UNIVERSITY OF CALIFORNIA

Los Angeles

Enabling Decentralized Applications: A Transport Perspective

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Varun Patil

2025

ABSTRACT OF THE DISSERTATION

Enabling Decentralized Applications: A Transport Perspective

by

Varun Patil
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2025
Professor Lixia Zhang, Chair

The growing centralization of the Internet is raising concerns over user privacy, third-party censorship, and the control of the Internet by a few dominant players. This centralized control persists even in recent applications explicitly designed for decentralization. Fundamental limitations within the current Internet architecture, particularly the node-centric, point-to-point connectivity model of TCP/IP, and the absence of built-in security protection, compounded by Network Address Translation (NAT) and the lack of platform-independent end user identity, significantly hinder the ability for users to communicate directly and securely, which is the necessary condition to build truly decentralized applications.

Named Data Networking (NDN) shows promise as a foundation for addressing these challenges through its data-centric communication model, which utilizes semantic names to identify users, applications, and data, and secures data directly. In this dissertation, built on the foundation of NDN, we present the design and implementation of several key components in the NDN protocol stack to enable resilient decentralized applications. First, we introduce State Vector Sync (SVS), a multi-party transport protocol to support NDN applications through simple, resilient, and efficient dataset synchronization. Second, we describe the

NDN Distance Vector Routing Protocol (ndn-dv), which can establish reachability in name-based NDN networks, scalably and efficiently. ndn-dv is both a showcase of SVS usage and a unique solution in its own right, demonstrating how NDN's stateful forwarding plane mitigates traditional distance-vector routing issues. Third, to enhance developer accessibility, we implemented NDNd, a consolidated Golang NDN stack featuring high-level APIs.

We integrate the above primitives to build Ownly, a decentralized collaborative workspace application. By leveraging SVS for communication and NDN's data-centric, name-based security, Ownly successfully eliminates the dependency on infrastructure-based control points, which could lead to centralized control.

This research provides an analysis of Internet centralization from a networking perspective, contributing practical NDN-based components and a real-world application. We describe in detail the insights gained from our development experiences regarding the reasons for Internet centralization, namely the lack of direct user-to-user communication and the dependency on platform-specific user identifiers. With these insights, we hope to demonstrate a viable path towards long-term decentralization and resilience in the digital ecosystem.

The dissertation of Varun Patil is approved.

Beichuan Zhang

Harry Xu

Songwu Lu

Lixia Zhang, Committee Chair

University of California, Los Angeles

2025

Here's to the ones who dream

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I want to express my heartfelt gratitude to my doctoral advisor, Prof. Lixia Zhang. Throughout this journey, her unwavering support, intellectual rigor, and resilience have been a constant source of inspiration. Being her student taught me to think independently and fearlessly, to pursue unconventional ideas, question established assumptions, and develop confidence in my own voice; these lessons extend far beyond the scope of this thesis. I am deeply thankful for her mentorship, both academically and personally.

I am also grateful to my current and former Internet Research Laboratory colleagues Philipp Moll, Xinyu Ma, Tianyuan Yu, Mark Theeranantachai, and Adam Thieme; and our collaborators Davide Pesavento, Junxiao Shi, Prof. Alexander Afanasyev, Prof. Beichuan Zhang, Prof. Lan Wang, and many other members of the NDN community. Our stimulating discussions, encouragement, and shared persistence made this journey both challenging and immensely rewarding.

My understanding and insights have been profoundly shaped by my time in industry; a special thanks to Scott Gray, Randy King, Andrew Bartels, Shawn Green, and everyone else at Operant Networks for supporting me through this journey. I would also like to thank Kathleen Nichols for many thought-provoking discussions that significantly contributed to my understanding.

And to my friends and family, whose constant belief in me never wavered – thank you for being my foundation, for your patience, your love, and your gentle reminders of what really matters.

VITA

2016–2020	B.Tech. Mechanical Engineering, Indian Institute of Technology Bombay
2019	Software Engineering Intern, Mercari Inc., Tokyo, Japan
2020–2021	M.S. Computer Science, UCLA
2022	Software Engineering Intern, LabN Consulting L.L.C., US
2024	Software Engineering Intern, Systems and Infrastructure, Meta Platforms, Inc., Menlo Park, California, US
2022–2025	Teaching Assistant, Computer Science Department, UCLA
2023–2025	Software Developer Intern, Operant Networks, US
2021–2025	Graduate Student Researcher, Internet Research Lab, Computer Science Department, UCLA

PUBLICATIONS

Philipp Moll, Varun Patil, Nishant Sabharwal, and Lixia Zhang. "A Brief Introduction to State Vector Sync." *Named Data Networking, Tech. Rep. NDN-0070*, 2021.

Philipp Moll, Varun Patil, Lixia Zhang and Davide Pesavento. "Resilient Brokerless Publish-Subscribe over NDN." In *Proceedings of IEEE Military Communications Conference (MILCOM) 2021*, 2021.

Varun Patil, Philipp Moll, and Lixia Zhang. "Poster: Supporting Pub/Sub over NDN Sync." In *Proceedings of the 8th ACM Conference on Information-Centric Networking (ICN) 2021*, 2021.

Siqi Liu, Varun Patil, Tianyuan Yu, Alexander Afanasyev, Frank Alex Feltus, Susmit Shannigrahi, and Lixia Zhang. "Designing Hydra with Centralized versus Decentralized Control: A Comparative Study." In *Proceedings of the Interdisciplinary Workshop on (de) Centralization in the Internet (IWCI) 2021*, 2021.

Philipp Moll, Varun Patil, Lan Wang, and Lixia Zhang. "SoK: The Evolution of Distributed Dataset Synchronization Solutions in NDN." In *Proceedings of the 9th ACM Conference on Information-Centric Networking (ICN) 2022*, 2022.

Varun Patil, Sichen Song, Guorui Xiao, and Lixia Zhang. "Poster: Scaling State Vector Sync." In *Proceedings of the 9th ACM Conference on Information-Centric Networking (ICN) 2022*, 2022.

Varun Patil, Hemil Desai, and Lixia Zhang. "Kua: A Distributed Object Store over Named Data Networking." In *Proceedings of the 9th ACM Conference on Information-Centric Networking (ICN) 2022*, 2022.

Tianyuan Yu, Hongcheng Xie, Siqi Liu, Xinyu Ma, Varun Patil, Xiaohua Jia, and Lixia Zhang. "CLedger: A Secure Distributed Certificate Ledger via Named Data." In *Proceedings of IEEE International Conference on Communications (ICC) 2023*, 2023.

Varun Patil, Seiji Otsu, and Lixia Zhang. "Poster: Timers in State Vector Sync." In *Proceedings of the 10th ACM Conference on Information-Centric Networking (ICN) 2023*,

2023.

Varun Patil, Tianyuan Yu, Xinyu Ma, and Lixia Zhang. "Poster: Decentralized Photo Sharing via Named Data Networking." In *Proceedings of the 10th ACM Conference on Information-Centric Networking (ICN) 2023*, 2023.

Philipp Moll, Varun Patil, Alex Afanasyev, Junxiao Shi, and Lixia Zhang. "Lessons Learned from Fixing the Dead Nonce List in NFD." In *Named Data Networking, Tech. Rep. NDN-0077*, 2023.

