol enforces each

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; if 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.

3.4 CCNx 1.0 Sync

The design proposal of CCNx 1.0 Sync [Mos14] 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 generate new data. The advertisement Interests are also named under the data collection prefix and forwarded to all sync nodes announcing that prefix. They have short lifetime and do not retrieve any data. To increase the possibility that all nodes can receive the advertisement, the node repeats the advertisement Interest once or twice within a few seconds after the first advertisement is sent. Once a node receives a different hash, it should also advertise its own hash under the control of some gossip protocol (with random backoff and duplicate suppression). It then sends out Interests to retrieve the corresponding (possibly segmented) manifest packets, compares the names listed in the manifest with its local namespace, and then retrieves the missing data over the network. This approach is similar to iSync but without the benefit of efficient encoding and differentiation provided by the IBF data structure.

3.5 ChronoSync

Different from CCNx 0.8 Sync and iSync, ChronoSync [ZA13] improves the efficiency of dataset synchronization by utilizing naming conventions to simplify the sync protocol design. In ChronoSync, each node publishes data that contains application-layer messages under its own unique name prefix, which also serves as the identifier for the node in the sync group and is aligned with the routable prefix of the access network for each node. The data name is constructed by concatenating the node prefix with a sequence number that starts from zero and gets incremented by one for each new data published by the sync node. Real-world applications may require more complex naming conventions to encode richer information other than sequence numbers. ChronoSync supports those applications using "one level of indirection" by encapsulating the application data names (or the data itself if the size is small) in the content of the sequentially named data.

The sync node maintains a 2-level "flat" sync tree, as is shown in Figure 3.4, with each leaf containing the data prefix and the latest sequence number of each producer in the sync group. Each leaf is associated with the digest calculated over node's prefix and the latest sequence number. The root of the tree maintains the digest of concatenation of leaf digests canonically ordered by the corresponding prefix names. Since the naming convention is to publish data with continuously increasing sequence numbers (starting from zero), this sync tree is essentially a condensed representation of the namespace containing all the data ever published in the group, and the root digest is a short summary of the dataset.

ChronoSync nodes maintain *long-lived Sync Interests* in the network by transmitting a new Sync Interest immediately when the previous one expires or gets satisfied. The long-lived Interest stays in the pending Interest table 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 sync 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 solicits the next state changes generated on top of

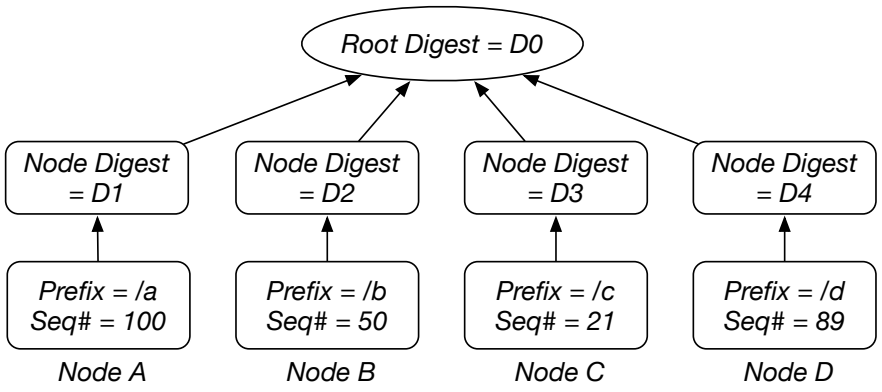

Figure 3.4: Example of a sync tree in ChronoSync

the state identified by the digest carried in the Sync Interest.

In the steady state, all nodes generate identical state digests and send out the same Sync Interest that is aggregated by the NDN forwarders. When some node publishes new data and increments its sequence number, instead of replying to the long-lived Sync Interest with its new root digest as in CCNx 0.8 Sync, the node replies with the name of its newly published data (i.e., the node prefix and the sequence number).[3] This *Sync Reply* is efficiently delivered to all the other nodes in the group, following the multicast tree built by the pending Sync Interest. After they receive the reply, the nodes update their local sync tree, recompute the root digest, and then send out Sync Interests carrying the new digest. An example of the synchronization process in ChronoSync is shown in Figure 3.5.

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

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

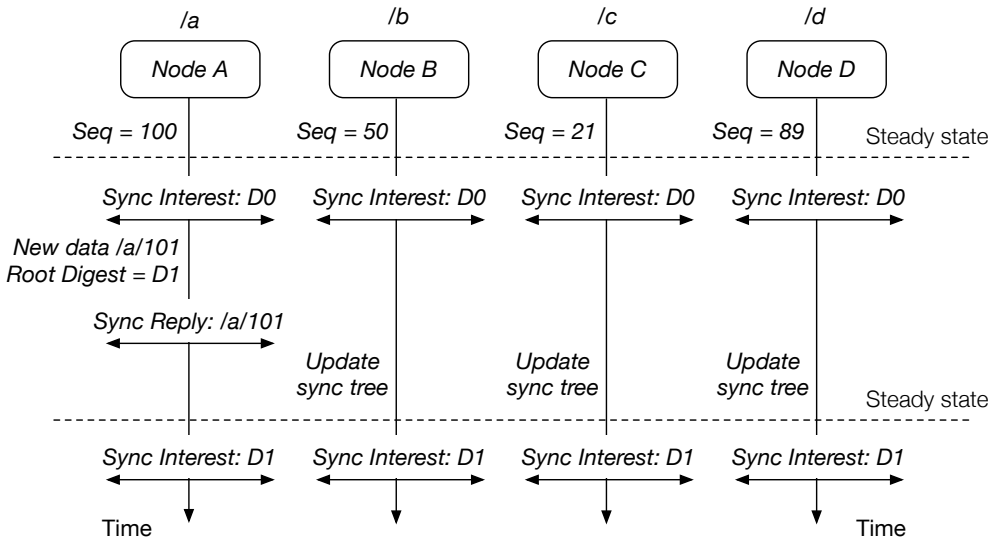

Figure 3.5: Synchronization process in ChronoSync

who is trying to catch up with the rest of the group.

There are several cases where a node may receive Sync Interests with unrecognized digests. For example, a node may receive a Sync Interest with an updated digest before receiving the Sync Reply that triggered the update. To handle that situation, ChronoSync injects a random delay to process the Sync Interest with unknown digest at a later time, hopefully after the corresponding Sync Reply has been received. Another scenario is when multiple nodes publish new data simultaneously, multiple Sync Replies will be generated in response to the same Sync Interest, and each node can receive at most one of the Sync Replies. After they update their local sync state based on the received Sync Reply, different state digests will be computed and announced in the sync group. A more complicated scenario 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 handles the simple case when the nodes diverge by at most one Sync Reply by resending the previous Sync Interest with exclude filters that contain the implicit digests of the received Sync Replies. However, if multiple changes have been applied to the sync state at some node, the mechanism using exclude filters will not be able to retrieve the

diverging sync replies generated by every node (see 3.6 for detail). In such case ChronoSync falls back to a *recovery* mechanism: when a node observes an unknown digest, it will send out a special *Recovery Interest* containing the unknown digest; the nodes who recognize that digest will reply with the complete information about its sync tree, rather than the specific changes that lead to that digest; when the requesting node gets the reply, it will merge the received sync tree into its local sync tree by taking the higher sequence number from both trees for each sync node.

3.6 RoundSync

RoundSync [HCS17] revises the ChronoSync design based on the following key observation: the Sync Interest in ChronoSync is overloaded with two functions: (1) detecting different states among the sync nodes and (2) retrieving the updates from other nodes. As a result, the Sync Replies carrying the updates to the shared dataset will be named after the previous Sync Interest name which contains the digest of the corresponding sync state. If a node generates Sync Replies on top of a diverged state (e.g., in the scenario with partitioned sync group), nodes with different state will not be able to derive the correct name for those Sync Replies and therefore cannot send Interests to retrieve them.[4] In that case ChronoSync must rely on the recovery mechanism to bring the group in sync again.

To address this problem, RoundSync introduces a new type of Interest packet called *Data Interest* in order to decouple state notification from update fetching. In RoundSync, the Sync Interest carrying the state digest merely serves as a notification mechanism (similar to iSync) so that the sync nodes can detect state divergence in the group when it happens. The updates generated by other sync nodes are retrieved via Data Interests whose names do not depend on the state digests. This allows the sync nodes to construct Data Interests to fetch the updates even if their states are not fully synchronized. The replies to the Data Interest achieve the same functionality as the Sync Reply in the original ChronoSync design, i.e., carrying node prefix and sequence number of the newly published data.

[4]Note that merging the diverged sync states will only create another sync state with a new digest value.

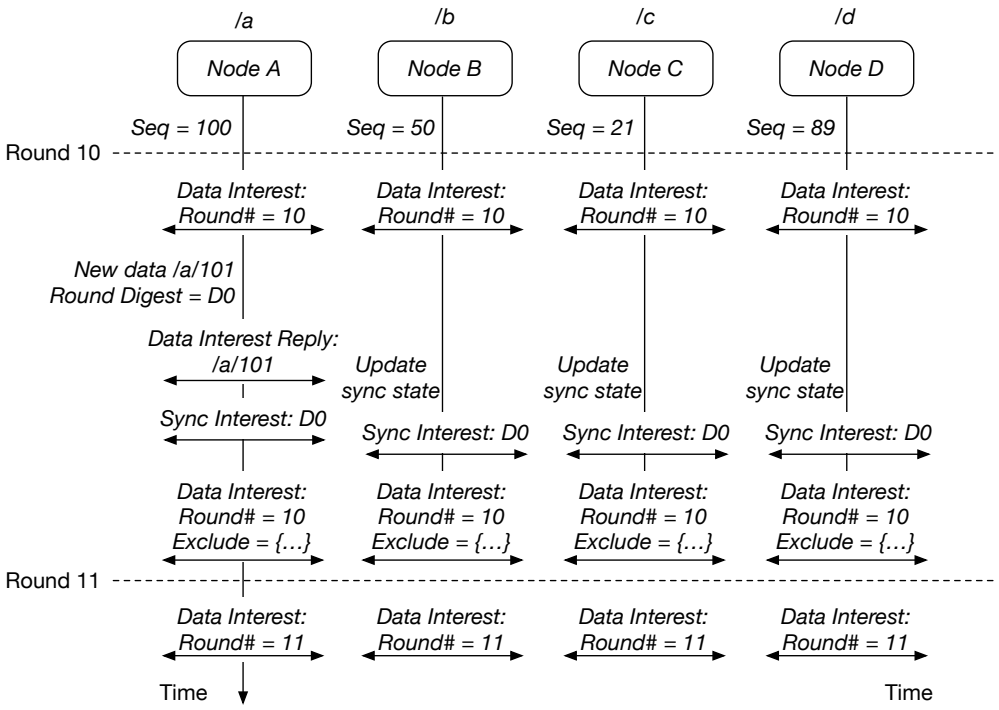

Figure 3.6: Synchronization process in RoundSync

Another major change made by RoundSync is to divide the synchronization process into multiple *rounds*, which are identified by unique round numbers. A sync node can publish at most one data packet in each round and must move to a new round when it receives new data published by others in the current round. This helps reduce the chances of state divergence caused by simultaneous data production. The names of both Sync Interest and Data Interest carry the round number so that each round is synchronized independently. For example, 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. A basic example of the synchronization process in RoundSync is shown in Figure 3.6.

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* for the previous rounds that have been stable for a long time (which therefore has high probability of remaining stable in the future). The cumulative digest for a round covers the entire dataset as observed in that 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.

3.7 PSync

PSync [ZLW17] was originally designed for the consumers to synchronize a subset of a large data collection with a single producer. The data packets published by the producer are organized into *data streams* which are identified by the unique stream prefixes. Like in iSync, PSync also employs IBF to represent the namespace by storing the hash of the names (called *KeyID*) in the fixed-length slot of the IBF. However, PSync also adopts the naming convention in ChronoSync and RoundSync that data packets from the same stream are ordered by the continuous sequence numbers. Therefore the IBF only needs to store the latest data name from each stream. This further reduces the amount of information stored by the IBF and allows the applications to choose a smaller IBF size that can be transmitted more efficiently over the network.

To support the synchronization of a subset of the producer's data (a.k.a., *partial sync*), PSync introduces the subscription list to encode the prefixes of the data streams that the consumer is interested in.[5] The subscription list is a Bloom Filter (BF) that stores the hashes of those stream prefixes. The size of the Bloom Filter is determined by the total number of

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

streams a consumer may subscribe to and the false positive rate the consumer is willing to accept. Special cases like empty and full subscription may be encoded more efficiently with special markers.

During the sync process, the consumer keeps a local copy of the producer's IBF which indicates the data it has received so far. To sync up with the producer and retrieve new data, the consumer maintains *long-lived Sync Interest* whose name contains the local IBF copy and the consumer's subscription list. When the producer publishes new data, it first subtracts the IBF in the pending Sync Interest from its new IBF, and extracts the KeyIDs of the new data packets that have not been received by the consumer yet. Then the producer checks whether the stream prefixes of those new data packets are included in the consumer's subscription list (subject to certain false positive rate). Finally the producer generates a Sync Reply containing the original names of the new data packets in the subscribed streams and also its latest IBF. Upon receiving the Sync Reply, the consumer updates its local IBF copy with the received IBF, and sends out Interests to fetch the new data. An example of the synchronization process in PSync is shown in Figure 3.7.

An important feature in the PSync design is that each consumer maintains its own data consumption and subscription status. The producer, on the other hand, does not maintain per-consumer state, which significantly reduces the amount of data stored by the producer. If multiple producers are serving the same set of data streams, the consumers may send Sync Interests via *anycast* to get replies from any producer that is available online, assuming that these producers have run sync protocols among themselves to sync up their dataset. However, this *stateless* producer design introduces two additional costs: first, the Sync Interest and Sync Reply need to carry the IBF and the subscription list (BF) which will bloat the size of the Interest name up to hundreds of bytes; second, the producer needs to generate Sync Replies in real-time for each Sync Interest since it does not remember the previous consumption status of each consumer and cannot pre-generate the next Sync Reply.

While it was initially designed for producer-consumer synchronization, PSync can also be extended to support group synchronization (like other sync protocols previously discussed) where each sync node is both producer and consumer at the same time. This is achieved by

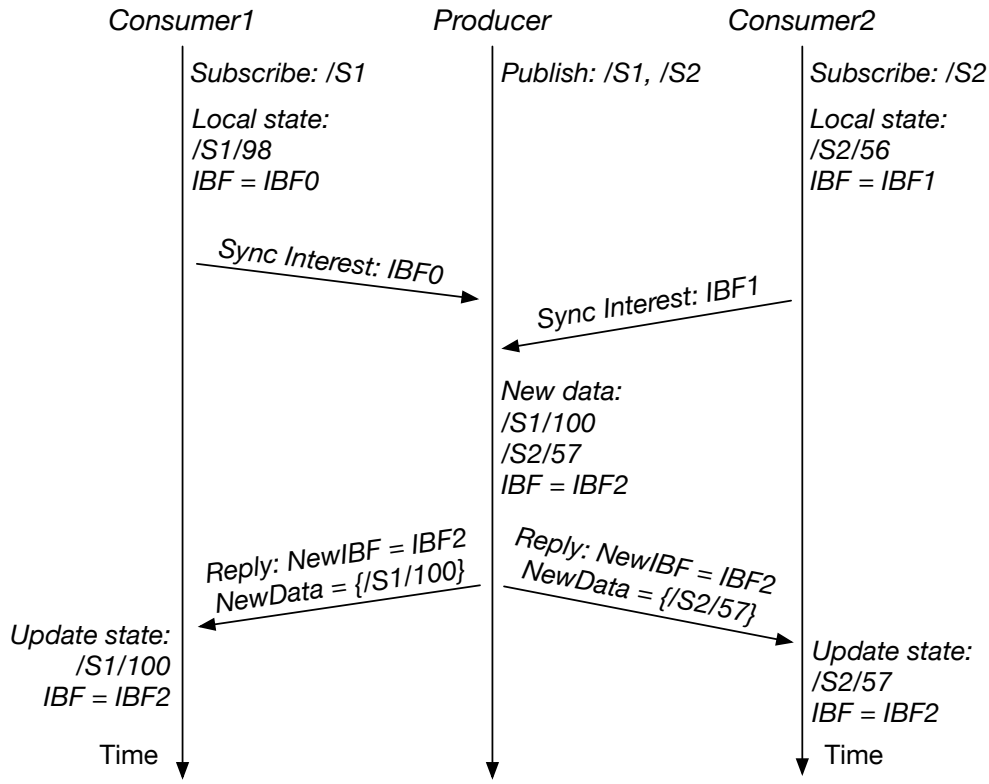

Figure 3.7: Synchronization process in PSync

having every node subscribe to all data streams published by every other node. However, in this "full synchronization" mode each node only needs to maintain a single IBF which represents the state of the whole dataset, rather than keeping a separate IBF for each node in the group. In addition, the Sync Interests need to be forwarded via *multicast* to the entire group so that any node who has produced new data can respond with a reply that carries the updates.