Justin Presley, Xi Wang, Xusheng Ai, Tianyuan Yu, Tym Brandel, Proyash Podder, Varun Patil, Alex Afanasyev, F. Alex Feltus, Lixia Zhang, and Susmit Shannigrahi. "Hydra: A Scalable Decentralized P2P Storage Federation for Large Scientific Datasets." In *Proceedings of the 2024 International Conference on Computing, Networking and Communications (ICNC)*, 2024.

Tianyuan Yu, Xinyu Ma, Varun Patil, Yekta Kocaogullar, Yulong Zhang, Jeff Burke, Dirk Kutscher, and Lixia Zhang. "Secure Web Objects: Building Blocks for Metaverse Interoperability and Decentralization." In *Proceedings of the 2024 IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom)*, 2024.

Tianyuan Yu, Xinyu Ma, Varun Patil, Yekta Kocaoğullar and Lixia Zhang. "Exploring the Design of Collaborative Applications via the Lens of NDN Workspace." In *Proceedings of 2nd Annual IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom) 2024*, 2024.

Tianyuan Yu, Jacob Zhi, Xinyu Ma, Yekta Kocaogullar, Varun Patil, Ryuji Wakikawa, and

Lixia Zhang. "Repo: Application Agnostic and Oblivious In-Network Data Store." In *Proceedings of the 2024 IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom)*, 2024.

Varun Patil, Sirapop Theeranantachai, Beichuan Zhang, and Lixia Zhang. "Poster: Distance Vector Routing for Named Data Networking." In *Proceedings of the 20th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT) 2024*, 2024.

CHAPTER 1

Introduction

The centralization of the Internet has been attracting increasing attention during recent years. Concerns over user privacy, third-party censorship, and the control of a few dominant players have raised questions about the resilience and fairness of the digital ecosystem. Some recent designs such as the AT protocol and IPFS claim to enable decentralized applications; however, measurements have shown that these systems also tend to quickly move towards centralization of control power [WTP24, BSA24]. This trend indicates the need to further understand the root causes of Internet centralization. We identify design principles that promote long-term decentralization and resilience by analyzing the technical, economic, and social factors that drive this shift.

In this dissertation, we examine the issue of centralization from a networking and transport point of view. We provide insights into how the point-to-point connectivity and security model of TCP/IP are among the fundamental driving forces behind today's Internet centralization. We identify secure peer-to-peer communication as a cornerstone requirement for decentralized applications and discuss how the current Internet architecture cannot fulfill this requirement due to IP's host-centric connectivity, NAT, and the lack of platform-independent user identity.

Built on the foundation of NDN architecture [ZAB14], we contributed several new components to the NDN protocol stack, and implemented a functional decentralized application using these primitives.

1. We designed State Vector Sync (SVS) [MPS21], a secure, efficient, and performant

Distributed Dataset Synchronization (Sync) protocol to provide an NDN transport service. SVS allows instances of a distributed NDN application to communicate resiliently without reliance on centralized server infrastructure, serving as a multi-party transport layer in the NDN protocol stack.

2. NDN forwards packets using names rather than addresses. This requires new routing paradigms to establish reachability in NDN networks. We designed and implemented the NDN Distance Vector Routing Protocol (ndn-dv) to satisfy this requirement. The ndn-dv design uses SVS to efficiently synchronize prefix reachability state within the routing domain. We discuss in detail how the stateful forwarding plane of NDN can automatically work around the traditional shortcomings of DV routing.

3. Using NDN primitives and building on lessons learned from the *NDN Workspace* [Ma24] application, we designed and implemented *Ownly*, a decentralized collaborative text editor. By utilizing SVS as the transport together with name-based security, Ownly eliminates dependencies on centralized control points.

4. Application developers are typically not familiar with low-level network layer primitives. We implemented NDNd [Nam25d], a new consolidated Golang implementation of the NDN stack with a focus on developer usability and high-level APIs.

This work focuses on State Vector Sync as a crucial component for enabling the development of data-centric decentralized applications. SVS serves as a transport service in NDN, providing lightweight, secure, and efficient namespace synchronization to a group of communicating entities. The design of ndn-dv both supports and serves as a use case of SVS. The routing daemon is an NDN application that uses SVS in a specific setting, where all routers participate in a communication group to exchange updates using Sync. We further integrate these components into Ownly, a fully decentralized, collaborative workspace that leverages SVS for real-time synchronization. Through Ownly, we demonstrate how SVS enables higher-layer application logic such as persistent storage and offline support. Ownly

marks a concrete step towards realizing decentralized applications, and further improves the understanding of the role NDN's Sync transport plays in building new applications.

The rest of the dissertation is organized as follows. Chapter 2 provides the requisite background on NDN and relevant discussion on transport. We provide an in-depth discussion on the SVS design in Chapter 3 together with performance evaluations. Chapter 4 presents the design of ndn-dv along with some preliminary evaluation results. The system design and implementation overview of Ownly is described in Chapter 5, along with a discussion on the role of the transport layer in the design of such a decentralized application. Chapter 6 briefly describes other contributions the author has made to the overall NDN ecosystem, including the development of NDNd. The dissertation concludes with Chapter 7.

CHAPTER 2

Background

In this chapter, we review the evolution and current understanding of the transport layer in modern applications. We also provide a brief overview of the Named Data Networking (NDN) architecture, and an introduction to distributed dataset synchronization (Sync) in NDN.

2.1 The Transport Layer

In the initial design of the Internet, the Transmission Control Protocol (TCP) was introduced as a generic packet-switching protocol that provided a broad range of functions, including packet forwarding, multiplexing, in-order delivery, and reliability [CK74, LCC09]. However, it soon became evident that, in order to support a diverse set of applications, a separation of concerns was necessary – a concept proposed by Postel [Pos77]. This led to the split of TCP into two distinct protocols: the Internet Protocol (IP) [Pos81a], responsible for best-effort packet delivery, and a new TCP [Pos81b], which focused exclusively on end-to-end transport services. TCP was designed to provide reliable delivery of a stream of bytes between applications running on hosts communicating over an IP network. TCP would eventually also be tasked with congestion control, ensuring that the network could handle the data flow without overwhelming it.

Following the separation of TCP/IP, IP became the *narrow waist*[1] of the Internet, serving

[1] `https://www.ietf.org/proceedings/51/slides/plenary-1/`

as the *span layer* to connect together various operational networks across the globe. In a layered protocol stack, the functionality of a lower layer determines what can be built over it. IP's nature of point-to-point datagram delivery, for example, implicitly constrained all transport protocols built over it to follow a similar point-to-point communication model. We discuss the effects of IP's design further in §2.1.2.

Internet applications have evolved in fundamental ways over the last four decades. Yet, most of them continue to utilize the "virtual circuit" communication service offered by TCP, where application instances exchange data over a reliable point-to-point connection. Newer transport layer protocols, such as QUIC [IT21], also provide similar functionality that largely mirrors that of TCP.

2.1.1 A Changing Application Landscape

Computer networking started with the initial goal of sharing resources. Early applications such as remote login and simple host-to-host file transfer were a natural fit for the virtual circuit communication model, where one end was the user and the other was a remote server. Email, the first killer application on the Internet, did not fit well into the virtual circuit model; email is inherently *multiparty*, requiring always-online servers in the middle to handle the asynchronous nature.

The applications the Internet supports today are much more diverse. Internet traffic today is dominated by video [Cis23], with streaming services such as Netflix and YouTube accounting for a significant portion of the total traffic. These applications are fundamentally different from the earlier ones, since they often have a large number of receivers all requesting the same content. The same can be said for other popular applications, such as social media, where popular influencers can have thousands of followers, and the same content is delivered to all of them. This application traffic pattern is in stark contrast to the virtual circuit service offered by TCP, which assumes that each application instance communicates with a single peer.

The conflict between these modern applications and the traditional virtual circuit communication service is evident in the way they are implemented. Video streaming applications use a client-server model; the scalability challenges introduced by a large number of clients has led to the proliferation of content delivery networks (CDNs) which cache content closer to the end users. With CDNs, the end user continues to use a point-to-point connection, but the other end is now chosen from a pool of nearby servers [ZSR20].

A second example of new applications is the Internet of Things (IoT), where a large number of devices communicate with each other. These applications are often characterized by variety of communication patterns, where each device may need to communicate with any other nearby devices. Implementations of IoT often rely on connecting all devices to a centralized cloud provider, which acts as a broker for all communication. This model is not only inefficient, but also introduces a single point of failure and privacy concerns, since all data must be sent to the cloud provider [ZMK15, ARJ22].

These examples highlight the gap between TCP's virtual circuit model and the requirements of modern applications. In today's Internet, these conflicts are largely mitigated by adding additional layers of indirection, such as CDNs or cloud providers, which do not require changes to TCP/IP's network delivery model. These solutions come with significant drawbacks, which we discuss later. This indicates the need for a new approach to the transport layer, one that is better suited to the requirements of modern applications.

2.1.2 Multicast in IP Networks

As described in the previous section, several classes of modern applications are characterized by multi-party communications. One developed solution to reducing the inefficiency of IP's point-to-point deliveries is IP multicast routing, which can deliver IP datagrams from one node to multiple receivers. This is a natural fit for applications such as video streaming, where the same content is delivered to multiple users.

Protocols such as Internet Group Management Protocol (IGMP) [Fen97] and Multicast Listener Discovery (MLD) [CV04] have been designed and implemented for multicast group membership management. Several multicast routing protocols have also been designed, including Distance Vector Multicast Routing Protocol (DVMRP) [WPD88], Protocol-Independent Multicast (PIM) [FHH16], Multicast OSPF (MOSPF) [Moy94] and Multiprotocol BGP (MP-BGP) [CBR07]. However, despite these efforts, multicast has not been widely adopted in operational IP networks.

There is a fundamental reason why IP multicast is not rolled out – it represents a disguised architectural change. IP is designed for point-to-point datagram delivery, and each unicast IP address represents a network attachment point. An IP multicast address, on the other hand, represents a group of receivers that desire *the same content*, a fundamental change to the IP address semantics; a change disguised by making both have the same byte count.

The second architectural change that we notice is that IP multicast forwarding is *stateful* in the data plane. To forward a packet destined to a multicast address, IP routers need to set up and maintain the forwarding state that leads to group members down the paths, a fundamental departure from IP's stateless forwarding. The members of the group, as well as their topological locations, may change dynamically at any time, which requires the IP multicast routing solutions to manage the membership of each group. Therefore, the forwarding state at IP routers must be updated throughout the network whenever members join or leave the group.

Finally, we note that receivers determine which groups to join in an multicast network, depending on what data they desire. In other words, IP multicast follows a receiver-initiated (pull) model, which is in sharp contrast to the sender-initiated (push) model of IP unicast.

In short, unicast and multicast IP datagram deliveries represent completely distinct concepts that reflect a fundamental change in the network architecture. The difficulty of deploying *architecturally different* IP multicast in existing IP networks is a key reason for the lack of wide adoption.

Nevertheless, extensive investigations have been conducted in the areas of IP multicast support and new transport protocols for operating over multicast packet delivery. These efforts have provided many valuable insights. We identify some common themes across various multicast transport protocol designs, including Scalable Reliable Multicast (SRM) [FJL97], Pragmatic General Multicast (PGM) [SCG01] and NACK-Oriented Reliable Multicast (NORM) [MBH09]. These insights served as part of the inputs into the design of Named Data Networking described further in §2.2.

1. NACK-based recovery is used for reliable delivery. In the event of a packet loss, the receiver sends a NACK to the sender, which then retransmits the lost packet. While somewhat similar to TCP's approach to reliable delivery, this approach suffers from either the sender being overwhelmed by NACKs, or receivers being overwhelmed by unnecessary retransmissions. For example, SRM uses NACK suppression to limit the number of NACKs sent by receivers, but data retransmissions are still multicast to all receivers.

2. SRM also uses separate session messages for collaborative loss recovery in case an original data transmission is lost[2].

In addition to being an architectural change, the issues in IP multicast deployment have been further exacerbated by the lack of usable security support for multi-party communication. We elaborate on this missing piece in the next section.

2.1.3 The Role of Security

The Internet protocol stack was originally designed with no security protection. The introduction of security protocols such as TLS [RD08] with the associated WebPKI, and

[2]Described in Chapter 3, the timer-driven Sync Interests and suppression mechanisms of State Vector Sync are directly inspired from work on SRM's loss recovery [SEF97]

IPsec [SK05] was a response to the requirements of security that arose from the commercialization of the web. We make two important observations from this evolution.

1. Given that the security consideration was an afterthought, these security enhancements were intended to be *patched into* the existing protocol stack. Therefore, they must be deployable without changing either the semantics of the underlying protocols (e.g. TCP), nor the applications built on top of the existing protocols. Since the applications were designed to expect point-to-point virtual circuit delivery, these security protocols offer the same delivery semantics. That is, they do not understand application-level semantics and do not provide any security beyond an authenticated and encrypted communication channel. For instance, TLS offers a secured channel between two endpoints, but does not provide any guarantees about the authenticity of the data being exchanged over this channel [ZYZ18].

2. These protocols secure point-to-point communication and do not support multicast. This lack of usable multicast security protocols has been another major barrier to the adoption and further development of multicast in IP networks [Par20].

We note that TLS, in particular, was largely deployed within an already popular client-server application model on the web. While TLS theoretically supports client authentication, deployment is scarce. This is because applications wish to authenticate services by domain names instead of IP addresses, and clients use TLS to authenticate servers' ownership of a given domain name using a certificate issued by a WebPKI CA. The clients themselves, on the other hand, possess neither DNS names nor certificates. This design has led to several unintended consequences.

1. In typical usage, TLS establishes an asymmetrically authenticated channel, leaving user authentication to the application. This has led to a proliferation of user authentication mechanisms, which are often not interoperable. As a result, users' identities are tied

to a single service provider, leading to a lack of portability of user data. This issue provides fuel to the "walled garden" problem, where users are locked into a single service provider, ultimately leading to the centralization of user data [ZMM20].

2. The WebPKI is a system where a relatively small number of certificate authorities (CAs) are trusted to issue certificates for all domain names. Application services are authenticated solely through the WebPKI, leading to a major consolidation of power in a small number of entities. Each CA in the WebPKI possesses the maximum privilege in the sense that it can issue certificates to any application services. Thus, a single point of failure could lead to unbounded damage; compromising a single CA can lead to the compromise of all domain names [ES00, SHC19].

3. Relying on the WebPKI inherently places a third party (the CA) at the center of every secure communication. The authenticity and integrity of connections between clients and servers rely on the judgment and operational security of external entities that are not part of the communicating endpoints themselves. In other words, neither the client nor the server has direct control over the security of their communication.