3.8 Summary

From the previous discussion, we can see that a few common design patterns have arisen in the key design aspects among the existing NDN sync protocols. ChronoSync, RoundSync, and PSync have adopted the sequential data naming conventions, i.e., naming the data packets using sequence numbers under a common name prefix for each producer or data

stream, to simplify the representation of the shared dataset namespace. Having continuous and monotonically increasing sequence numbers in the data name allows the cumulative data collection generated by the same producer or in the same data stream to be summarized by the highest sequence number. This reduces the amount of information that needs to be encoded in the sync state and also simplifies the protocol design since the sync protocol needs to only focus on synchronizing the latest sequence numbers rather than the whole namespace.

The existing sync protocols have used a variety of data structures to represent the sync state. All of those data structures provide lossless encoding of the data names (or the hashes of the names) in the shared dataset. CCNx 0.8 Sync and CCNx 1.0 Sync enumerate the dataset namespace in the hierarchical sync tree and the "flat-structured" manifest, respectively. To reconcile the set difference, the sync nodes simply compare the content in the sync tree or the manifest and then retrieve the missing data from the remote nodes. ChronoSync and RoundSync also enumerate the dataset namespace by listing the latest sequence numbers from all data producers in the sync state. State reconciliation is achieved by comparing the sequence number of each producer between the local and remote sync states and taking the maximum as the latest sequence number. iSync and PSync use IBF to compress the dataset namespace and perform set reconciliation using IBF subtractions. However, due to the limited IBF capacity, both iSync and PSync have to provide means for controlling the size of the set difference between the IBFs maintained by different nodes.

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

Table 3.2: 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)	Depending on how to retrieve manifest segments	Min is 0.5 RTT; can be long with simultaneous publishing	Min is 1.5 RTT; can be long with simultaneous publishing	1.5 RTT (assuming a single producer)
Interest overhead	Periodic	Periodic	One per update	Long-lived Interest	Two per update	Long-lived Interest
Factors affecting Interest size	Node hash	IBF digest	Manifest digest	State digest (+ exclude filter)	Round digest (+ exclude filter)	IBF + subscription list
Factors affecting Data content size	Number of children under the requested node	IBF size (depending on the number of new data)	Size of the entire dataset	Number of names with new seq#	Number of names with new seq# in a round	IBF size + number of names with new seq#

It is often difficult to compare the efficiency of different sync protocols because it usually depends on the application scenarios and the implementation choices. Table 3.2 compares the existing NDN sync protocols on a few important performance metrics. One metrics is the data dissemination delay, i.e., the number of round-trips necessary for propagating new data to other nodes. In CCNx 0.8 Sync and iSync, the synchronization process is triggered periodically based on an internal sync timer. Once the process starts, the number of round-trips required to retrieve all updates from a remote node in CCNx 0.8 Sync depends on the depth of the sync tree, while in iSync the process usually finishes within 3.5 RTT, unless the number of changes exceed 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 there is new data published in the dataset, and the data dissemination delay depends on

how the nodes retrieve the segmented manifest (e.g., sequentially or pipelined).

ChronoSync and RoundSync achieve optimal synchronization delay when there is no simultaneous data publishing. If multiple nodes generate Sync Replies at the same time, the protocols need additional round-trips to retrieve all Sync Replies using Interests with exclude filters. Therefore the worst-case RTT will be proportional to the number of simultaneous updates in the group, which is bounded by the number of data publishing nodes in the group. PSync achieves the data dissemination delay of 1.5 RTT because the Sync Interests carry specific information about the state of the consumer, which allows the producer to reply with specific changes without spending additional round-trip to request more information. Note that both ChronoSync and PSync require maintaining long-lived Sync Interests in the network so that the replies can be propagated to other nodes as soon as possible. This leads to the overhead of keeping long-lived soft state in the forwarders' PIT (with one PIT entry per sync group).

Another performance metrics is the packet size of the sync protocol messages, which reflects the network bandwidth requirement of the sync communication. Here we mainly focus on the encoding size of the sync state (or state updates) carried in the Interest and/or Data packets. For iSync and PSync, the size of the IBF that summarizes the dataset namespace depends on the size of the hash function output and the data publishing rate of the applications. In a typical implementation that uses 64-bit hash functions and 32-bit counter values, the size of each slot in the IBF is 20 bytes. For an IBF with capacity of 20 items (i.e., allowing at most 20 items to be extracted successfully), the encoded size of the IBF [EGU11] is around 1.5 * 20 * 20 = 600 bytes. Once the size of the IBF is chosen, all Interest and Data packets carrying the IBF will have the same size even if the number of updates is lower than the maximum capacity. Note that PSync usually requires a smaller IBF than iSync because the sequential data naming simplifies the namespace and effectively bounds the number of changes by the number of data prefixes. This enables PSync to carry the IBF directly in the Sync Interest. In iSync, the number of changes within an Interest period depends on the data publishing rate and is unbounded, therefore requiring a larger IBF to accomodate bursty data publishing events.

In contrast, ChronoSync and RoundSync require only the updates to be propagated in reply to the Sync Interests and Data Interests, respectively. Those update data packets contain the prefix and the latest sequence number from each producer who has published new data. Assuming the average size of the data name (i.e., prefix + sequence number) is 40 bytes, the maximum content size of the update is 40 * N, where N is the number of producers in the sync group. In practice, not all producers will be publishing at the same time and the size of the update packets is typically smaller than in the IBF-based approaches. The Sync Interests and Data Interests in ChronoSync and RoundSync usually carry the current state digest only. However, when simultaneous data publishing happens, the nodes need to send additional Interests with exclude filters that enumerate the 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; also, the protocol requires multiple NodeFetch packets to resolve all the differences. In CCNx 1.0 Sync, the size of the manifest is proportional to the number of data names listed in the manifest.

CHAPTER 4

Design of VectorSync Protocol

This chapter presents the design and evaluation of VectorSync, a new sync protocol for the NDN architecture. VectorSync learns from the past experience in developing the existing NDN sync protocols analyzed in Chapter 3. The protocol design also benefits from the rich set of literature on distributed synchronization algorithms and protocols developed in the distributed system research area. The goal of designing this new protocol is to (1) address the issues identified in the design of existing NDN sync protocols, and (2) enable new functions not covered by the existing works.

4.1 Motivation

From the discussion in Chapter 3, we can see that all existing sync protocols have their own design issues. CCNx 0.8 Sync represents the dataset namespace using a tree structure, which requires multiple round-trips to walk down the tree and detect all the differences in the namespace. iSync and CCNx 1.0 Sync reduce the synchronization round-trip at the cost of larger encoding size for the namespace representation. ChronoSync, RoundSync, and PSync all use Interest packets to fetch the update generated on top of the state identified by the digest or IBF in the Interest name, which faces the issue with simultaneous data publishing from different producers:[1] the nodes need to send additional Interests (with exclude filter) to discover the simultaneous replies in multiple round-trips. Moreover, ChronoSync and PSync use long-lived Interest to pre-establish the return path for the data carrying the state changes, which causes the overhead of maintaining the soft-state PIT entries in the network

[1]Note that PSync was originally designed for synchronizing with a single producer.

(through periodic retransmission of the Interests).

VectorSync addresses those issues with a new protocol design. First, VectorSync adopts the sequential data naming convention that has been used in ChronoSync, RoundSync and PSync to simplify the dataset namespace. This enables VectorSync to represent the state of the dataset concisely and efficiently using *version vector* [PPR83], a well-known mechanism for detecting and reconciling mutual inconsistency in distributed systems. Second, VectorSync avoids the long-lived Interest mechanism and instead uses a notification-driven approach in the sync communication. The notification is an Interest packet that carries specific information about the name of the newly published data, which enables the sync nodes to fetch the new data as soon as they receive the notification. Third, VectorSync piggybacks the version vectors in the sync-layer data (which carries the application data name or content) to allow the receiving nodes to detect inconsistency in the sync state without requiring another round-trip to fetch the state information. With those design decisions, VectorSync is able to achieve lower data synchronization delay and less communication overhead compared to the existing sync protocols.

Another key motivation of designing a new sync protocol is based on the observation that many distributed applications require explicit group membership management so that the nodes can always keep track of the current active participants in the application. A typical example of such applications is resource discovery in IoT networks, where the IoT devices announce their services by publishing the relevant information (e.g., service name and description) in a "discovery dataset" and synchronize with other devices. In this application scenario, the devices need to know what other devices are active in the system so that their services are still available online. This requires the nodes to maintain explicit membership information about the sync group. Another application example is sync-based routing, where the routers need to keep track of the reachability information advertised by each active router in the network and remove obsolete routes from the dead routers as soon as possible.

The existing sync protocols operate over *unmanaged* sync groups and do not provide the built-in support for group membership management, which makes it difficult to remove inactive nodes from the protocol state maintained by each distributed node. It also creates

challenges in implementing services such as group-wide dataset snapshot and data total ordering for the sync-based applications due to the lack of consistent group membership information at each node. 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 controlling the size of the protocol state, but also for efficient data loss detection within a "closed" space of the protocol state. It also enables VectorSync to provide high-level sync-related services for the distributed applications (which will be described in Chapter 5).

In the rest of this chapter, we first introduce the system model for the VectorSync protocol. Then we describe the design components of VectorSync in detail. After that we present an extensive simulation-based study to evaluate the performance of the protocol under various network environments.

4.2 System Model

In this section we describe the system model and assumptions about the sync nodes and the networks in the design of the VectorSync protocol. We consider a sync group with a finite number of nodes (called *sync nodes*) participating in some distributed NDN application. Nodes may join and leave the sync group at any time. Although the synchronization algorithm in VectorSync can support a sync group of arbitrary size, in practise we limit the size of the group so that the entire sync state can be encoded in a single Data packet. Each node is assigned a data publishing prefix that is aligned with the topological prefix of the underlying network and unique within the sync group. The underlying network is unreliable: packets may be lost, corrupted, delayed, duplicated, or reordered during transmission. However, we assume the sync nodes in the same group can communicate with each other most of the time. The network may be partitioned temporarily, dividing the sync group into multiple subgroups, but will eventually reconnect.

Applications running on the sync nodes publish data in the shared dataset through the

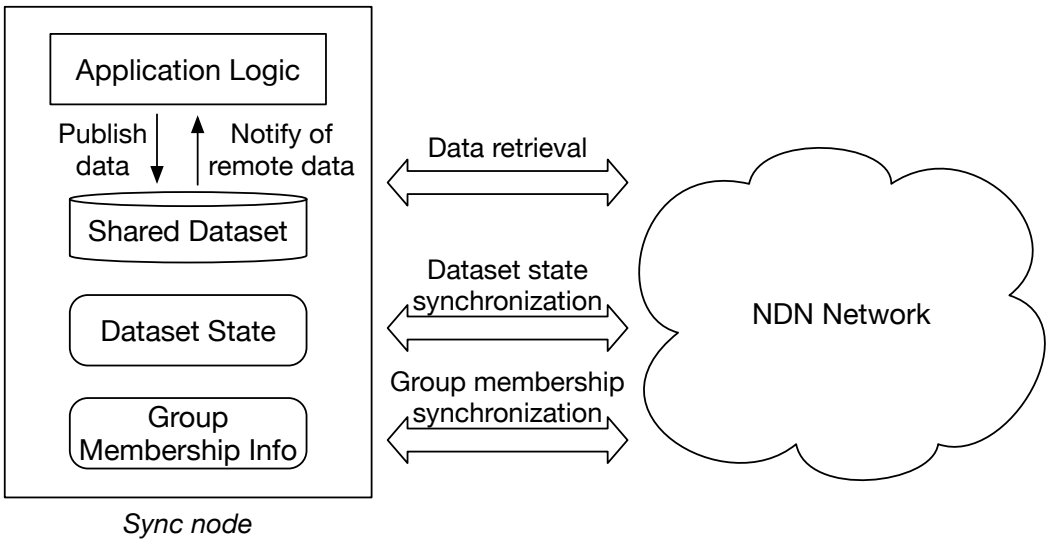

Figure 4.1: VectorSync Protocol Software Architecture

interface provided by the VectorSync service module. The information about the published data is propagated to other sync nodes over the network by the VectorSync protocol. After it receives new data published by other nodes, the VectorSync module notifies the application layer using a pre-configured callback function. When multiple applications are running on the same set of nodes, each application creates its own sync group and operates independently. Different applications do not share the dataset in which they generate new data.

Figure 4.1 illustrates the software modules implemented by a VectorSync node. Each VectorSync node maintains three important data structures:

- *Shared dataset*: the local storage of the data items published in the sync group. Data published locally or received from the remote nodes via VectorSync is stored in this data structure for easy access by the local applications.

- *Dataset state*: a version vector representation of the shared dataset namespace containing all published data that the sync node is aware of. This data structure summaries the node's latest knowledge about the state of the shared dataset and supports efficient set difference reconciliation between two copies of the dataset.

- *Group membership list*: the list of current active members in the group and associated

information, including their data publishing prefixes and public key certificates. The membership information is often referred to as the *view* of the group, a terminology initially devised by the Viewstamped Replication protocol [OL88, LC12] and adopted by many other synchronization protocols that also perform group management. Each view is uniquely identified by a *view identifier*, which is used for detecting inconsistent knowledge of the group membership among the active participants.

4.3 Basic Protocol

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

In this section, we first introduce the naming design and the representation of the dataset namespace. Then we describe the details of the dataset state synchronization process and the group membership synchronization mechanism. At the end of this section we briefly describe how to secure the VectorSync protocol using NDN's built-in data authentication support.

4.3.1 Data Naming and Dataset State Representation

Figure 4.2(a) shows the naming convention for the data in the shared dataset (called *Node Data*). Each node publishes data in the shared dataset under its own data publishing prefix that is aligned with the unicast prefix of the underlying network. This data prefix also serves as the unique node name in the sync group. The "app-name" component indicates the name of the application, which is known to every node in the group a priori. The last component

(a) Node data name:
/[unicast-node-prefix]/[app-name]/[seq#]

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

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

Figure 4.2: VectorSync naming conventions

in the data name is a monotonically increasing sequence number that uniquely identifies the data objects from the same producer. Note that many applications publish data in a dedicated namespace that is independent from the topological prefixes of the nodes and may require more expressive data naming conventions (e.g., for expressing trust relationship). Like in other sync protocols that utilize sequential data naming, the application data names can be encapsulated in the sequentially named Node Data published by the sync module. After receiving the Node Data, the applications can decide whether to fetch the application-layer data using the encapsulated name. If the size of the application data is small, the applications may also encapsulate the whole data object in the content of the Node Data, which saves the round-trip of retrieving it by name.

Since each node's sequence number increments continuously, the entire namespace of the shared dataset is concisely summarized by a list of *(node name, latest sequence number)* pairs. This namespace representation essentially becomes a *version vector* [PPR83] that identifies the state of the shared dataset. VectorSync further compresses the version vector representation by pre-configuring the data prefix of each node via the consistent group membership list maintained at each node (described later). This allows VectorSync to omit the node prefixes in the version vector, reducing it to an array of nonnegative integers (called *state vector*) that can be efficiently encoded and transmitted over the network. The order of each node in the version vector is determined by the canonical order of the node prefixes so there is no ambiguity in interpreting the vector as long as the nodes have the same view of the group membership. Figure 4.3 shows an example of representing the namespace of a

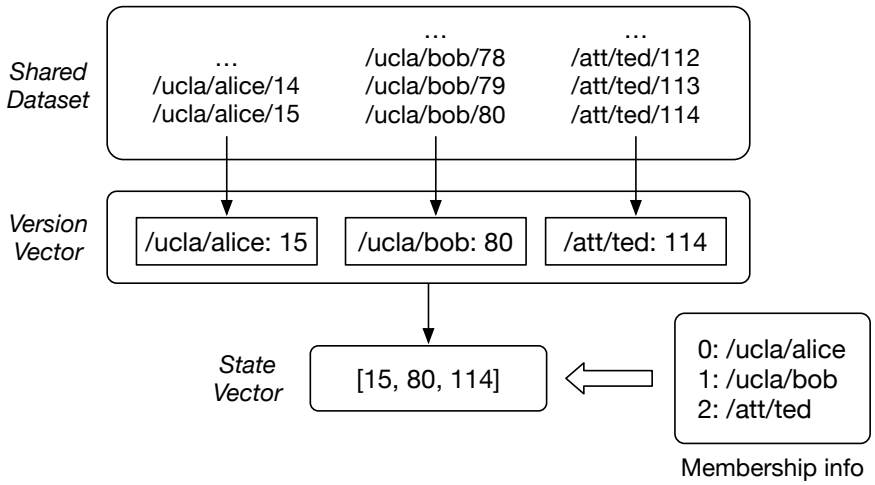

Figure 4.3: Representing the dataset namespace using a state vector

dataset with a state vector based on the pre-configured group membership information.

4.3.2 Dataset State Synchronization

The synchronization of the dataset state is performed in two steps. When a node publishes new data, it sends out a multicast notification Interest to announce the data in the sync group. After receiving the notification Interest, other sync nodes issue a second Interest to fetch the data based on the information carried in the name of the notification Interest. The Node Data carries the publisher's full state vector together with the application-layer message, which allows the receiving node to check for inconsistency in the dataset caused by various factors such as the loss of previous notification Interest and/or Node Data.

Figure 4.2(b) shows the naming convention for the notification Interest name, which starts with a multicast prefix that uniquely identifies the sync group (and the application), and carries the publisher's name and the sequence number of the published data at the end. Upon receiving a notification Interest, a node sends a reply packet with a short freshness period (e.g., 5ms) to satisfy the pending notification Interests in the network and provide an acknowledgement to the notification. The reply also carries the replying node's own state vector, which provides an opportunistic channel for propagating sync states to the

data publisher.[2] If the publisher 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 other nodes 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.

Since the notification carries specific information about the published data (i.e., publisher name and the data sequence number), the node can easily construct the name of the new data following the naming convention in Figure 4.2(a), and issue Interest to retrieve that data. The Node Data carries the publisher's state vector at the time the data is produced and the view ID representing the publisher's knowledge of the group membership, which provides context for interpreting the state vector. If the publisher's view ID is different from the node's local view ID, the node may need to synchronize the group membership information (described in the next subsection) before processing the state vector. Having the consistent group membership information ensures both local and received state vectors contain latest sequence numbers from the same set of nodes, with each node's sequence number listed at the same position in the vector. The receiving node then performs a *Join* operation that takes the entry-wise maximum of the received and local vectors. The result represents the union of the local and the remote dataset.

Definition 4.3.1. Given $v_1 = (a_1, a_2, ..., a_n)$ and $v_2 = (b_1, b_2, ..., b_n)$,

$$Join(v_1, v_2) = (Max(a_1, b_1), Max(a_2, b_2), ..., Max(a_n, b_n)) \tag{4.1}$$

The node replaces its local state vector with the output of *Join*. If any entry in the new state vector contains a higher sequence number than before, it will issue Interests to fetch the new data identified by the sequence numbers in between. The data publishing prefix of the node corresponding to that entry is readily available in the membership list.

An example of dataset synchronization among three nodes is illustrated in Figure 4.4. In

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

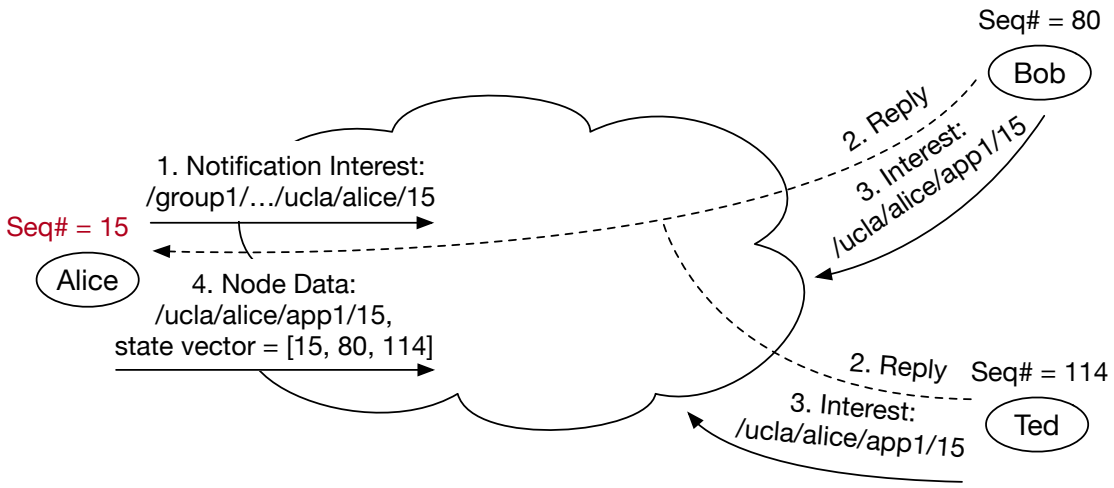

Figure 4.4: VectorSync protocol message exchange in a group of three nodes

this example, Alice publishes a new data packet with sequence number 15 and sends out a notification Interest via multicast to the group. After sending the reply, Bob and Ted issue Interests to fetch the new data from Alice. When there is no packet loss and no cached data in the network, the minimum delay for propagating the data from the publisher to another node is $1.5 \times RTT$ (plus processing delay).

Packet loss and link failure in the network may cause the nodes to miss some data published by others, due to the loss of notification Interests and/or Node Data. VectorSync provides two measures for detecting and eventually reconciling inconsistency in the dataset state. First, VectorSync requires each node in the group to periodically publish new data. When the application is idle, the nodes still publish *heartbeat* packets with no application message. (The heartbeat mechanism is also required for group membership maintenance, which is described in the next subsection.) As long as there is no permanent failure in the network, the nodes will eventually receive some new data from each member in the group. Since the sequence number is continuous, the receipt of a newer sequence number will allow the nodes to detect any missing data generated earlier by the same publisher. Second, both the notification reply and the Node Data carry the state vector of the node who generates the data. Due to periodic data publishing from each node in the group, the nodes can periodically detect and reconcile inconsistent state using the state vectors.

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 fast recovery from packet loss in the network, instead of waiting for new data to be published in the group.

4.3.3 Group Membership Synchronization

VectorSync operates over *managed groups*, where each node keeps the information about the current active members in the sync group, which is called the *view* of the group, and synchronizes dataset state with the nodes in the same view. To synchronize the view among a group of nodes, the node with the highest-ordered name is selected as the leader of the view and publishes the definitive information about the current members (called *ViewInfo*) which is followed by other nodes in the group. Each view is uniquely identified by a pair that includes a monotonic-increasing view number and the name of the view leader. The leader increases the view number when it creates a new view after membership change. The view ID essentially becomes a logical clock [Lam78] that keeps track of the view change events. All notification Interests carry the current view ID of publisher in the Interest name, as is shown in Figure 4.2(b). The view ID is also piggybacked in the Node Data together with the publisher's state vector. This allows the nodes to detect inconsistency in the group membership knowledge and then synchronize their views.

In VectorSync, each node is required to publish data in the shared dataset periodically in order to refresh its membership in the group. When the application is idle, the node publishes heartbeat data with no application message in the shared dataset.[3] The application data and heartbeat packets are propagated to all other group members via dataset synchronization and serve as the authenticated assertion of the node's active participation in the group. This allows the nodes in the same group to monitor the membership status of each other without introducing additional liveness detection mechanism. Like normal application data, the heartbeat data contains the publisher's current view ID and state vector, which helps the receiving node synchronize the dataset state as is described in Section 4.3.2.

[3]The heartbeat messages are processed only by the VectorSync module and not passed to the application layer.

If a node's heartbeat is missed for M times ($M = 3$ in our current implementation), the node is considered to have left the group. When the view leader detects some node has left the group, it initiates a *view change process* to move the remaining members to a new view.[4] 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.

Note that the heartbeat mechanism assumes the nodes' clocks advance at roughly the same speed, but does not require the clocks to be synchronized. A smaller heartbeat interval allows the group to react to node departure more promptly, but may lead to higher communication overhead because more heartbeat messages need to be propagated. Setting the heartbeat interval too small may also introduce unnecessary view changes when some node experiences short-term network disconnection. In practice, applications should choose the heartbeat interval to be roughly the same as the average application data rate and adjust the node timeout threshold M based on the deployment scenarios.

To start the view change process, the leader increments the current view number by one and then publishes the ViewInfo packet for the new view following the naming convention shown in Figure 4.2(c). Note that the ViewInfo is named under the multicast group prefix rather than the unicast leader name, which allows any node in the group to store and serve the ViewInfo packet to other nodes. The ViewInfo contains the list of active members in the view with their unique node names (i.e., data publishing prefixes) and public key certificates, which essentially provides a certificate bundle for all members in that view. The ViewInfo may be segmented if it is too large to fit into a single Data packet. After the ViewInfo is generated, the leader publishes a heartbeat packet in the new view, which triggers a notification Interest that carries the new view ID to inform other nodes about the new view.

Upon receiving a notification Interest with a higher view number (than its own), a node tries to fetch the corresponding ViewInfo based on the view ID carried in the Interest name.

[4]Optionally, the leader may inject a short delay before removing the inactive node. If multiple nodes leave the group around the same time, the leader can remove all of them from the view via a single view change process.

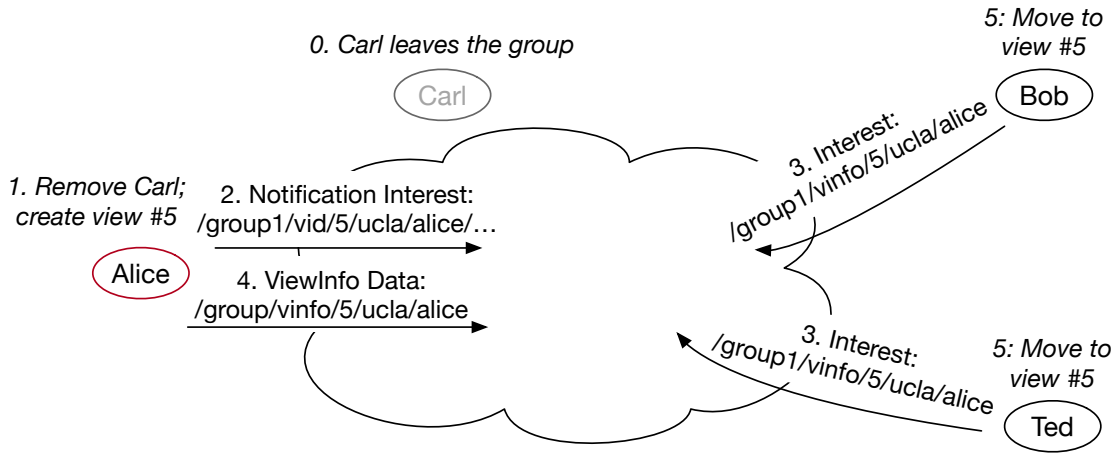

Figure 4.5: View change process after removing a node

Before receiving the new ViewInfo, the node keeps publishing data in its current view. The node joins the new view if it is included in the membership list of the new view. Otherwise it ignores the received ViewInfo and stays in its current view. After joining the new view, the node needs to update the dataset state by resizing the state vector to the number of members in the new view and filling the state vector with the latest sequence numbers for the members the node already knew. For unrecognized members their entries in the state vector will be initialized to zero. After that the node can start processing the state vector carried in the data packets published in the new view, as is described in the previous subsection.

Figure 4.5 illustrates a simple example of the view change process happening in a group when the leader Alice announces a new view (5, /ucla/alice) after Carl leaves the previous view (4, /ucla/alice). When there is no packet loss, other nodes will join the new view $1.5 \times RTT$ after the leader sends out the notification Interest to announce the new view. Note that this view change process is similar to the dataset synchronization process described earlier: the leader publishes the ViewInfo data packet and announces the view ID in the notification Interest name so that other nodes in the group can fetch the ViewInfo packet using the data name constructed using the view ID (see Figure 4.2(c)). In other words, VectorSync 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.

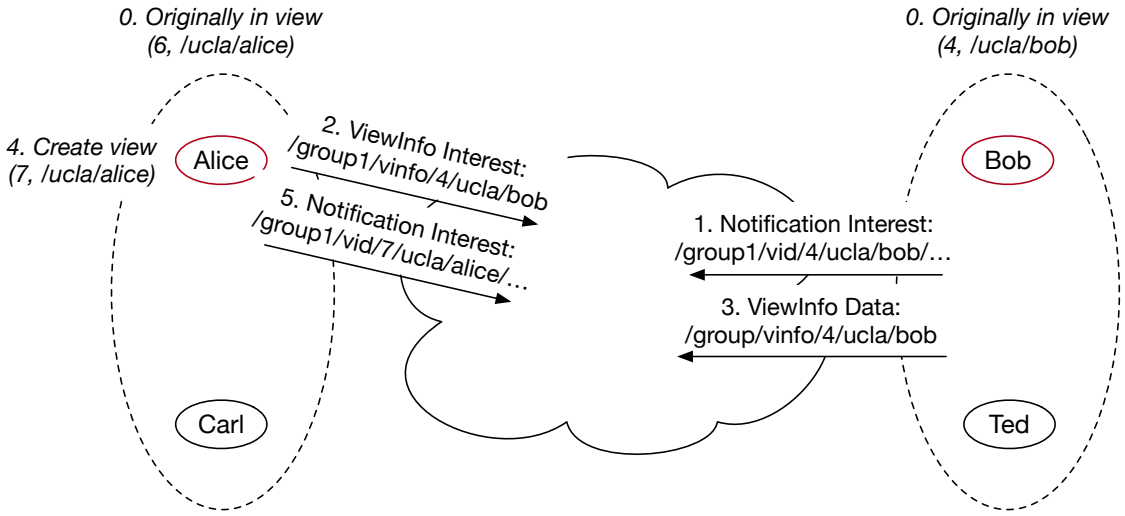

Figure 4.6: Merging two sub-groups after network partition heals

When network partition happens, each partition may select its own leader that creates a new view including a subset of nodes in the group. After the network partition heals, the leaders of different views will eventually receive notification Interests from each other that indicate the view IDs of the divided sub-groups. In this case, VectorSync requires the leader with the highest-ordered name among all existing leaders to take the responsibility of merging the sub-groups. This ensures the new view will still have the node with the highest-ordered name as its leader. When a leader node detects a different view with a smaller leader name, it fetches the corresponding ViewInfo, creates a new view containing members from both existing views, with the view number set to one plus the higher of the existing view numbers, then announces the new view ID to the group. Before merging the views, the leader needs to authenticate the received ViewInfo data to make sure the nodes in the other view are authorized to participate in the sync group. The detail of member authentication is discussed in the next subsection.

Figure 4.6 illustrates an example of the view change process that merges two sub-groups originated from the same sync group, with view IDs (4, "/ucla/alice") and (6, "/ucla/bob"), respectively, into a single group (7, "/ucla/alice").

An interesting fact to note is that VectorSync does not provide a separate group-joining mechanism. To join an existing sync group, a node first creates a single-node view with view

number equal to 1 (the smallest possible view number) and itself being the leader. Then the leader of the existing view will follow the view-merging process described above to add the new node into the group.

4.3.4 Securing VectorSync Communication

VectorSync requires that the nodes in the same 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 nodes, 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 node'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 [ZWY11]. The leader may periodically generate a symmetric data encryption key and distribute the key to every node on the current membership list [YAZ16]. The key distribution process can utilize the dataset synchronization mechanism: the leader publishes the data encryption key (encrypted with each member's public key) in the shared dataset, which is synchronized to every member in the current view. The producers encrypt the application messages using the latest encryption key, which can be decrypted only by the members in the same view.

4.4 Simulation Study

We implemented a prototype VectorSync module [Sha17] using the ndn-cxx library.[5] In this section we study the behavior of VectorSync under various network environments by running simulation experiments on the latest version of ndnSIM [MAM16] simulator which supports easy integration of real-world applications developed with ndn-cxx. To evaluate the performance of VectorSync, we developed a simple application that runs on top of VectorSync and publishes data continuously in a shared dataset following the Poisson process with a pre-configured data rate. We focus on three important performance metrics:

- Data dissemination delay: the time needed for a piece of data to be received by *any* node in the group after it is published;
- Data synchronization delay: the time needed for a piece of data to be received by *all* nodes in the group after it is published;
- Network traffic volume: the amount of Interest and Data packets sent in the network during the experiment.

We start by analyzing the data synchronization process in VectorSync using a simple hub-and-spoke network topology with a fixed sync group. Then we compare the performance between VectorSync and ChronoSync using more realistic topologies, including a small campus network and a large ISP network. In the end, we analyze the behavior of VectorSync under dynamic group changes.