4. As an application layer overlay, CDNs are also required to use the same point-to-point security protocols, and must terminate users' TLS connections to enable caching at the edge. To allow end-users to authenticate the TLS channel, CDN servers need to be able to prove ownership of the domain name of the service. As a result, users and service providers must now implicitly trust a third party (the CDN) to handle any sensitive user data safely [LJD14].

Our observations suggest that the notion of security as an afterthought has led to a number of issues in the implementation of modern applications. Additionally, protocols such as TLS do not address critical requirements, including user authentication and access control. Today's applications implement ad hoc solutions such as Access Control Lists (ACLs), which are complex to implement and maintain, and provide poor security guarantees and limited

auditability [KS02, Nag01]. In the next section, we introduce the Named Data Networking architecture, which was designed to address many of these issues.

2.2 Named Data Networking

Named Data Networking (NDN) [ZAB14, ABR18] is a proposed future Internet architecture that shifts the network communication paradigm from the host-centric model of TCP/IP to a data-centric model. In NDN, packets are identified by hierarchically structured identifiers called "names" rather than a source and destination address. These names identify each piece of data uniquely, and describe application-layer semantics of the contents of the data. NDN uses data names across the stack, for example, in routing, forwarding, security, as well as at the application layer.

NDN uses named secured data as the basic building block, providing a pull-based datagram delivery model for these pieces of named data. We note that the NDN design is built over various insights gained from protocol designs of the past. Some of these are as follows.

1. NDN encourages the use of Application Level Framing (ALF) [CT90], where each Application Data Unit (ADU) represents a single piece of named and individually secured data. ALF allows applications to define data units as viewed by the applications; this data identification can then be used for securing the data (once the ADU is semantically named) and by lower layers for efficient data delivery.

2. NDN follows the steps of IP multicast design as described in §2.1.2. More specifically, NDN adopts a receiver-driven approach to data dissemination, which is further elaborated in the description below. This pull-based approach takes a data-centric view of networking and leads to stateful forwarding, both are fundamental to the architecture, eliminating the conflict observed in IP multicast support.

3. The idea of securing data can be traced back to the early versions of DNSSEC [Hof23].

DNS itself is a data-centric protocol which heavily relies on distributed caching; as a result it cannot rely on secured pipes and one must directly secure the records instead. At a high level, DNSSEC promotes the use of individually secured records to ensure integrity and authenticity. Instead of relying on secure pipes, DNSSEC directly attaches cryptographic signatures to DNS resource records, allowing resolvers to verify that the data has not been tampered with, and originates from a trusted source. Due to its independence from secure channels, this model works even with the distributed caching model of DNS. NDN adopts a similar mechanism, inheriting these benefits for all types of data.

At the network layer, NDN defines two types of packets. *Interest* packets are used by data consumers to express their interest in a specific piece of data, identified by a name. *Data* packets carry the application data contents. When a consumer desires a piece of data, it sends an Interest packet to the network. From this point, it is the network's responsibility to locate the data and deliver it to the consumer. The network uses the name in the Interest packet to forward it towards a producer, which is a node that claims to have the requested data. The producer responds with a Data packet, returned to the consumer. This architecture fundamentally differs from TCP/IP, where the network only forwards packets to a given location identifier, i.e. an IP address, and the application is responsible for locating the desired data in the network.

The NDN data plane uses *stateful* [YAM13] forwarding, where the data plane maintains the state of pending Interests in the network. Each node in an NDN network runs a NDN packet forwarder. When a forwarder receives an Interest, it performs a longest prefix match on its local forwarding table (FIB) to find the ideal next hop to reach the data producer. This FIB is built by an NDN routing protocol, which establishes the reachability to name prefixes announced by data producers in the network. When an NDN forwarder forwards a packet to a neighbor, it creates an entry in a Pending Interest Table (PIT), indicating the interface on which the Interest was received. These PIT entries generate a breadcrumb

trail in the network for each Interest packet. When the Interest reaches a producer, the producer responds with a Data packet, which is propagated back to the consumer following the breadcrumb trail in reverse, clearing the PIT state along the way.

The NDN forwarding design has a few commonalities and some important differences when compared to IP forwarding. Both NDN and IP use a FIB to forward packets, but NDN uses the stateful PIT to forward Data packets. In both cases, the FIB uses a longest prefix match; in IP this match is done for an address identifying a host in the network while in NDN the match is performed for a data name prefix. NDN forwarders also use additional state tables such as the Content Store for caching and the Dead Nonce List to break packet loops. As a notable commonality, both IP and NDN also utilize routing protocols to populate the FIB and establish prefix reachability in the network.

Stateful forwarding in NDN naturally enables multicast delivery of Data packets. When a forwarder receives an Interest packet, it checks if it has already forwarded an Interest with the same name. If so, it does not forward the Interest again but adds the incoming interface to the PIT entry. This allows multiple Interests for the same data to be aggregated in the network forming a multicast tree, and a single Data packet can be sent back to all consumers that expressed interest in it. Further, since Data is immutable, forwarders may cache Data packets in a *Content Store*, allowing multicast delivery of Data even if two Interests for the same Data do not arrive simultaneously. As a result, NDN can efficiently support data dissemination in applications that require one-to-many communication.

NDN's network layer must support multicast routing, which is used for Sync Interest multicast, a key requirement in enabling Sync (§2.2.2). All NDN Sync protocol designs, except the State Vector Sync (SVS) described in this dissertation, multicast Sync Interests to *all* participants in an application who may reply with either newly updated dataset names or newly produced Data. When a producer replies to the Interest, the Data is propagated back to the consumer in a manner similar to the reply to a unicast Interest. However, one Interest can only bring back exactly one piece of data. In case of multiple producers reply to

the same multicast Interest, only one reply is propagated back to the Sync Interest sender, the remaining replies are cached within the network and eventually discarded; different participants may also receive different data replies.

2.2.1 Data-Centric Security

Today's TCP/IP model views a network as made of interconnected nodes which are identified by IP addresses; communication security is an afterthought that is patched onto each end-to-end connection, e.g. running TLS on top of TCP. NDN, on the other hand, views a network as made of semantically named entities with various trust relations among each other; these named entities can be users, devices, services/app instances, or anything that produces and/or consumes named data.

NDN's approach is based on the principle of *data-centric security* [ZYZ18]. Each piece of data is cryptographically signed by a key owned by the producer. Every piece of data is associated with a *KeyLocator*, an NDN name that points to a certificate for the key used to sign this data. Consumers of data directly verify the signature of the data. This allows any consumer to verify the authenticity of the data, regardless of where it is received from.

This data-centric security model is distinctly different from TLS's channel-based security model. In TLS, security is established between two endpoints, but TLS does not subsequently control what data is sent over the channel. Further, TLS does not guarantee the authenticity of data received from a third party. In contrast, NDN security is built into the data itself, independent of the channel. This also means the data is always secure in transit and at rest, as soon as it is produced.

To verify data from others, all nodes in an NDN network undergo a process of security bootstrapping [YMX23a]. During bootstrapping, each entity is provisioned with one or more trust anchors, a valid identity certificate and a set of security policies that define trust relationships with other entities. Notably, these security policies may be schema-

tized [YAC15] by utilizing the semantic nature of NDN data names. This provides a flexible and scalable mechanism for establishing trust in the network without the need for complicated ACLs [Nic21, YMX23b]. Once an entity is bootstrapped, it can produce signed data and verify data produced by others. The security policies provide fine-grained control on which data can be produced by which entity.