4.4.1 Synchronization in Lossless Network

We start with the basic scenario when there is no packet loss in the network. We set up the simulation scenarios with a simple hub-and-spoke network topology, where all nodes connect to a common central hub via point-to-point links. Using such simple topology allows us to calculate the expected values of the performance metrics based on the protocol

[5] `https://github.com/named-data/ndn-cxx`

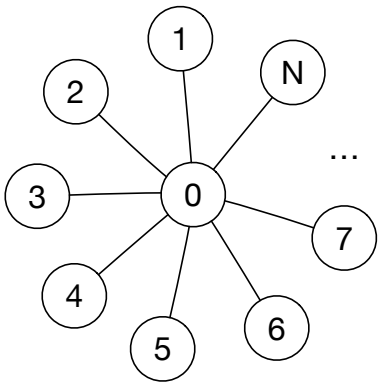

Figure 4.7: Hub-and-spoke topology

design and then validate our analysis using the simulation results. All participating nodes are pre-configured the group membership information that stays unchanged throughout the simulation. Each node publishes new data continuously in the shared dataset with an average inter-arrival time of 1 second. Each simulation runs for 100 seconds, allowing each sync node to publish about 100 packets in total. The heartbeat interval is set to be the same as the average inter-arrival time of the application data (i.e., 1 second).

A hub-and-spoke network topology with N nodes is shown in Figure 4.7. To simplify the analysis, we configure each point-to-point link to have the same link delay D. As a result, the RTT between any pair of sync nodes is $4 \times D$. When there is no packet loss, the minimum data dissemination delay from the producer node to another sync node is $1.5 \times RTT = 6D$ (plus a small amount of queueing delay modeled by the ns3 simulator). Since the RTT is the same between any pair of nodes, both the average data dissemination delay and the average data synchronization delay should be $6D$, regardless of the size of the group.[6]

Figure 4.8 shows the data dissemination delay with link delay $D = 10ms$, as the size of the group grows from 4 nodes to 10 nodes. Figure 4.9 shows the data synchronization delay under the same configuration. As is expected, VectorSync performs consistently across different group sizes, with average data dissemination and synchronization delay staying close to 60ms.

[6]Note that the data cache at the central hub node does not help reduce the delay because the notification Interest from the producer node will trigger other nodes to fetch the published data at roughly the same time.

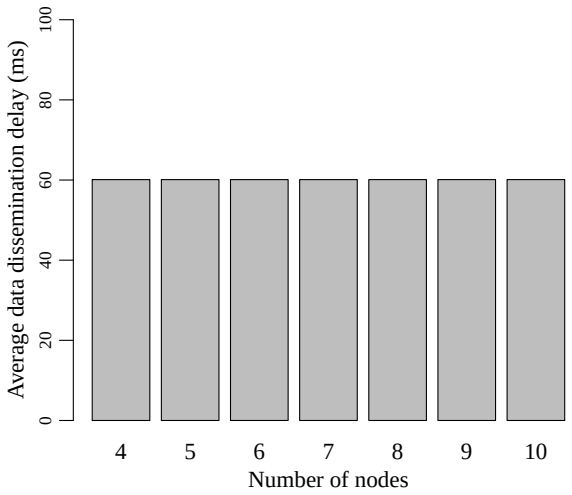

Figure 4.8: Data dissemination delay in a hub-and-spoke network with different number of nodes

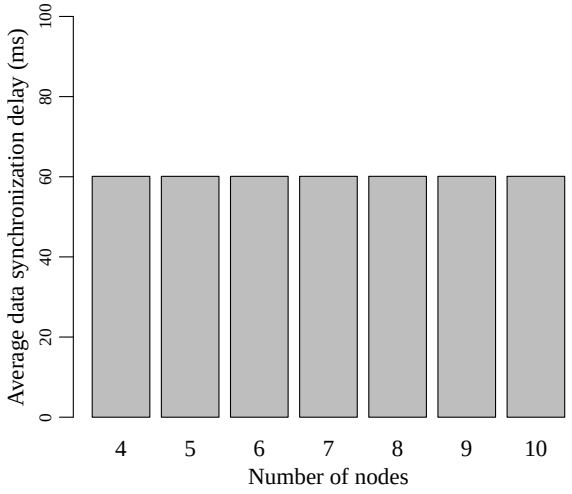

Figure 4.9: Data synchronization delay in a hub-and-spoke network with different number of nodes

Figure 4.10 and Figure 4.11 show the data dissemination and synchronization delay with 10 nodes in the group while the RTT changes from 200ms to 800ms. In both scenarios, the average data dissemination and synchronization delay grows linearly with the RTT as expected.

When a node publishes new data and sends out a notification Interest, triggering multiple

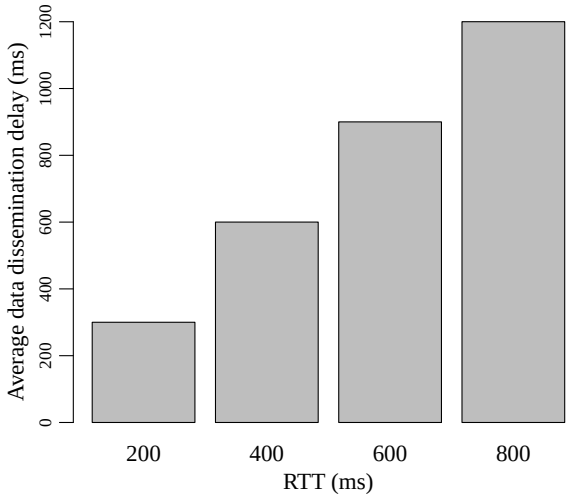

Figure 4.10: Data dissemination delay in a hub-and-spoke network with different link delays

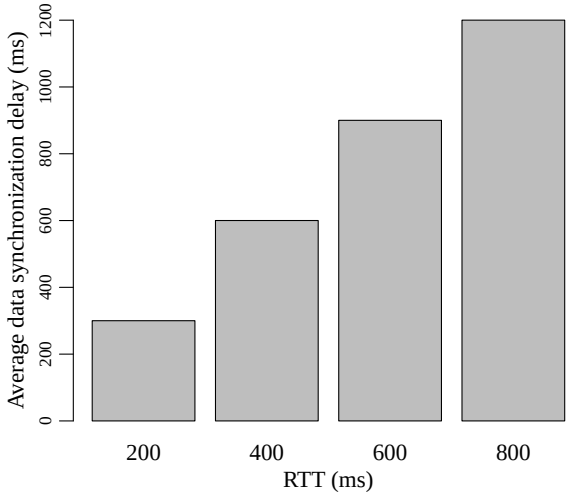

Figure 4.11: Data synchronization delay in a hub-and-spoke network with different link delays

nodes to issue Interests to fetch the new data around the same time, those Interests will be aggregated by the central hub node and only one of them (i.e., the first one arrived) will be forwarded to the producer to bring back the data. The data is then returned to all other nodes following the multicast delivery path established by the pending Interests. Therefore each Interest and the corresponding data will traverse each link in the hub-and-spoke network exactly once. The same property also holds for notification Interests and the

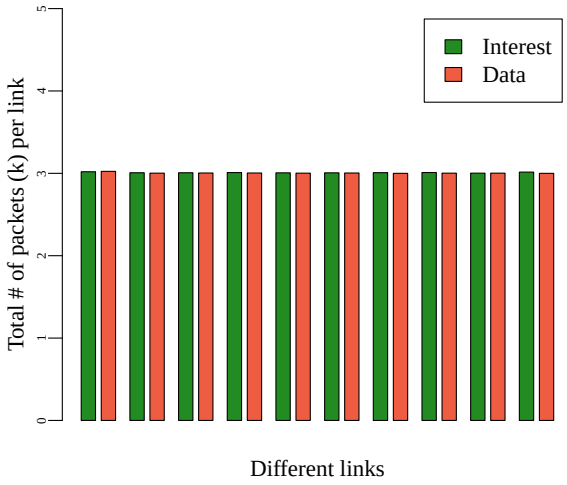

Figure 4.12: Total network traffic per-link in a hub-and-spoke network with 10 nodes

replies. Figure 4.12 shows the total amount of packets sent over each link of the hub-and-spoke network with 10 nodes in the group, which confirms our analysis.

4.4.2 Synchronization in Lossy Network

In this subsection we turn our attention to the performance of VectorSync over lossy networks. We use a hub-and-spoke network topology with 10 nodes and 10ms link delay, but configure the network device on each sync node to randomly drop received packets at a pre-configured packet loss rate. We configure VectorSync to retransmit expired Interests up to 5 times. The lifetime of both notification Interest and Node Data Interest is set to 50ms (which is larger than the RTT between any two nodes) in order to avoid premature retransmission.

We run the experiment with three different configurations of data publishing rates and heartbeat intervals. In the first configuration each node publishes with 10-second average inter-arrival time and 10-second heartbeat interval. In the second configuration each node publishes with 1-second average inter-arrival time and 10-second heartbeat interval. In the third configuration each node publishes with 1-second average inter-arrival time and 1-second

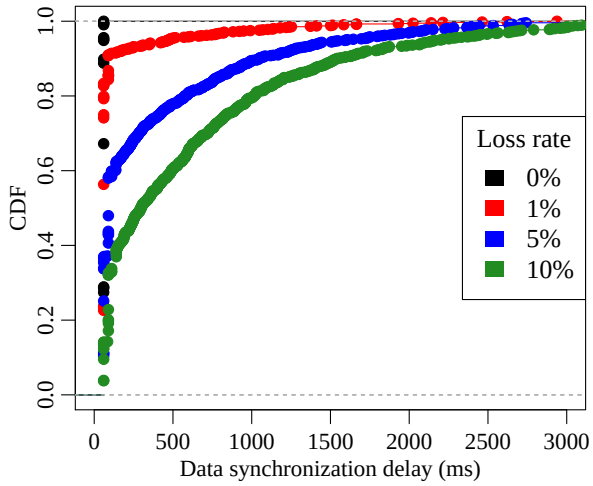

Figure 4.13: Data synchronization delay (0.1 pps data rate and 10-sec heartbeat interval)

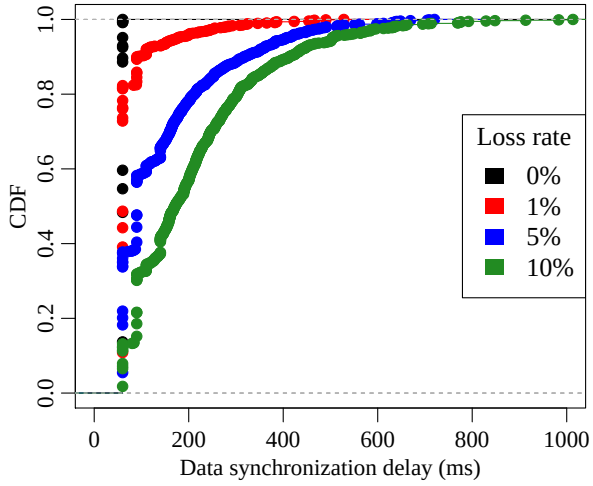

Figure 4.14: Data synchronization delay (1 pps data rate and 10-sec heartbeat interval)

heartbeat interval.[7] With each configuration we run the experiment under 0%, 1%, 5%, and 10% packet loss rates, respectively. The CDF plots of the data synchronization delay are shown in Figure 4.13, Figure 4.14, and Figure 4.15.

It is clear to see that as the loss rate increases the average data synchronization delay also increases in all scenarios. On the other hand, increasing the data rate and/or the heartbeat

[7]The running time of the experiments is adjusted based on the data rate to ensure that each node publishes around 100 application data packets in total.

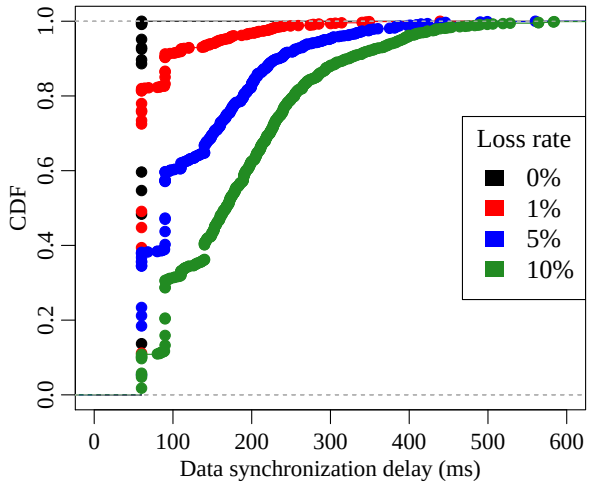

Figure 4.15: Data synchronization delay (1 pps data rate and 1-sec heartbeat interval)

rate can mitigate the impact of packet loss on the synchronization delay. This is because VectorSync enables each node to detect inconsistency in the whole dataset using the state vector carried in every application data packet and heartbeat packet published in the group. Furthermore, it allows the data producer to detect inconsistency based on the state vector carried in the notification replies, which provides additional opportunity for the nodes to recover from packet loss. However, the lower synchronization delay is achieved at the cost of generating more traffic in the network. By comparing Figure 4.16 with Figure 4.17, we can see that there is roughly a 50% increase in the total amount of network traffic when the heartbeat interval is reduced from 10 seconds to 1 second (under the same application data rate).

4.4.3 Comparison with ChronoSync

In this subsection we compare the performance between VectorSync and ChronoSync. The main reason of choosing ChronoSync for comparison is that it is currently the only sync protocol developed for the NDN architecture with mature published implementation[8]. As we have discussed in Chapter 3, ChronoSync achieves a minimum data dissemination delay of

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

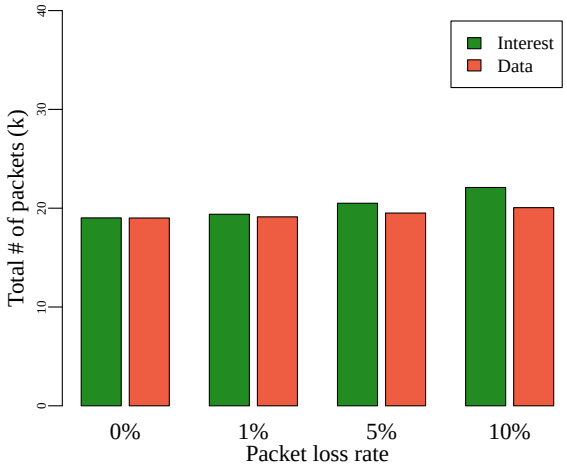

Figure 4.16: Total network traffic (1 pps data rate and 10-sec heartbeat interval)

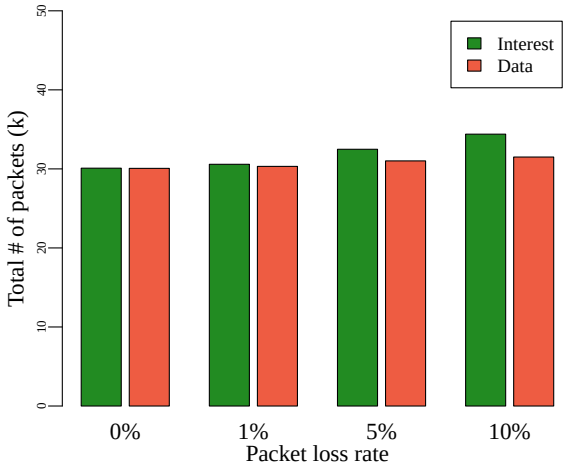

Figure 4.17: Total network traffic (1 pps data rate and 1-sec heartbeat interval)

$1.5 \times RTT$[9], but may experience longer delay if multiple nodes publish data simultaneously, 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 synchronization delay and higher

[9]Here we assume the published data is retrieved by Interest in a separate round trip rather than encapsulated in the Sync Replies, in which case the minimum dissemination delay is $0.5 \times RTT$.

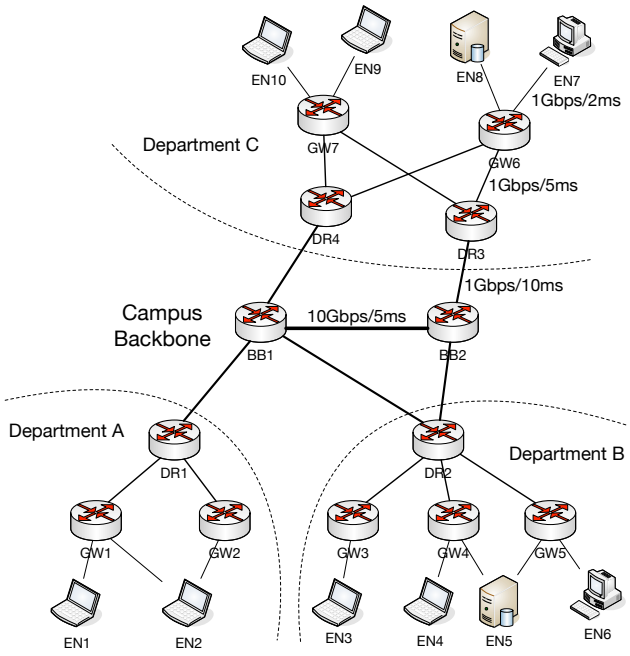

Figure 4.18: Campus network topology

traffic in the network (because the recovery Interest is sent to multicast namespace and forwarded to all nodes in the group).

To ensure fair comparison, we ported the same data publishing application used in the VectorSync experiments on top of ChronoSync. We conduct the experiments using two realistic network topologies: a small-scale campus network and a large-scale ISP network. In both topologies we vary the data publishing rate of the sync nodes[10] and the packet loss rate in the network, then measure the data synchronization delay and the total amount of network traffic for both sync protocols.

The small campus network topology used in the simulation is shown in Figure 4.18: two backbone routers (BB) interconnect the department routers (DR) from different department networks; each department network has a few gateway routers (GW) that connect end nodes (EN) to one or two department routers. Table 4.1 summarizes the bandwidth and delay of the links connecting different types of network devices. Each experiment involves the 10 end nodes (EN1 – EN10) participating in the same sync group, each publishing around 100

[10]The heartbeat interval is set to be the same as the average application data interval.

Link Type	Bandwidth	Delay
BB – BB	10Gbps	5ms
BB – DR	1Gbps	10ms
DR – GW	1Gbps	5ms
GW – EN	100Mbps	2ms

Table 4.1: Bandwidth and delay for different types of links in the campus network topology

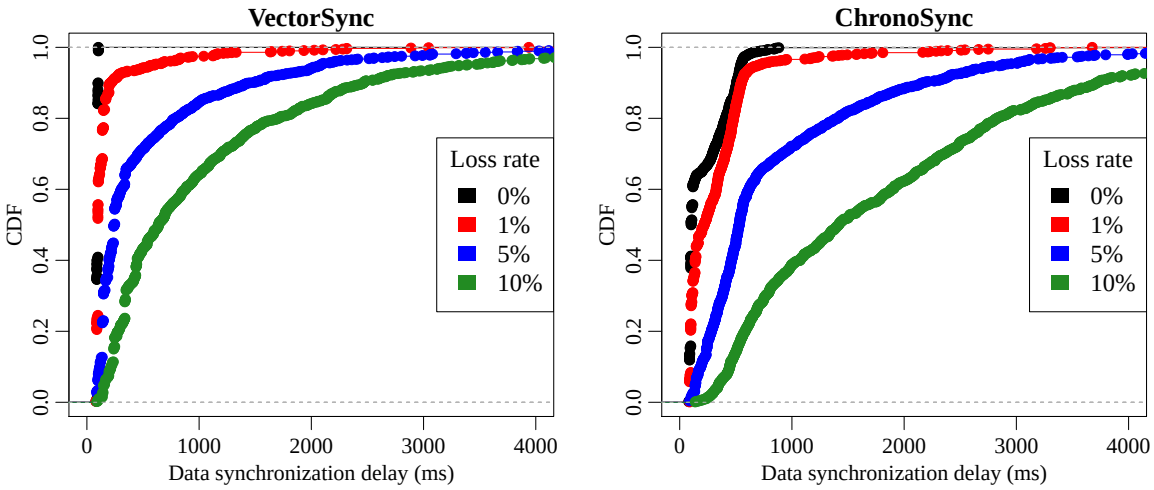

Figure 4.19: Data synchronization delay in the campus network (data rate = 0.1pps)

application data.

We first compare the data synchronization delay under various packet loss rates with each node publishing data at 0.1 pps on average. Figure 4.19 shows the CDF plots of the data synchronization delay in the campus network for VectorSync and ChronoSync. We can see that even if 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 that causes 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.

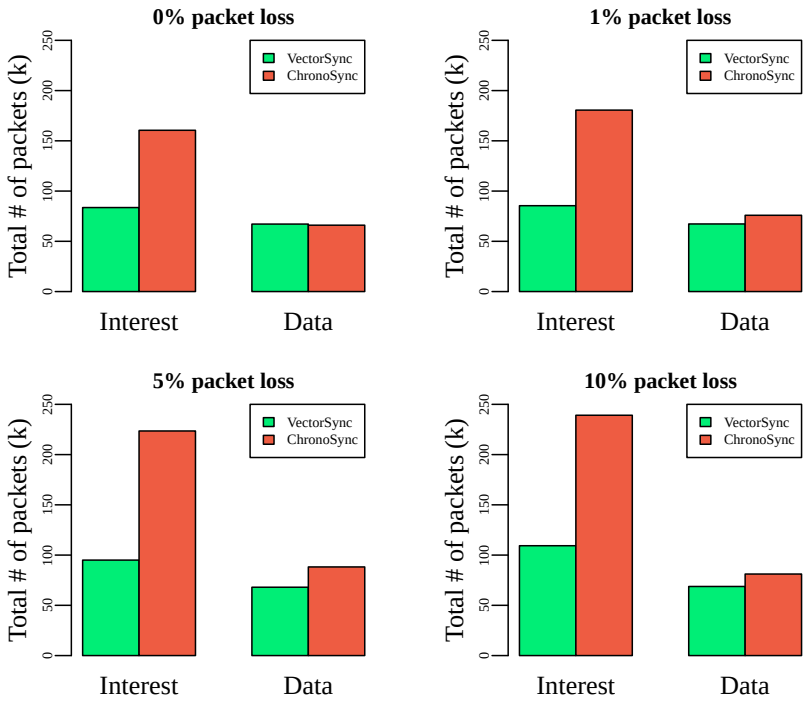

Figure 4.20: Total number of packets transmitted in the campus network (data rate = 0.1pps)

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 entire campus network during the experiment, which is shown in Figure 4.20. 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 diverging states also contribute to the high number of Interests. On the other hand, in VectorSync the Interest traffic volume shows only slight growth as the packet loss rate increases, due to the retransmission of expired Interests.

We repeated the same experiments with higher per-node data rate (1pps), which causes lots of simultaneous publishing that leads to conflicting states in the sync group. Figure 4.21 shows the results of data synchronization delay for VectorSync and ChronoSync. As we can

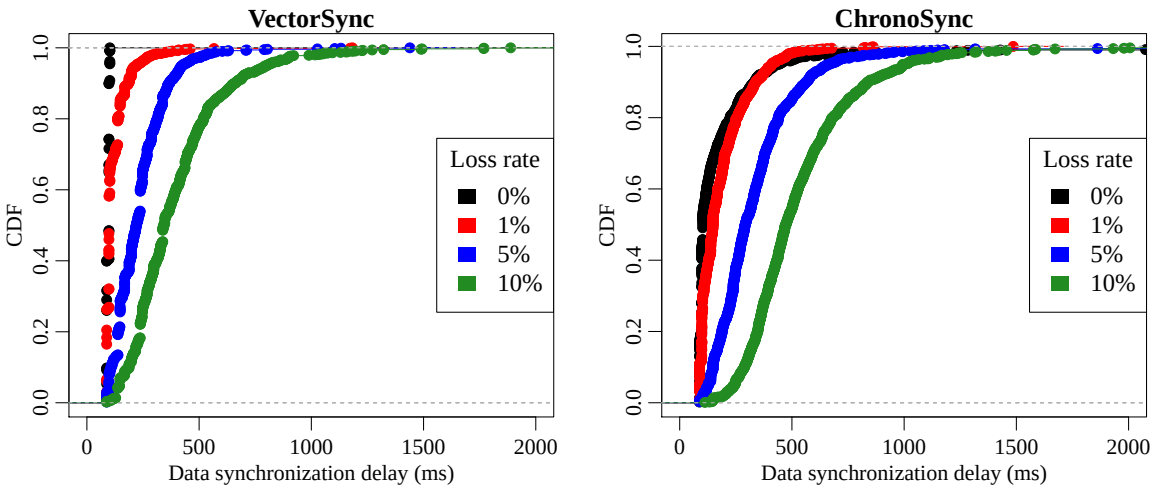

Figure 4.21: Data synchronization delay in the campus network (data rate = 1pps)

see, increasing the data rate in the sync group enables VectorSync to recover from packet loss faster since the state vector carried in each data enables detection of missing data published by all group members. When there is no packet loss, ChronoSync exhibits significantly higher synchronization delay compared to VectorSync due to simultaneous publishing. However, as the packet loss rate increases, the synchronization delay of ChronoSync becomes similar to that of VectorSync. This is because ChronoSync invokes the recovery mechanism frequently under high data rate, and the recovery data carries the full ChronoSync state which is similar to the state vector in VectorSync. This allows ChronoSync to reconcile the dataset state in a similar way as VectorSync does, which leads to similar synchronization delay (especially under higher packet loss rates).

The key difference between the ChronoSync recovery mechanism and the VectorSync dataset state synchronization is that the recovery Interest is sent to every node in the group via multicast, and the recovery process is invoked in additional to the normal synchronization process in ChronoSync. This causes ChronoSync to generate much higher amount of traffic in the network, as is shown in Figure 4.22.

We conducted similar experiments using a larger ISP network topology generated from real-world measurements [SMW04], which is shown in Figure 4.23. We randomly pick 10 nodes in the network to participate in a sync group and publish data at 0.1pps on aver-

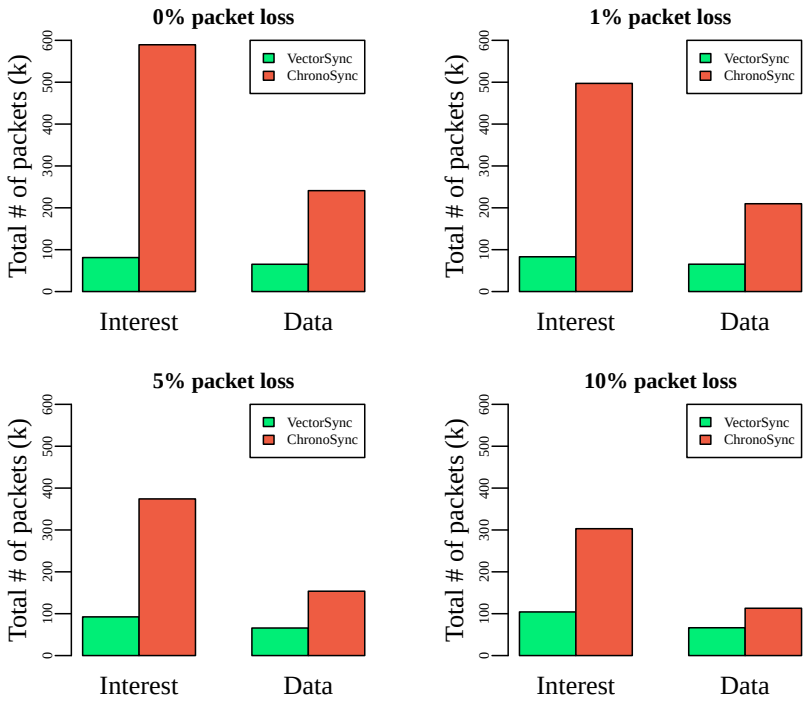

Figure 4.22: Total number of packets transmitted in the campus network (data rate = 1pps)

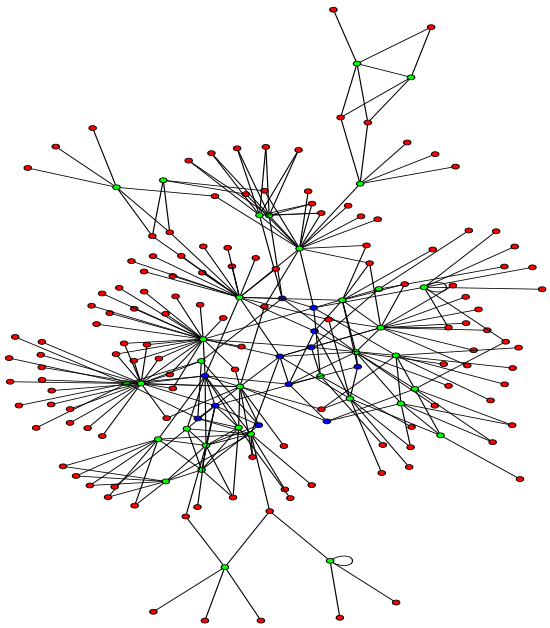

Figure 4.23: Large ISP network topology

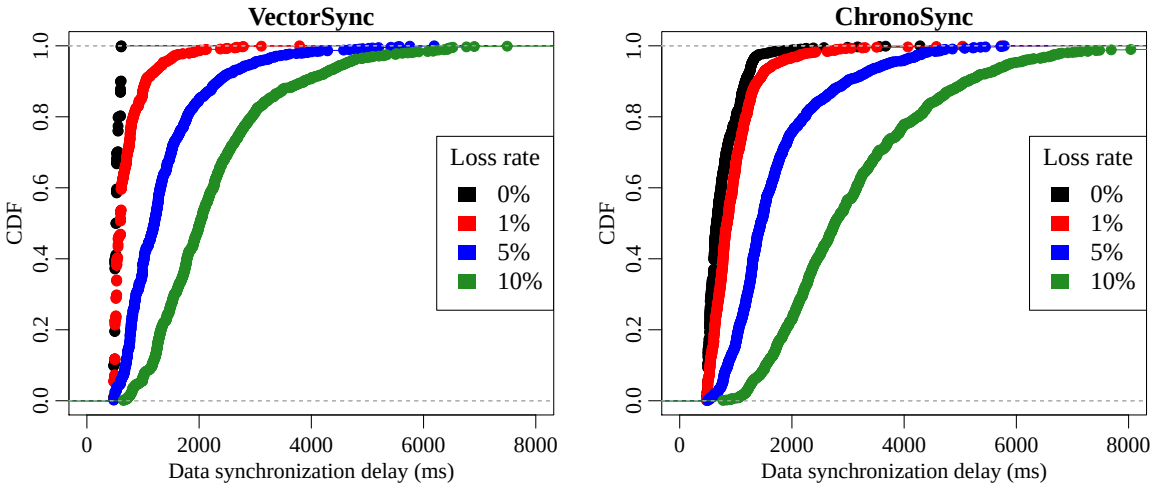

Figure 4.24: Data synchronization delay in the large ISP network (data rate = 0.1pps)

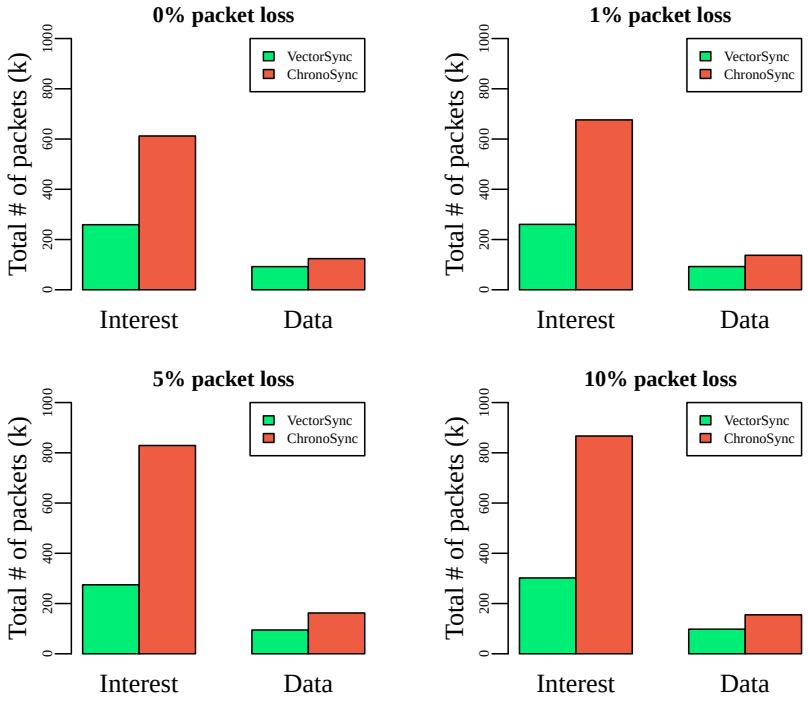

Figure 4.25: Total number of packets transmitted in the large ISP network (data rate = 0.1pps)

age. To simulate packet loss, we configure *all* nodes in the network to randomly drop the received packets at a pre-configured error rate. Figure 4.24 and Figure 4.25 show the data synchronization delay and the total number of packets transmitted during the experiment,

respectively, under different packet loss rates. Both results are consistent with the previous measurements based on the campus network topology.

4.4.4 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 in a sync group leaves the group (by shutting down the data publishing application and the VectorSync module) at a randomly selected time point, and compare the data synchronization delay to the scenario where there is no membership change throughout the experiment. We carry out the experiment on all three network topologies that have been used so far (hub-and-spoke network, campus network, and large ISP network) under different packet loss rates and data publishing rates, and the results are similar in all scenarios. Here we report only the results and analysis from the experiments over the large ISP network (with 0.1pps per-node data rate) to highlight the key insights we obtained.

Figure 4.26 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 synchronization delay is mostly unaffected by the view change processes during the experiment 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 all notification Interests, which allows the nodes to detect and synchronize their views quickly after the view change starts. Third, VectorSync utilizes a deterministic leader selection algorithm that allows the group to pick a new leader as soon as the current leader leaves, which also improves the view synchronization speed.