For confidentiality, producers may encrypt their data. NDN automates key distribution and management using Name-based Access Control (NAC) [ZYR18]. Once a piece of data is encrypted, NAC securely distributes the decryption keys only to authorized consumers. The encrypted data can also be directly cached and stored inside the network and served to multiple consumers. As a result, NDN does not require separate caching mechanisms like CDNs, and the data can be end-to-end secured, even when cached in the network.

2.2.2 Transport in NDN

In general, transport services bridge the gap between services that the network layer provides and the services that applications require. In a Named Data Network, the network layer fetches semantically named data chunks. As a result, the transport layer functions in NDN also differ from those in TCP/IP.

1. Since IP only supports host-to-host communication, a key function of TCP and UDP is to enable process-to-process communication using port numbers. NDN uses Data names for demultiplexing across all protocol layers, and thus it does not need port numbers or other additional information at the transport layer.

2. An NDN transport service does not need to perform congestion control, since this function is moved to the network layer where it belongs [SYZ16, SZ22, YAM13].

3. As a data-fetching protocol, NDN allows individual application instances to determine which data to fetch and when. As a result, unlike TCP, an NDN transport service does not push data to the application.

At first sight, it may seem that NDN does not need a transport at all and, in fact, the very first NDN proposals [ZEB10] did not have a transport layer. However, from experience gained through developing applications over NDN, we find new requirements in the data-centric architecture. To fetch data from the network, applications first need to know what data is available. In a multiparty communication group, the individual participants need to learn the names of all newly produced data by others as soon as possible, so they can retrieve the data promptly if needed by the application.

This requirement of *distributed dataset namespace synchronization*, or *Sync*, is a basic service needed by NDN applications. Sync provides reliable synchronization of the names in a shared dataset in a group of application instances, called a *Sync Group*. Sync has been used as a transport layer service for a number of existing NDN applications, such as a chatroom app [AZY15], a network routing protocol [WLH18], a network management framework [Nic19], a data collection system [DAT22] and a pub/sub implementation [MPZ21].

As a transport layer service running over NDN's Interest-Data exchanges, Sync offers unique advantages in supporting heterogeneous receivers and diverse data reliability requirements. While all applications desire some form of reliability, the definitions of reliability vary widely. The role of NDN Sync is to synchronize the dataset *namespace* (i.e. the collection of data names that have been produced in the group). Sync, however, does not push data to participants of the group. Instead, each individual member decides whether and when to fetch all or some of the data items, based on the application's needs as well as factors such as timeliness and local resource constraints.

2.2.3 Architectural Layering in NDN

Layering in the NDN architecture is nuanced due to the use of a single header across layers, and deserves further elaboration.

Each NDN packet carries a single name, which serves multiple purposes across different

layers. At the network layer, the name is employed to forward Interest and Data packets via longest-prefix matching using the FIB and the PIT. Importantly, the network layer treats the name as an opaque identifier, relying solely on hierarchical bitwise comparisons to perform forwarding functions. The transport service also uses the same name; Sync may interpret certain semantics embedded in the name, such as sequence numbers, to facilitate efficient namespace synchronization.

However, ultimately, the naming of data is under the purview of the application layer, which both defines and interprets the name. Applications sign and verify named data, with the name forming the basis for security, defining who may use this data and how it may be used. When an application produces a new piece of named and signed data, Sync notifies everyone in the communication group that this new data is now available to fetch. Consumers then use the network layer to fetch this data from the producer (if desired), with the network layer handling forwarding and tasks such as segmentation and fragmentation to fit packets within MTU constraints. Each layer in this process uses the same name that was defined by the application.

This discussion aims to illustrate that NDN maintains a layered architecture where each layer provides distinct functionality, but does not have separate protocol headers per layer. Instead, a single header, which carries a unified namespace with application-level semantics, is utilized across layers. Each layer makes use of the information provided by the header in different ways. Furthermore, NDN does not have the concept of a "security layer"; instead, each individual piece of named data is cryptographically secured and may be verified independently at any point in the stack.

CHAPTER 3

State Vector Sync (SVS)

In this chapter, we describe the design of the State Vector Sync (SVS) protocol. First, we examine various design aspects of an NDN Sync protocol, followed by a systematic analysis of all the previous Sync designs. We then detail the design choices of SVS and elaborate on SVS's approaches to packet loss recovery, protocol efficiency, and scalability.

3.1 The Design of NDN Sync Protocols

NDN consumers fetch data using names and control the level of reliability depending on the application. To enable fetching data by name, a consumer must first know what data is available – the job of Sync is to propagate this information to consumers efficiently. We first list some design goals for Sync protocols based on general observations of application needs [SYW17, MPW22].

1. **Reliable *namespace* Sync:** While applications decide which data to fetch, they must first reliably know which data is available to enable an informed decision. In the absence of permanent network partitions, a Sync participant should eventually learn all the names in the shared name dataset. Network environments may also vary widely, ranging from stable wired infrastructure networks with minimal losses to ad hoc wireless networks with intermittent connectivity and frequent network partitions. To provide a generally usable transport service, Sync protocols should work reliably regardless of network conditions.

2. **Low synchronization latency:** This property is required to meet the low-latency requirements of real-time applications such as online games or conference calls. Sync must promptly inform all participants in the same application about changes to the dataset, i.e. whenever any participant produces new data.

3. **Low overhead:** Like any other protocol, Sync itself should have a low network and processing overhead. This property is required to effectively use the Sync protocol in IoT applications and embedded environments.

Over the years, three basic design components of Sync protocol design have been identified. First, we must define a representation of the shared dataset's namespace. Second, the protocol must identify an efficient way of encoding the dataset or changes to the dataset so that these can be transmitted over the network. As we discuss below, the encoding of the dataset namespace and the encoding used for synchronizing the dataset changes may or may not be the same. Finally, a mechanism is needed to promptly propagate changes to the shared dataset to all other participants in the group. We describe each component in further detail below.

1. **Namespace Representation:** To synchronize the namespace dataset, we must first start by considering different representations of the namespace itself. Two different approaches have been observed in this regard. The first approach is to use application data names directly. Thus, the Sync namespace is simply a set of all the names of data items that the application has produced. The second approach names the data items generated by each producer P sequentially, and can thus be represented as a 2-tuple of [producer name, seq#]. Consequently, the namespace of an application can now be represented as a list of such tuples of [producer name, seq#], one tuple for each participant. The sequence number increases monotonically for each producer whenever that producer produces new data. Thus, a current value of the sequence number N

for P indicates the existence of data corresponding to each number in the range $[1, N]$ for P.

2. **Dataset State Encoding:** State encoding is required to convert the namespace representation into a compact form for transmission over the network. As we discuss in §3.2, each of the previous Sync protocols made its unique state encoding design, based on the tradeoffs among multiple factors: application data name representation (whether taking application data names as is, or naming data sequentially), the compactness of dataset state transmission, and the resiliency in Sync information (for example, whether a Sync message can be interpreted independently from other Sync messages.

3. **State Change Notification:** It is the responsibility of the Sync protocol to *promptly* inform all participants in an application group of data production updates. To support large and dynamic groups, Sync should not require each participant to know individual group members. One shared feature among all previous Sync protocol designs is the use of *network multicast* to forward Sync Interest to all participants in an application, i.e. the members in a *Sync group*; whoever has data with a matching name sends a Data reply to the Sync Interest sender.