The main reason for the noticeable increase in the synchronization delay under 10%

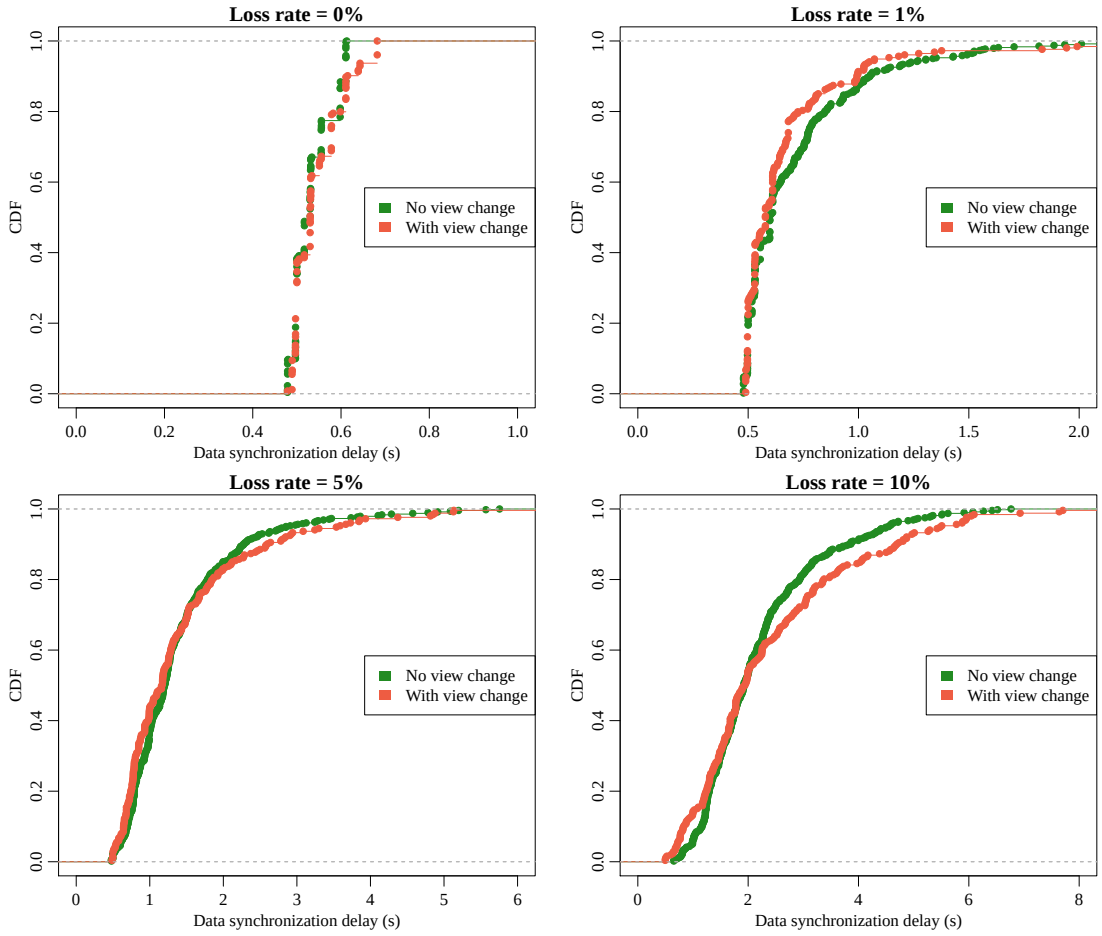

Figure 4.26: Data synchronization delay in the large ISP network with dynamic membership changes (data rate = 0.1pps)

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 nodes in the network to fetch the published data.

CHAPTER 5

Building High-level Services over VectorSync

Maintaining explicit group membership information enables VectorSync to support protocols and algorithms in distributed systems that require collecting information from all participating members. In this chapter we describe the design of two sync-related services, dataset snapshot and data ordering, on top of VectorSync. Both of them offer important functionality that is frequently used in distributed applications.

5.1 Dataset Snapshot

An important design choice made in VectorSync is that the state vectors generated in a view contain only the sequence numbers from the current members in that view. This allows VectorSync to effectively remove the departed nodes from the state vector, preventing the size of the vector from growing unbounded as nodes come and leave in a long-running application session. Consequently, the new nodes that join the group late will not learn about the data published by the departed nodes from the state vectors of the current and future views (unless a departed node rejoins the group later). While this design is suitable for applications that require synchronization among the active members only (e.g., resource discovery and routing protocol), some application may also require preserving the historical data, including the data 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 *dataset snapshot* service on top of VectorSync to generate snapshots that capture the dataset state at the end of each view throughout the history of the group.

The snapshot process is triggered by the view change events. After joining a new view,

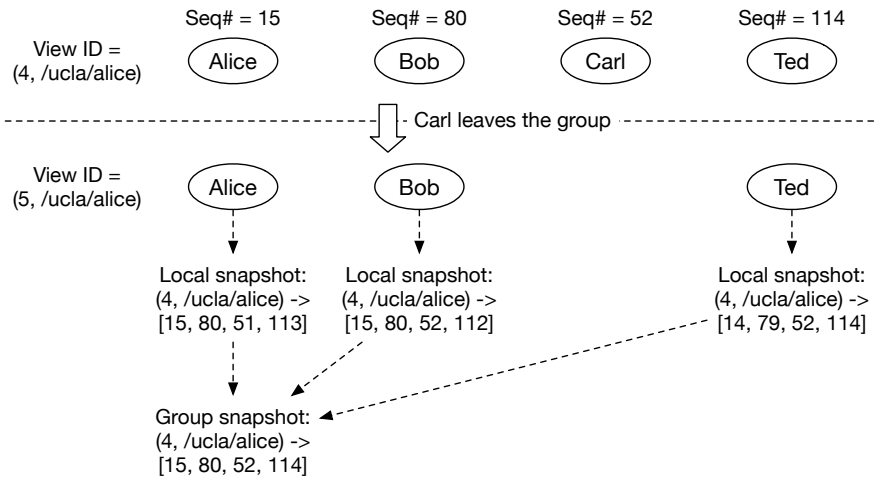

Figure 5.1: Example of generating a group snapshot after a view change

each node in the group records and publishes its local state vector of the previous view (together with the view ID) in a data packet called *local snapshot*, which represents the dataset state observed by the node at the end of the previous view. The local snapshot packets are published in the shared dataset and propagated to other group members via VectorSync. The leader of the new view is responsible for collecting local snapshots from all current members in order to compute the *Join* of the state vectors reported in those local snapshots which summarizes the knowledge of the dataset state of the previous view.[1] The leader then publishes the "joined" state vector and the corresponding view ID in a data packet called *group snapshot* and notifies other nodes that a group snapshot has been generated, upon which the nodes can remove the recorded state vector (since it has been covered by the group snapshot). Figure 5.1 illustrates the snapshot process with a simple example. To permanently store the historical data, the dataset snapshot service requires a stable storage component (e.g., a repo) to collect the data published in the shared dataset based on the group snapshots and the corresponding ViewInfo.[2]

[1]Note that if a node crashes immediately after publishing some data, the information about that data will not be retained in the group snapshot since none of the remaining members in the group knows about such data.

[2]As an optimization, the leader may expand the state vectors into version vectors (i.e., annotating each entry in the vector with the corresponding node name) when it publishes the group snapshot.

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 using the following naming convention: "/[group-prefix]/snapshot/[view-number]/[leader-name]", and the content carries the ID of the previous view. Therefore the group snapshot packets published over time essentially record a chain of view IDs, which links successive views and can be used to trace the history of view changes. Second, when group partition happens, each sub-group will generate its own group snapshot for the previous view. If a node wants to fully recover the dataset state for that view, it would need to trace the branches in the view change history to obtain all published group snapshots. Third, after the group partition heals, the nodes that belonged to different subgroups will publish local snapshots for different views. Consequently, the generated group snapshot will contain multiple state vectors, one for each unique view reported in the local snapshots. Finally, calculating the group snapshot requires collecting state vectors from all members in the current view. If a node crashes before it publishes its local snapshot, the leader will not be able to generate a group snapshot until it removes the dead node and creates a new view. As a result, the remaining nodes need to keep the state vectors from all previous views not covered by any group snapshot and report all of them in the local snapshots published in the future. Figure 5.2 shows an example of different group snapshots generated over the history of multiple view change events.

5.2 Data Ordering

Like other existing NDN sync protocols, VectorSync provides eventual consistency which guarantees that all nodes sharing the same distributed dataset will eventually receive all the data published by each other, as long as there is no permanent node or network failure. During temporary group partition, the nodes may continue publishing data (if the application semantics permits) which will be propagated to other nodes by the sync protocol after the partition heals. This weak consistency model is sufficient for synchronizing *unordered* dataset (e.g., replicating the data collections among the repos).

However, many distributed applications require some data ordering property to be pre-

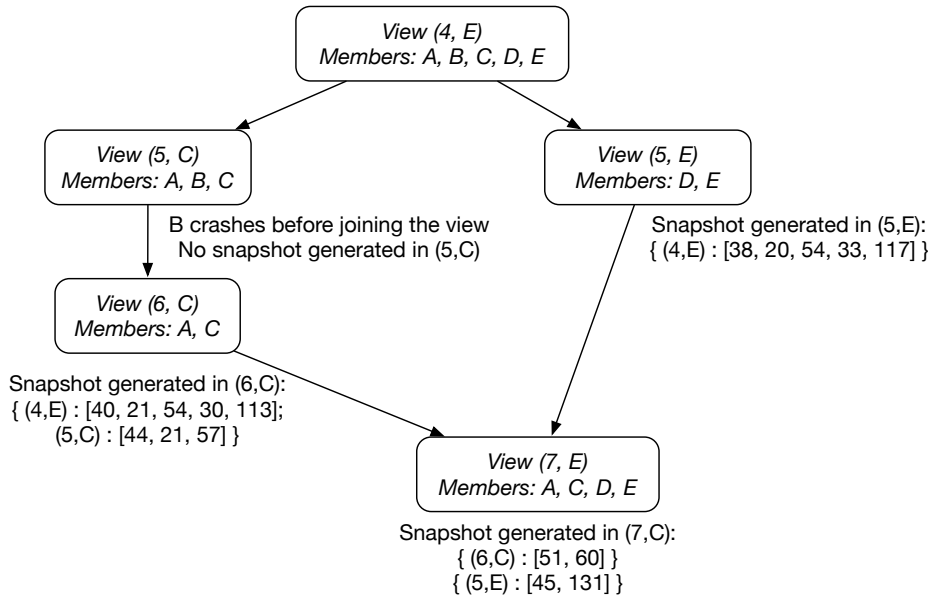

Figure 5.2: Example of dataset snapshots generated in different views over a group's history

served while the data is published and synchronized in the distributed system. For example, a group chat application may require the messages from each participant to be displayed in their publishing order in the chat window; moreover, it is beneficial if the order of the chat messages from different participants can also preserve their causal relations: if message A was generated in reply to message B, the participants in the chat room should observe message B before A in order to avoid confusion. Some applications such as distributed databases may further require a consistent total ordering of all data published in the system (e.g., database transactions issued by the clients from different server replicas). Satisfying those application-layer requirements would require data ordering support to be built on top of the eventually consistent sync protocols.

To support different data ordering models over VectorSync, we take a layered approach that decouples the synchronization of the shared dataset from the enforcement of ordering constraints. The eventually consistent VectorSync protocol ensures that data published by any active node in the group will be delivered to every other node in the sync group. On top of the core protocol we can build a shim layer that delays the notification of new data to the application until the ordering constraints are satisfied. This separation of responsibility

allows VectorSync to support multiple data ordering models to fit the semantics of various applications.

VectorSync can easily support per-node FIFO data ordering thanks to the sequential data naming convention adopted by the protocol. The data ordering layer buffers the out-of-order data packets in a per-node FIFO queue and delays the notification of a data packet from a node with sequence number S until all data published by that node with sequence numbers up to $(S-1)$ has been received and consumed by the upper layer. Causal ordering can also be supported by utilizing the state vector piggybacked in each data packet, which essentially serves as a *vectorstamp* that provides a causal order over the data publishing events in the group. Tagging data with vectorstamps is a well-known method that has been utilized in many distributed systems such as Amazon DynamoDB [DHJ07]. If two data packets are published in different views and carry incompatible state vectors, the causal ordering can be resolved by using the view IDs (also piggybacked in the data) as a logical clock [Lam78]: the data published in an earlier view (ordered by the view ID) is causally ordered before the data published in a later view. Similar to FIFO ordering, the causal ordering layer delays the notification of a new data packet until all other data that is causally ordered before the new data has been received and consumed by the upper layer.

Many algorithms and protocols have been developed to solve the total ordering problem in a distributed system. Traditional solutions usually achieve total ordering by picking a stable master node to order all the messages and replicate them to the slave nodes in the system via two-phase commit (2PC) or consensus algorithms [Lam98, Lam01]. While it is possible to implement the classic 2PC or consensus protocols over VectorSync, our design of the total ordering module employs a simple and elegant algorithm proposed by Lamport using totally ordered logical clocks [Lam78] to ensure the data published in the group is *committed* in a consistent total order by all participants. The key idea behind this algorithm is that a node can commit a data packet with logical clock LC only if it has committed all data packets with logical clock smaller than LC from all other nodes in the group. The original algorithm assumes in-order delivery of messages over point-to-point links between any pair of nodes. In the rest of this section we describe the design detail of the total ordering

mechanism over VectorSync that adapts the original algorithm to the NDN environment.

Each node maintains a logical clock (independent from the view IDs and the data sequence numbers) which contains a monotonic-increasing counter c and the node name N. A total ordering on the logical clocks is defined as follows:

Definition 5.2.1. Given two logical clocks (c_i, N_i) and (c_j, N_j) from nodes i and j, respectively, $(c_i, N_i) < (c_j, N_j)$ if either (1) $c_i < c_j$ or (2) $c_i = c_j$ and $N_i < N_j$.

Every data packet published through the total ordering layer carries the value of the producer's current logical clock as its timestamp. When a node receives an application data packet, it advances the counter in its logical clock to be one higher than the counter in the timestamp of the received data (unless its logical clock is already higher than the received one), and immediately publishes an acknowledgement (no-op) data packet timestamped with the new value of its logical clock.[3] Both application data and acknowledgement data are published in the shared dataset and synchronized to other nodes via VectorSync. All data packets, including remote and local data, are placed temporarily in a local message queue in the timestamp order and wait to be committed by the application. Note that the application do not commit a local data packet immediately after publishing it because there could be data packets from other nodes with smaller logical clocks that have not been committed or received yet.

Before committing a data packet in the FIFO queue with timestamp (c, N), a node needs to receive, from each node i in the current view, at least one data packet D_i (possibly an acknowledgement) with timestamp greater than (c, N) and all data published before D_i by node i (i.e., with smaller sequence numbers).[4] In addition, the node needs to commit all received data with timestamp smaller than (c, N) (i.e., the data to be committed must be at the head of the message queue). These two conditions (adapted from Lamport's original algorithm) ensure that all data packets published in the group with logical clock smaller than (c, N) have been received and committed. Once the data is committed by the application

[3]The nodes do not generate acknowledgement for the acknowledgement packets.

[4]For node N, only the data published before (c, N) needs to be received.

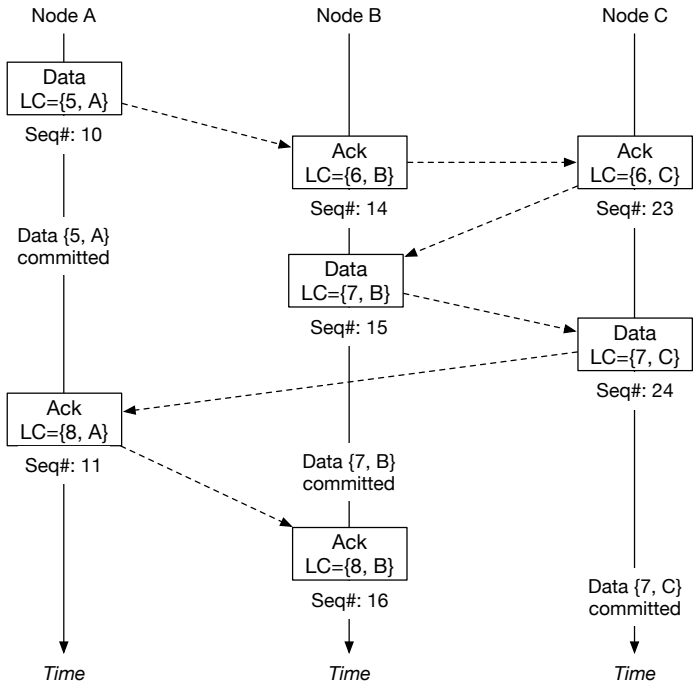

Figure 5.3: Example of publishing and committing data in total order

layer, it is removed from the message queue. Figure 5.3 shows an example of publishing and committing total-ordered data in a sync group with three nodes. Note that the order in which the data is committed is consistent with the total order of the logical clocks of the data, which is indicated by the dashed arrows in the figure.