   In the previous Sync protocols, we observed two different usages of replies to multicast Sync Interest. In the first usage, every participant sends a multicast Sync Interest to *pull* new dataset state changes, and receives the new dataset *state changes* in the reply Data packets. In the second usage, data producers directly encode the newly produced data in the reply Data packets as a response to a multicast Sync Interest.

We would like to emphasize that *Sync Interest multicast* differs from *multicast data delivery*. NDN has built-in support for the multicast data delivery, which is achieved by NDN routers aggregating data Interests carrying the same data name. On the other hand, Sync Interest multicast requires multicast routing support from the network to forward Sync Interest packets to all participants in a Sync group. NDN multicast routing support is yet to

be fully developed; the current NDN network implementation imitates multicast forwarding by broadcast forwarding, which is an inefficient way to achieve the same goal.

3.2 The Evolution of NDN Sync

Several NDN Sync protocols have been developed over the years [MPW22], each is designed based on lessons learned from earlier versions. In this section, we describe the evolution of NDN Sync protocol designs by describing the representative examples. In particular, we discuss the various choices for each of the three Sync design components described in §3.1, while grouping these protocols by the dataset state encoding they use.

3.2.1 Digest Encoding

The earliest Sync protocol, CCNx Sync [Pro12, Mos14], represents the namespace as a hierarchical name tree. The CCNx Sync design computes a digest at each node in the tree from the bottom to the top and uses the root digest to represent the dataset state. CCNx Sync's hierarchical name tree can scale in terms of the number of items in the dataset, but receiving an unmatching digest triggers a costly walk down the tree to find the changing piece. *ChronoSync* [ZA13] was the first Sync protocol to be used in applications, and takes the same approach of using a single digest to represent the dataset state. However, ChronoSync was the first Sync protocol to use sequential naming as described in §3.1, representing the namespace as a list of [producer name, seq#] tuples instead of a name tree. The *state digest* in this case is defined as a cryptographic hash over an encoded list of these tuples.

In the ChronoSync protocol, each participant P multicasts a Sync Interest carrying its state digest H. Receiving H informs all participants in the group of the P's dataset state, and serves as a request for any changes to the dataset produced by other group members. Each such *long-lived* Sync Interest is kept pending in the network with a *persistent PIT entry*, waiting for the next dataset change notification. In steady state, all participants are

synchronized and send identical Sync Interests which are aggregated within the network. In the absence of replies, each participant refreshes the pending Sync Interest periodically (i.e. every Sync period) with some randomized jitter.

When one of the participants R produces new data, the dataset state and state digest at R changes. R then replies to all locally pending Sync Interests with the new state information. R then refreshes its own Sync Interest with the new updated digest. Assuming no packet losses, the data reply from R containing the state change notification is multicast to all other participants in the Sync group, who subsequently update their own local state and digest to match that of R.

Using a simple digest encoding produces very compact messages but comes with downsides. When a participant receives an incoming digest D_r that differs from the local state digest, the receiver does not know the exact namespace differences. This means the receiver P cannot know if the sender has older or newer state compared to P. ChronoSync participants maintain a log of recently seen digests. If the received state digest D_r is found in the digest log, it indicates that the sender is lagging behind P. In this case P will send a reply with its current dataset state to update the sender's state. If D_r is not found in the log, P waits for a random time period for potential incoming Sync replies that may carry any newly produced state that is unknown to P. If no reply is received in time, P multicasts a *recovery* Interest carrying D_r, soliciting the full dataset state corresponding to D_r from either the sender of D_r or any other informed group member.

We note two causes that may lead to unrecognized state digests. First, packet losses may cause members to miss updates. For example, if the reply to a Sync Interest sent by P is lost, P will now have outdated state compared to one or more other group members. The second case is that of simultaneous data publication. NDN's flow balance principle requires that one Interest retrieve at most one Data packet. If two participants change the dataset state simultaneously, only one data reply to the pending Sync Interest will be propagated to the sender of the Sync Interest, and the other reply will be dropped. As a result, the

sender will now have a partially outdated state, and will send out an unrecognizable state digest. In both cases, ChronoSync will eventually reach synchronized dataset state after some additional round trips.

3.2.2 Encoding with Invertible Bloom Filters

The design outcomes of ChronoSync suggest that each Sync Interest should carry additional information to help infer the exact dataset state differences to enable the protocol to reconcile dataset changes more effectively. Some of the subsequent protocols utilized the Invertible Bloom Filter (IBF) [EGU11] data structure for this purpose. The IBF is a probabilistic space-efficient encoding allowing membership queries and set difference calculation. Each Sync Interest can carry an IBF, and receiving an IBF allows the recipient R to calculate the difference between its own state (f_1) and the sender S's state (f_2). An important caveat is that a data name must be hashed into a number before being inserted into an IBF. As a result of this one-way hashing, R can infer only the names that S does not have i.e. $f_2 - f_1$, but it cannot infer what names are missing in R's local dataset. We briefly discuss three IBF-based Sync protocols below.

1. iSync [FAC15] was the first Sync protocol to use an IBF. iSync uses all application names as the dataset state. A 2-level IBF structure is used to encode the dataset to allow scaling to a large number of items. First, the dataset is divided into multiple *collections*. Each collection's publications are encoded in a *Collection IBF*. The collection IBFs are then grouped together and encoded in a top level *Sync IBF*.

   The multicast Sync Interest in iSync carries the digest of the top-level Sync IBF. Unlike ChronoSync, however, these Sync Interests do not solicit replies, instead acting as pure notifications of inconsistent state. When a digest is received that does not match the receiver P's digest, P fetches the Sync IBF directly from the group. On receiving the Sync IBF, P can identify and fetch the differing Collection IBF(s). Finally P,

computes the set difference for the given collection and fetches the missing data names with an Interest carrying the exact set difference.

We make two noteworthy observations regarding the iSync design. First, iSync utilizes multicast Interests as pure notifications, and thus does not suffer from the problem of lost data replies. Since subsequent Interests are unicast, iSync can better support simultaneous publications. Second, since iSync uses application names directly, the dataset size increases continuously, requiring larger IBFs. Theoretically, it is possible to further extend the 2-level IBF hierarchy at the cost of additional complexity and latency due to extra round trips.

2. syncps [Nic19] directly uses application names for Sync, similar to iSync, and encodes the entire dataset of hashed names in a single IBF. Sync Interests in syncps, however, carry the IBF directly as a component in the Interest name. These Sync Interests are multicast to all group members. When a participant P receives a Sync Interest, it computes the set difference. If both states are identical, the Interest is allowed to stay pending in the network, similar to ChronoSync's long-lived Sync Interests. If the set difference is non-empty and P carries some data items that the sender does not, P sends a *Sync reply*, which is a data packet containing as many missing data items as possible. The size of the Sync reply is constrained by the network MTU; if some data items do not fit in a single reply packet, then they are synchronized in the next round of Sync. We note that syncps directly synchronizes the application data itself rather than just the namespace dataset. This approach is suitable when all participants desire to fetch all data produced within the group.

   Unlike iSync, syncps uses a single IBF to encode all items in the dataset, and the size of this IBF is limited by the network MTU since it must be carried in the Sync Interest. An increasing number of data names in the same IBF size increases the risk of false positive IBF lookups. To enable continuous data production, syncps removes data names from the dataset after a predefined *lifetime* period. If a particular data

name is not synchronized within its lifetime, it is dropped permanently. We note that syncps uses absolute timestamps for publication expiry, and thus must assume some limits on the acceptable clock skew within Sync group participants. This maximum value of clock skew is used to determine a grace period for preventing race conditions; during the grace period, dataset names are not included in a participant's outgoing Sync Interest but are recognized as known when examining received IBFs.

Since syncps uses replies to multicast Sync Interests, the design suffers from similar issues as ChronoSync regarding simultaneous publications. Further, if the process of recovery takes longer than the predefined lifetime period, the expired data items are lost permanently without being propagated to all group members. In other words, syncps does not guarantee reliable dataset synchronization.

3. PSync [ZLW17] supports either one-way *partial sync* or bidirectional *full sync*. We focus on the full sync mode which performs similar functions as the other protocols described in this section.

   PSync encodes the dataset in a single IBF and carries the IBF in a long-lived Sync Interest similar to syncps. The key difference is that PSync uses sequential naming, and the data names carried in the IBF look like /producer-name/latest-seq#. PSync replies contain the data names missing in the sender's IBF, and do not directly push the missing application data to the sender.

   The IBF in a PSync group scales with the number of producers rather than the number of names in the dataset. This is a result of PSync's sequential naming, which makes it possible to scale Sync without added protocol round trips, as in iSync, or removing old items from the dataset as in syncps. However, PSync's multicast Sync Interests also solicit data replies, leading to performance issues with simultaneous publications similar to ChronoSync and syncps.

3.2.3 Raw State Encoding

In dynamic environments, it is challenging to reliably complete multiple rounds of protocol exchanges that are required by digest and IBF-based Sync protocols. VectorSync [SAZ18] and subsequently DSSN [XZL18] and DDSN [LKM19] first proposed using the raw namespace representation directly for encoding. Both protocols utilize sequential naming, with the dataset being represented as a list of [producer name, seq#] tuples. VectorSync uses a list of sequence numbers for encoding the dataset state on the network. Receivers can directly compute difference between the received state and the known local state. However, VectorSync assumes a known view of group membership, since the dataset encoding does not carry this information. This requires additional synchronization for group membership. DSSN aims to support asynchronous communication in IoT environments, where such additional exchanges may be infeasible. In DSSN, the group membership is directly encoded along with the sequence numbers. Thus, each Sync Interest directly contains an encoded list of [producer name, seq#] tuples. However, DSSN Sync Interests solicit data replies to correct the missing state, thus continuing to suffer from the problem of simultaneous publication, as described for protocols such as ChronoSync.

We note that all the protocols discussed above with the exception of iSync use multicast Sync Interests to solicit Data replies, leading to two problems. First, long-lived Interests accumulate persistent state in the network, and may also be lost without feedback making recovery difficult. Second, multiple replies may lead to different members getting different data; as a result, these protocols suffer from the issues with simultaneous publication.

The State Vector Sync protocol design takes lessons learned from all of the above protocols. In the next section, we describe this new Sync design in detail. We note here that the PLI-Sync [HSW21] and ICT-Sync [APD18] protocols were developed in parallel with SVS. These protocols adopt sequential naming and use state vectors similar to SVS, but differ in other unique ways [MPW22].

3.3 State Vector Sync Protocol

In this section, we describe the design of the State Vector Sync (SVS) protocol. A distinct goal of the SVS design is the ability to work efficiently and reliably in both infrastructure-based and ad-hoc mobile network environments. The design of SVS reflects lessons learned from previous Sync protocols.

1. Sequential naming is a requirement for Sync scalability and resiliency. SVS represents the dataset namespace using sequential names, similar to protocols such as ChronoSync and PSync. Thus, the dataset state for purposes of Sync is defined as a list of [producer name, seq#] tuples. When applications desire to use their own names, SVS can utilize a mapping from the sequence number to the application data name (§3.7).

2. SVS uses raw state encoding in Sync Interests. Receiving a single Sync Interest informs the receiver of the entire state of the sender, allowing for faster recovery from losses.

3. Multiple replies to multicast Sync Interests may be dropped in the network leading to desynchronization. SVS uses notification Sync Interests instead to address this issue.

During operation, producers in an SVS Sync group increment their own sequence number whenever they produce new data. SVS then promptly propagates this change to the shared dataset to all other participants in the group using multicast Sync Interests, informing them about this newly produced data. We describe the state encoding used by SVS and processing of Sync Interests below.

3.3.1 Dataset State Encoding

At a high level, SVS uses raw dataset encoding. Each producer in an SVS Sync group is associated with a sequence number. An SVS *state vector* is a list of the tuples of [producer name, seq#] encoded using the NDN TLV [Nam25e] encoding specification, following the

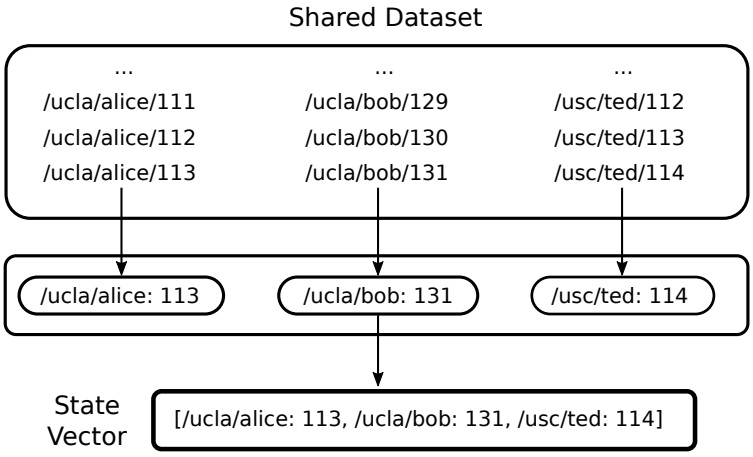

Figure 3.1: Dataset Encoding in SVS

conventions described in the SVS specification [Nam25h]. The representation of the namespace dataset and the corresponding encoding of the state vector is illustrated in Fig. 3.1.

An encoded state vector is wrapped inside a Data[1] packet signed by a Sync participant. These Data packets in turn are carried in the Application Parameters field of a Sync Interest. This design decouples the security of the Sync protocol from the mechanism used to propagate the Sync Interest, as discussed further in §3.8.

The real-world usage of Sync with applications such as the NDN Workspace [Ma24] and mGuard [DAT22] highlighted a new issue in dataset representation. Each producer in a sequentially named dataset is identified by the identity of the producer. However, one producer identity may bootstrap into the same Sync group multiple times, for example after a catastrophic loss of local state, or on a different device. In this case, the two instances should have different sequence numbers associated with them to prevent conflicts and to preserve the immutability of data. In Version 3 of SVS, these instances are uniquely identified in the dataset using the *bootstrap timestamp*[2] for that particular instance. Thus, each producer

[1]Before SVS Version 3, the protocol used signed Interest packets, with the state vector contained in either the name or the Application Parameters field of the Sync Interest.

[2]Using a monotonically increasing instance identifier allows pruning old instances from the state vector. If collisions are likely, the identifier may be partially randomized instead.