Like the dataset snapshot service described earlier, the data total ordering algorithm requires active participation from every node in the group. If one or more nodes fail, they need to be removed from the group through the view change process before the applications can commit any more data.

CHAPTER 6

New Applications of NDN Sync in Internet of Things

In this chapter we discuss the applications of sync protocols (including VectorSync and several other existing sync protocols) in the emerging application area of "Internet of Things" (IoT) where the ICN architecture is widely accepted as a key enabler for more powerful applications and services than what is available in the current TCP/IP architecture. We first give an overview of how NDN brings fundamental change to the IoT network and application architecture compared to TCP/IP. Then we describe a few pilot IoT applications developed on top of the NDN architecture that rely on various sync protocols to support multi-party communication and achieve critical functions that are often challenging, if not impossible, to implement using the traditional TCP/IP protocol stack.

6.1 Named Data Networking of Things

(The content of this section is adapted from the previous publications [SBL16] and [SWA17].)

The Internet of Things (IoT) vision proposes to interconnect *things* of all kinds by leveraging the proliferation of increasingly small and affordable embedded devices for processing, sensing, actuation, and wireless communication. The global realization of this vision will easily exceed the scale of devices and data objects found in the current Internet by orders of magnitude [WSJ15]. However, the roll-out of IoT faces two fundamental and often conflated challenges. The first is how to enable all different types of digital *devices* that provide IoT functionality to communicate locally and globally. The second is how to consistently, securely communicate the data associated with the *things* themselves, once connectivity is achieved. The latter is the heart of the IoT vision, providing access to everything from door

lock status and lighting levels in home automation to the flow of water measured by a municipal meter in a smart city, an individual's blood-glucose level, and the soil pH measured across a field by a truck-mounted sensor.

Current TCP/IP-based IoT systems and frameworks focus on interconnecting devices, primarily addressing the first challenge. Building up from the host-to-host communication paradigm of IP, these frameworks conflate the embedded devices with their associated real-world *things* at the network layer. They tend to emphasize solutions for device-to-device connectivity and then meet the applications' need of accessing the associated real-world data through a series of mappings. To fetch data about a *thing* itself, a typical application process may have to traverse a long series of mappings among interface addresses, devices, channels, and subnetworks, each of which must be secured. Such mappings add complexity and brittleness to what are often simple communications of sensor data, actuation commands, and configuration operations. For example, consider a light (a *thing*) in a contemporary building automation and management system. To control its intensity, an application must be able to get packets to the appropriate VLAN and IP subnet, as well as know the lighting gateway device's IP address and protocol, before dealing with the light itself via an application-layer identifier. While consumer devices have made this easier, often allowing Web-based control over IoT devices from the operator's laptop or mobile phone, they do so by requiring complex (and often manual) configurations to specify the mappings between identifiers at different layers in the network stack[1], or relying on cloud services to achieve what is essentially local communication [SWA17].

By naming and securing the *things* and data directly at the network layer, NDN is able to provide a more straightforward and secure solution to IoT networking as compared to TCP/IP:

- The Interest-Data exchange model in NDN closely resembles the RESTful protocols such as HTTP and CoAP that are widely adopted in today's IoT systems. In monitoring and measurement applications, clients can issue Interests to retrieve named sensor

[1]Certain level of auto-configuration may be supported by making assumptions about such mappings, for example, that all devices are on the same subnet.

data over the NDN network. In actuation applications, controllers can use Interests to express the actuation commands, with the Interest names identifying the object and what needs to be done to the object, e.g., "/LivingRoom/Lighting/OFF".

- Name-based forwarding simplifies the network stack by removing the extra step of resolving application names to network identifiers (e.g., IP and MAC addresses). The IoT devices advertise and discover application names directly at the network layer. The stateful forwarding plane allows fine-grained control and adaptation of forwarding decisions at each node, adapting to network connectivity changes.

- Data-centric security is more efficient and IoT-friendly than the channel-based or physical/logical isolation-based alternatives. By securing the named data directly, NDN enables IoT data to traverse boundaries between heterogeneous network environments without losing security properties. It is also possible to store data in an application-transparent manner in in-network caches and persistent data storage. NDN allows IoT applications to freely distribute data to any place in the network without requiring them to trust any intermediate node to keep data intact and confidential.

- Ubiquitous in-network data caching and permanent storage (i.e., repos) helps improve the efficiency of information dissemination, especially for resource constrained IoT environments. For example, sensors with limited storage deployed in an agricultural field can transfer monitoring data immediately after its acquisition to a nearby repository. A remote controller can later retrieve this data from the repository, more effectively using available bandwidth and consuming less energy. In typically disconnected environments, "data mules" can carry Data packets in their in-network storage, enabling data to be diffused even when consumers and producers never have a directly connected channel between them.

6.2 Sync in IoT Networks

NDN sync can be utilized to achieve many important functions in the IoT environments that are often challenging to implement in the traditional TCP/IP architecture. In this section we describe three sync-based IoT applications in order to demonstrate the power and usage of sync. First, we describe how to use sync to provide resource discovery and rendezvous service in smart home environments without relying on centralized servers or remote cloud services. Second, we describe how to use sync to improve the data availability and facilitate information dissemination in constrained IoT mesh networks where the devices need to go to sleep mode periodically to conserve energy. Last, we introduce a distributed publisher-subscribe (pub-sub) framework built on top of multiple sync protocols to support pub-sub communication in enterprise building management systems (BMS).

6.2.1 Resource Discovery in Local Environments

IoT applications often benefit from auto-discovery of resources in the network, which allows the devices and services to discover and interact with each other without human intervention (i.e., machine-to-machine communication). For example, it is desirable for a newly installed light switch to automatically discover and associate with the light bulbs in a smart home or smart building without manual configuration from the users, since there could be a large number of those devices in the same network and those devices may not even provide a user-friendly configuration interface. In TCP/IP-based IoT systems, resource discovery is typically achieved by deploying centralized servers (e.g., CoRE Resource Directory [BSS17] and DNS-SD [CK13a] servers) in the local network or offloading the device management and discovery task to the remote cloud service such as AWS-IoT[2]. However, both solutions have major drawbacks that affect the usability and reliability of the IoT system: the resource discovery servers can easily become a single-point-of-failure and require additional management effort from the users, while the cloud service introduces external dependency to the local communication in the IoT system which becomes subject to the failures in the cloud

[2]https://aws.amazon.com/iot

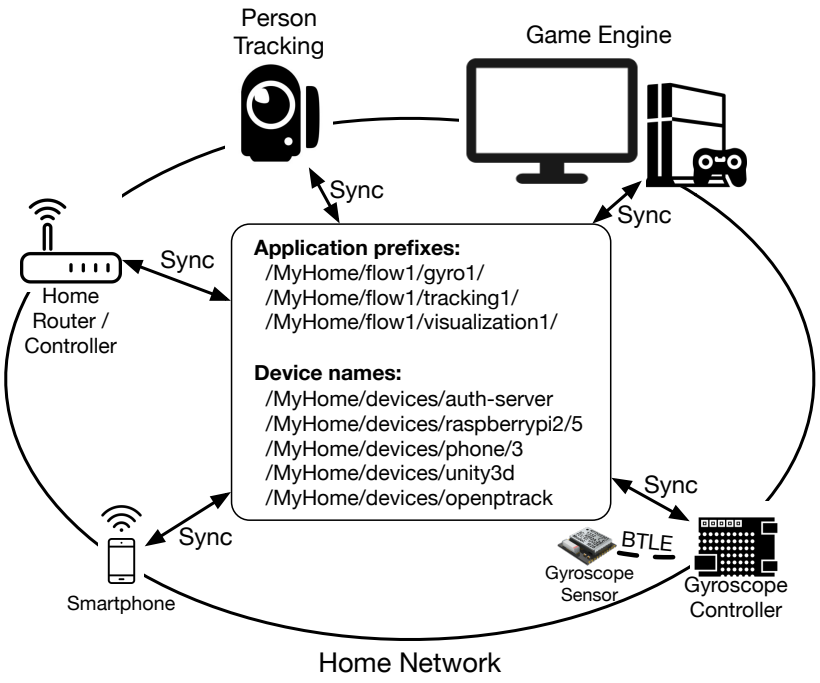

Figure 6.1: Synchronization of the "discovery" dataset in Flow home entertainment system

platform itself [Ama17] or the connection to it through the public Internet.

NDN sync enables a robust and distributed design of resource discovery as a local service running in the IoT network without external dependency. The shared dataset maintained by the sync protocol essentially becomes a rendezvous point where multiple IoT devices and services can publish information about their resources (identified by the NDN names) and obtain the up-to-date knowledge about other resources available in the system. One of our pilot NDN-IoT applications called "Flow" [SWA17], a home entertainment system implemented on top of NDN network, utilizes sync to implement resource discovery locally in a smart home environment by requiring the local devices to publish the list of device names and application prefixes in a shared "discovery" dataset. The IoT devices can join the dedicated "discovery" sync group to synchronize the "discovery" dataset using ChronoSync and learn about the resources available in the home network. Those who wish to provide services to other devices also join the sync group and publish their device or application prefixes in the "discovery" dataset. Figure 6.1 illustrates the synchronization of the "discovery" dataset in a Flow instance deployed in a smart home network.

The sync-based discovery service generally achieves higher reliability, efficiency, and security for information discovery, and provides more flexibility in supporting the application-specific semantics, compared to the general network-layer discovery mechanisms such as flooding and self-learning. Note that the mDNS [CK13b]-based DNS-SD service also tries to achieve resource discovery in a distributed fashion over the IP networks. However, it merely utilizes IP multicast as a rendezvous channel where the devices in the same local network can send DNS queries and receive answers. Although the devices may cache the answers and reply to similar requests on behalf of the resource origin, it does not *synchronize* the devices to provide up-to-date knowledge about the resources in the local network. Furthermore, the security issues such as the authentication of the DNS records are left unaddressed by the mDNS standard and must be resolved through additional higher-level protocols.

6.2.2 Improving Content Availability

Just like traditional Internet applications use synchronization protocols to replicate data across multiple nodes for fault tolerance, the IoT applications can also leverage sync to replicate the sensor data on multiple devices in order to improve content availability. This is particularly useful for constrained environments where the IoT devices need to go to sleep mode periodically in order to conserve energy. If the IoT data is stored solely on the producer device itself, the data will become unavailable during the producer's sleeping period. In contrast, if the data is replicated cross multiple devices and those devices to not exhibit synchronized sleeping schedule, a data request is more likely to get replied by one of those devices at any time, even when some of those devices are offline.

To explore this idea, we have recently started a new research project aiming at designing a scalable and robust forwarding scheme for wireless IoT mesh networks.[3] The key idea is to leverage an existing geo-forwarding mechanism (originally proposed for NDN-based vehicular networks [WAK12]) to support NDN communication over a large-area IoT mesh network, and run VectorSync among devices within the vicinity of each other to improve the

[3]This ongoing work is in collaboration with Xin Xu, a visiting student at UCLA IRL lab.

availability of IoT data. NDN sync is session-less and based on representing the knowledge of each participant about the shared dataset. This makes it suitable for disseminating information in disruptive environments where the network exhibits intermittent connectivity due to sleeping nodes and/or wireless interference.

The proposed mesh forwarding scheme targets application scenarios that consist of stationary and energy-constrained devices communicating with each other in a mesh network over low-energy wireless technology such as IEEE 802.15.4. The entire mesh network is divided into multiple regions, each identified by a 2D or 3D geo-coordinate that represents the physical location of that region. Each device is configured with its location information (i.e., which region it belongs to) at installation time so that they do not need to be equipped with GPS interface (which is typically energy-inefficient). We require the mesh network to have certain density so that each region contains multiple devices that are within the coverage of each other's wireless signal and can directly communicate with each other in one hop over the wireless channel. To support cross-region communication, we adopt the existing geo-forwarding scheme where the devices farther away from the previous hop and closer to the destination region transmit the Interests sooner than other devices in order to disseminate the packet faster. However, different from the previous work, our new design applies geo-forwarding only at the level of regions. Once the Interests reach the region where the data producer resides, they are propagated to the nodes within that region via the one-hop wireless channel.

To increase the availability of sensor data in the presence of sleeping nodes, the devices in the same region form a VectorSync group and synchronize the dataset that contains the most recent data published by all the devices in that region (subject to the storage limit of each device). Both the group sync prefix and the data publishing prefix of the devices in the group carry the geo-coordinate of the region. When an Interest requesting the data published in that region arrives, any device who is currently awake and has a local copy of the requested data can reply to the Interest.[4] Figure 6.2 shows an example of fetching the

[4]The device also adds a random delay before sending the Data packet so that it can suppress its own reply if another device in the region has already replied to the same Interest.

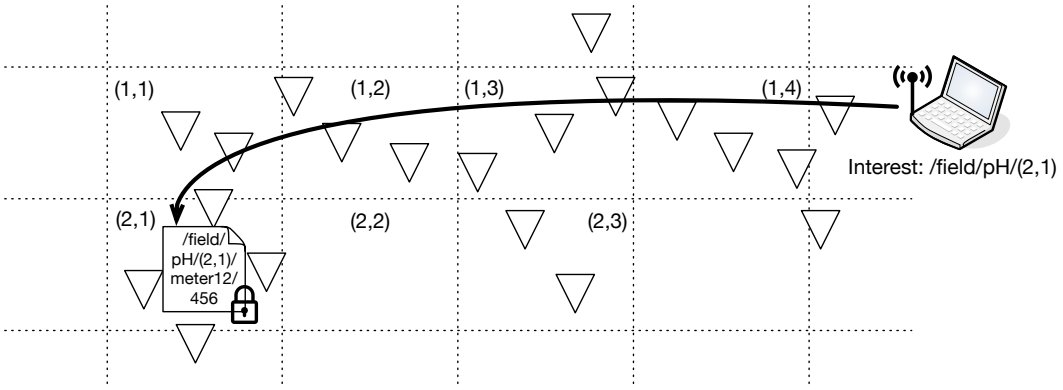

Figure 6.2: Fetching replicated data from a target region via geo-forwarding over a wireless mesh network

soil pH data via geo-forwarding from certain region of an agricultural field with the soil pH meters participating in a wireless mesh network and replicate the data within each region through VectorSync. The proposed design also includes a distributed scheduling algorithm to make sure at least *some* node is awake in each region to reply to data requests.

6.2.3 Pub-sub Communication in Building Management Systems

Publish-subscribe (pub-sub) is a common communication paradigm adopted by many IoT applications: sensors "publish" their readings as it is generated, while aggregators, analytics engines, and actuators "subscribe" to the data sources of interest to receive notifications of new sensor data. It is a common misconception to confuse NDN's basic Interest-Data exchange model with the pub-sub pattern. The core NDN architecture implements a pull-based request-response paradigm. To ensure flow balance at the network layer, it does not directly provide persistent subscriptions with publisher-initiated communication of new data. However, the effect of push-notification can be achieved at a higher level via the sync protocol, where the subscribers participate in the sync group and consume the publishers' data propagated by the sync protocol. In many pub-sub scenarios the subscribers are interested in only a subset of the entire dataset generated by the publishers. The PSync protocol can be utilized to support "partial" synchronization efficiently between producers and consumers.

To demonstrate the feasibility of sync-based pub-sub communication, we designed NDN-

PS [SZA17], a pub-sub communication framework for transporting data streams produced by the sensors in building management systems (BMS). The core functionality of a building management system is the production and consumption of sensor data. As such, a major challenge we face is how to enable consumers to receive sensing data of their interest in real time over the network. Note that (1) each consumer may be interested in a different subset of the sensors; (2) sensors and consumers may not be online at the same time; and (3) the number of sensors and consumers can potentially be very large. NDN-PS builds on top of our earlier work NDN-BMS [SDM14] to address the challenges in BMS data consumption. Each sensor's data points form a *data stream*, which is published under an NDN name prefix by the smart panel that the sensor attaches to. To overcome intermittent connectivity and resource (e.g., CPU, storage, and energy) limitations, the published sensor data is stored in a nearby repo for archiving and access. In order to provide redundancy and efficiency of data retrieval, sensor data is replicated in multiple repos using ChronoSync. Consumers can then retrieve the data from any of the repos (or the in-network cache). The repos serving a particular replicated dataset and consumer applications interested in that dataset form a *pub-sub group*.

Consumer applications in a pub-sub group may subscribe to any subset of the BMS data streams that are identified by the stream name prefixes, and receive notifications from one of the pub-sub repos about the newly published data in their subscribed data streams. The PSync protocol can be readily used to support the communication between the consumers and the pub-sub repos. For example, a pub-sub group may generated data under the prefix `/Company/Building1/Electricity`, where each pub-sub repo stores data streams with prefixes of the form `/Company/Building1/Electricity/<panel>/<device>/<metric>/`. Suppose a consumer is interested only in Panel 2's data, it can subscribe to that panel's name prefixes and communicate with the repo using PSync to get notification whenever there are new data points published under those name prefixes. Based on the notification, the applications can then make local decisions of whether to retrieve the data, which can be done through regular NDN Interest-Data exchanges. The resulting system architecture is similar to the distributed pub-sub system in TCP/IP networks such as Apache Kafka [KNR11],

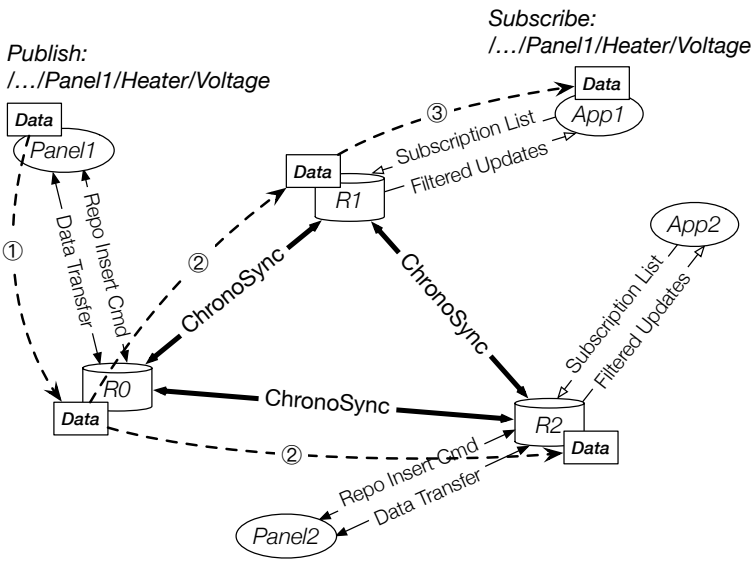

Figure 6.3: Data flow in a pub-sub group in NDN-PS

which implements the pub-sub semantics at the application layer. Figure 6.3 illustrates the data flow inside an NDN-PS pub-sub group with three repos.

All data packets in NDN-PS are authenticated using a hierarchical trust model expressed in the NDN names, which is aligned with real-world physical or logical structures such as campus buildings and enterprise management. The sensor data may be encrypted for access control [SDM14], in which case the data decryption keys (typically refreshed every few minutes) are also distributed to the consumers as data streams over NDN-PS.

Multiple pub-sub groups can be deployed independently on the campus network to support different applications and services either around the same location or across different buildings, which is illustrated in Figure 6.4. Different pub-sub groups can also be concatenated together, with the BMS applications subscribing to and processing the input data in one group and publishing the output data in another group. This enables a powerful design pattern of connecting multiple data aggregators and filters via pub-sub to achieve pipelined data analytics and event processing. For example, in a large enterprise campus the fine-grained raw data collected from the sensors in each building is usually consumed and processed by the data aggregation services deployed close to the panels and controllers, which may perform basic pre-processing for each data stream such as down-sampling or

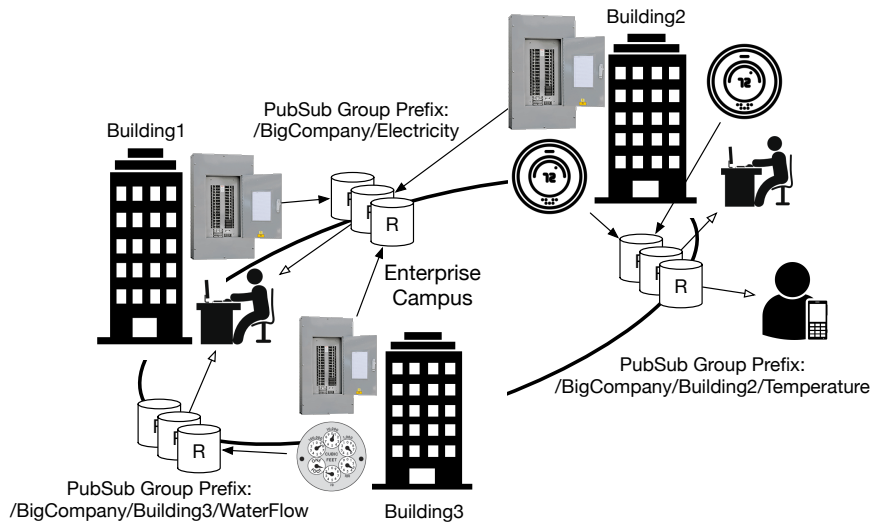

Figure 6.4: Deployment of three pub-sub groups on an enterprise campus network that serve different types of BMS data: electricity, temperature, and water flow

computing sliding-window average of the raw data points. Then the digested data is fed into a different pub-sub group and consumed by the next-stage processing jobs that further aggregate the data across multiple streams (e.g., computing the total power consumption of the whole building by adding up the power measurements from each room). A BMS data acquisition and analysis system can even go beyond simple aggregation and filtering by bridging multiple pub-sub groups to form arbitrary data flow graphs that can support complex data processing frameworks such as MapReduce [DG04].

CHAPTER 7

Conclusion

Distributed dataset synchronization provides a powerful abstraction for multi-party communication on top of NDN's network-layer primitives. It enables a group of distributed nodes to publish into and consume data from a shared dataset that holds the application data. At the center of the sync-based application architecture is the sync protocol that maintains a consistent state of the shared dataset among the participants in the sync group. The sync protocol typically provides means for representing the state of the shared dataset in order to support efficient detection and reconciliation of inconsistency among the sync nodes. When a new data packet is published in the shared dataset, the sync protocol disseminates the information about the new data to other nodes in the group, allowing others to update their local sync state to reflect the latest data published inside the group. Several sync protocols have been proposed for the NDN architecture to facilitate the development of distributed applications. While they all provide the basic dataset synchronization service, the existing sync protocols make different design decisions in several critical aspects such as data naming, dataset state representation, and state synchronization mechanism. In this dissertation we systematically analyze six existing sync protocols through a comparative study and identify the common design patterns in different approaches and the design trade-offs affecting the efficiency of sync communication under various conditions.

Informed by the past experience in developing the existing sync protocols, we design VectorSync, a novel sync protocol for the NDN architecture with built-in group membership management. Maintaining the group membership information at the sync layer facilitates data authentication and access control and improves the sync protocol efficiency by removing the departed nodes from the protocol state. It enables support for high-level services such as

dataset state snapshot and data total ordering that are often utilized by various distributed applications. We implement a prototype of the VectorSync module on top of the ndn-cxx library and evaluate the performance of VectorSync through simulation study. Our evaluation shows that VectorSync is able to provide consistent performance across different network environments with packet loss and simultaneous data publishing. Comparing with the widely adopted ChronoSync protocol, VectorSync achieves lower data synchronization delay and generates lower amount of traffic in the network when the nodes constantly publishing new data around the same time.

To demonstrate the utility of NDN sync in simplifying application design and improving data communication efficiency, this dissertation also describes three pilot IoT applications that utilize different NDN sync protocols to achieve important functions that are often difficult to implement in the TCP/IP-based IoT systems. Benefiting from NDN's data-centric communication mechanism, NDN sync efficiently synchronizes the knowledge of the data published in the IoT network without requiring pair-wise secured channels between the devices or deploying centralized server (either locally or in the cloud). This makes NDN sync suitable for supporting information dissemination in infrastructure-less edge network environments with energy-constrained devices and/or intermittent connectivity. NDN sync can also serve as the building block for designing large-scale distributed applications such as the NDN-PS publish-subscribe framework that utilizes distributed and synchronized repos to collect the sensor data and serve consumer requests in a building management system.

Our work on distributed dataset synchronization and its applications in NDN is still preliminary and there are a few interesting research directions that are worth exploring in future work.

First, an important research question is how to support group communication efficiently in the NDN network. All existing sync protocols, including VectorSync, assume the availability of network-layer multicast so that the nodes participating in a sync group can easily send multicast Interest packets to all the other nodes at once. However, there are many application scenarios where network-layer Interest multicast is either infeasible or prohibitively expensive, such as the ad hoc network environments. One alternative to network-layer multicast is

viral propagation (also called epidemic dissemination) [DGH87], where a node disseminates a message to a subset of its neighbors (e.g., nodes within the coverage of the wireless signal) and those neighbors further propagate the message until all nodes in the distributed system have received it. An interesting future work direction is to examine how to utilize the viral propagation mechanism to extend the applicability of NDN sync to the disruptive and infrastructure-less environments and how this new group communication model may affect the design of the sync protocol.

Second, we need to apply VectorSync to more NDN applications in order to (1) demonstrate the effectiveness and efficiency of the new protocol; (2) discover implementation issues in the prototype such as programming bugs and inconvenient API; (3) improve the performance of existing applications and explore new sync-based application design. We believe VectorSync will be able to support a variety of NDN applications and inspire new research activity in the area of NDN sync protocols and applications.

REFERENCES

[Ama17] Amazon Web Services. "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region." `https://aws.amazon.com/message/41926/`, March 2017.

[AZY15] Alexander Afanasyev, Zhenkai Zhu, Yingdi Yu, Lijing Wang, and Lixia Zhang. "The Story of ChronoShare, or How NDN Brought Distributed Secure File Sharing Back." In *Proceedings of the 12th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pp. 525–530, Oct 2015.

[Bre00] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)." In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, p. 7, New York, NY, USA, 2000.

[BSS17] Carsten Bormann, Zach Shelby, Peter Van der Stok, and Michael Koster. "CoRE Resource Directory." Internet-Draft draft-ietf-core-resource-directory-10, Internet Engineering Task Force, March 2017. Work in Progress.

[CK13a] S. Cheshire and M. Krochmal. "DNS-Based Service Discovery." RFC 6763 (Proposed Standard), February 2013.

[CK13b] S. Cheshire and M. Krochmal. "Multicast DNS." RFC 6762 (Proposed Standard), February 2013.

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[DGH87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic Algorithms for Replicated Database Maintenance." In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 1–12, 1987.

[DHJ07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 205–220, 2007.

[EGU11] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. "What's the Difference?: Efficient Set Reconciliation Without Prior Context." In *Proceedings of the ACM SIGCOMM 2011 Conference*, pp. 218–229, 2011.

[FBC15] Wenliang Fu, H. Ben Abraham, and P. Crowley. "Synchronizing Namespaces with Invertible Bloom Filters." In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 123–134, May 2015.

[FSD15] Chengyu Fan, Susmit Shannigrahi, Steve DiBenedetto, Catherine Olschanowsky, Christos Papadopoulos, and Harvey Newman. "Managing Scientific Data with Named Data Networking." In *Proceedings of the 5th International Workshop on Network-Aware Data Management (NDM)*, pp. 1:1–1:7. ACM, 2015.

[HCS17] Pedro de-las Heras-Quirós, Eva M. Castro, Wentao Shang, Yingdi Yu, Spyridon Mastorakis, Alexander Afanasyev, and Lixia Zhang. "The Design of RoundSync Protocol." Technical Report NDN-0048, NDN Project, April 2017.

[JST09] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. "Networking Named Content." In *Proceedings of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 1–12, 2009.

[KB12] Derek Kulinski and Jeff Burke. "NDN Video: Live and Prerecorded Streaming over NDN." Technical Report NDN-0007, NDN Project, September 2012.

[KNR11] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: a Distributed Messaging System for Log Processing." In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)*, Jun 2011.

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, **21**(7):558–565, July 1978.

[Lam98] Leslie Lamport. "The Part-time Parliament." *ACM Transactions on Computer Systems (TOCS)*, **16**(2):133–169, May 1998.

[Lam01] Leslie Lamport. "Paxos Made Simple." *ACM Sigact News*, **32**(4):18–25, 2001.

[LC12] Barbara Liskov and James Cowling. "Viewstamped Replication Revisited." `http://pmg.csail.mit.edu/papers/vr-revisited.pdf`, 2012.

[MAM16] Spyridon Mastorakis, Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. "ndnSIM 2: An updated NDN simulator for NS-3." Technical Report NDN-0028, Revision 2, NDN Project, November 2016.

[Mos14] Marc Mosko. "CCNx 1.0 Collection Synchronization.", Apr 2014.

[NDN17] NDN Project Team. "NDN-RTC Conferecing Library." `https://github.com/remap/ndnrtc`, 2017.

[OL88] Brian M. Oki and Barbara 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)*, pp. 8–17, 1988.

[PPR83] 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*, **SE-9**(3):240–247, May 1983.

[Pro12a] ProjectCCNx. "CCNx Synchronization Protocol." CCNx 0.8.2 documentation, 2012.

[Pro12b] ProjectCCNx. "Content-Centric Networking CCNx Reference Implementation." `https://github.com/ProjectCCNx/ccnx`, 2012.

[SBL16] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang. "Named Data Networking of Things (Invited Paper)." In *Proceedings of the 1st IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 117–128, Apr. 2016.

[SDM14] Wentao Shang, Qiuhan Ding, Alessandro Marianantoni, Jeff Burke, and Lixia Zhang. "Securing Building Management Systems using Named Data Networking." *IEEE Network*, **28**(3):50–56, May 2014.

[Sha17] Wentao Shang. "The VectorSync library." `https://github.com/wentaoshang/VectorSync`, 2017.

[SMK01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." In *Proceedings of the 2001 SIGCOMM Conference*, pp. 149–160, 2001.

[SMW04] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. "Measuring ISP Topologies with Rocketfuel." *IEEE/ACM Transactions on Networking*, **12**(1):2–16, Feb 2004.

[SWA17] Wentao Shang, Zhehao Wang, Alexander Afanasyev, Jeff Burke, and Lixia Zhang. "Breaking out of the Cloud: Local Trust Management and Rendezvous in Named Data Networking of Things." In *Proceedings of the 2nd International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 3–13, 2017.

[SYD16] Wentao Shang, Yingdi Yu, Ralph Droms, and Lixia Zhang. "Challenges in IoT Networking via TCP/IP Architecture." Technical Report NDN-0038, NDN Project, February 2016.

[SZA17] Wentao Shang, Minsheng Zhang, Alexander Afanasyev, Jeff Burke, Lan Wang, and Lixia Zhang. "Publish-Subscribe Communication in Building Management Systems over Named Data Networking." Manuscript submitted to ACM TCPS Special Issue on Internet of Things, 2017.

[VYW16] A K M Mahmudul Hoque Vince Lehman, Yingdi Yu, Lan Wang, Beichuan Zhang, and Lixia Zhang. "A Secure Link State Routing Protocol for NDN." Technical Report NDN-0037, NDN Project, January 2016.

[WAK12] Lucas Wang, Alexander Afanasyev, Romain Kuntz, Rama Vuyyuru, Ryuji Wakikawa, and Lixia Zhang. "Rapid Traffic Information Dissemination Using Named Data." In *Proceedings of the 1st ACM Workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*, NoM '12, pp. 7–12, 2012.

[WSJ15] Roy Want, Bill N Schilit, and Scott Jenson. "Enabling the Internet of Things." *Computer*, (1):28–35, 2015.

[YAC15] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, and Lixia Zhang. "Schematizing Trust in Named Data Networking." In *Proceedings of the 2nd ACM International Conference on Information-Centric Networking (ICN)*, pp. 177–186, 2015.

[YAM13] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. "A Case for Stateful Forwarding Plane." *Computer Communications*, **36**(7):779–791, April 2013.

[YAZ16] Yingdi Yu, Alexander Afanasyev, and Lixia Zhang. "Name-Based Access Control." Technical Report NDN-0034, Revision 2, NDN Project, Jan. 2016.

[ZA13] Zhenkai 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)*, pp. 1–10, Oct 2013.

[ZAB14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. "Named Data Networking." *ACM SIGCOMM Computer Communication Review (CCR)*, **44**(3):66–73, July 2014.

[ZBA12] Zhenkai Zhu, Chaoyi Bian, Alexander Afanasyev, Van Jacobson, and Lixia Zhang. "Chronos: Serverless Multi-User Chat Over NDN." Technical Report NDN-0008, NDN Project, October 2012.

[ZLW17] Minsheng Zhang, Vince Lehman, and Lan Wang. "Scalable Name-based Data Synchronization for Named Data Networking." In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, May 2017.

[ZWY11] Zhenkai Zhu, Sen Wang, Xu Yang, Van Jacobson, and Lixia Zhang. "ACT: Audio Conference Tool over Named Data Networking." In *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking (ICN)*, pp. 68–73, 2011.