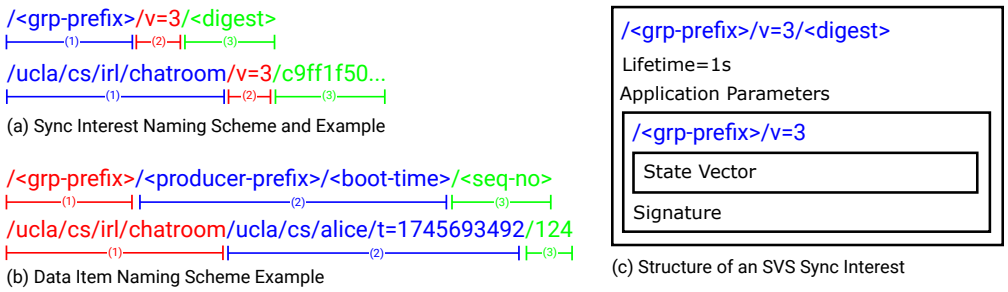

Figure 3.2: Naming and Sync Interest structure in SVS

name is associated with one or more instances, each with a unique bootstrap timestamp, and each instance has its own sequence number. This change is also directly carried over to the state vector encoding in the Sync Interest.

The naming scheme and structure of an SVS Sync Interest is highlighted in Fig. 3.2 with an example. It should be noted that SVS does not enforce a Data naming convention, and the illustrated Data naming scheme is merely a suggestion that provides uniqueness and can be directly used for Interest forwarding.

3.3.2 Sync Interest Generation and Processing

In SVS, the namespace is synchronized using multicast Sync Interests. These Sync Interests act as pure notifications and do not solicit Data replies[3]. Receiving a Sync Interest sent by another member in the group informs the receiver of the exact dataset state of the sender. The receiver may then compare the received state with the locally known state and perform one of several possible actions. We describe the process of Sync Interest generation and the processing of received Sync Interests below.

Every member of an SVS Sync group sends a Sync Interest under two conditions. When an SVS instance observes a new dataset state change locally, or receives a Sync Interest with

[3]Notification Sync Interests represent a design trade-off – these violate NDN's feedback mechanism, but resolve issues such as long-lived PIT entries and lost data replies during simultaneous production.

obsolete dataset state, it generates an *event-driven* Sync Interest. In the absence of any event-driven Sync Interests, a *time-driven* Sync Interest is sent periodically. Event-driven Sync Interests help disseminate new dataset state information through the network with minimal delay, while periodic Sync Interests maintain eventual consistency in the group dataset state in the face of packet losses and transient network partitions.

First, we examine event-driven Sync Interests. When a participant P in an SVS group produces a new piece of data, P increases its own sequence number by one. To inform others about this change in the dataset state, P will immediately emit a new Sync Interest that carries the new dataset state. When other Sync participants receive the Sync Interest, they will notify their local applications of the new data name; the application may then decide to fetch the newly produced data. On the other hand, if a participant does not receive this Sync Interest, for example due to packet losses, it will now have an outdated dataset state compared to P.

In the reversed case, when P receives a Sync Interest I_r from another member of the group, we have two possibilities. If the received state in I_r is the same or strictly newer than P's local state, the sender has the latest state known to P. If the received state in I_r has one or more sequence numbers that are outdated compared to P's local state, then P sends an event-driven Sync Interest, hoping to correct the sender's dataset state. In either case, if the sender has any newer state, P updates its local state with this information and propagates it to the application.

In the absence of any data production, the state of the Sync group does not change and no event-driven Sync Interests are produced. In this case, SVS instances send a Sync Interest periodically, so that the group's state is eventually consistent even if some event-driven Sync Interests were lost, for example, due to packet losses. In the next section, we describe the functioning of SVS timers in detail and the optimization of Sync Interest generation to reduce protocol overhead.

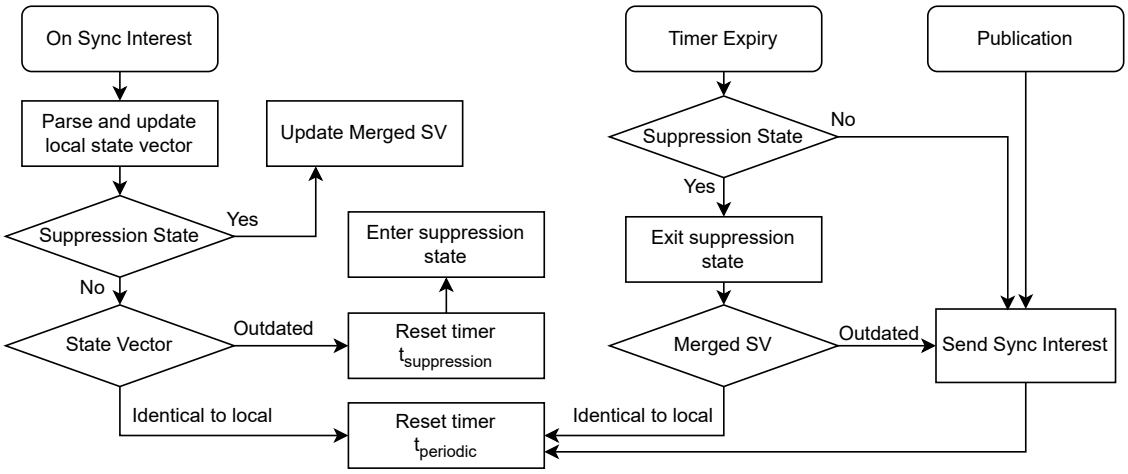

Figure 3.3: Sync Interest suppression in SVS

3.4 Loss Recovery and Sync Interest Suppression

In the event of packet losses or network partitions, some participants may have an obsolete dataset state compared to other members in the Sync group. As described in the previous section, SVS persistently synchronizes the dataset state of all group members by multicasting the latest Sync interest to the group. A Sync Interest can be triggered in three ways: when a member produces a new piece of data, when a member receives an outdated Sync Interest, and in the absence of any event, when a periodic timer goes off.

Each sync group member uses a single timer for loss recovery that takes on one of two values. In *steady state*, when no data is produced and no Sync Interests are received, the timer is set to a relatively longer *periodic timeout* value. When the SVS timer expires in steady state, the instance sends out a Sync Interest carrying its local state, i.e. the *time-driven* Sync Interest. If a participant P receives a Sync Interest that is identical or newer than its local dataset state, P resets its timer to the periodic timeout value. This is because since all Sync Interests are multicast to the entire group, there is no rush for P to repeat the same information that the group just received. This process, termed as *Sync Interest Suppression*, reduces the protocol overhead and ensures that approximately only one Sync

Interest is generated by the whole group for every period in steady state.

When P receives a Sync Interest I_r that is outdated compared to its local state, P needs to notify the sender of the latest dataset state. However, we note that all other members of the Sync group also receive the same I_r with multicast, and may act similarly as P. This may lead to a flood of redundant Sync Interests generated by various group members, all intending to correct the state of the sender of I_r. To minimize this redundancy, each member in the group waits for a random period before sending the Sync Interest, referred to as the *suppression time*. On receiving I_r, P enters the *suppression state* and sets the SVS timer to a randomized and relatively short *suppression timeout* value. If P receives an up-to-date Sync Interest while in suppression state, P resets the timer to the periodic timeout value, assuming that the received Sync Interest would also correct the dataset state at the sender of I_r. If the suppression timer expires without receiving any such Sync Interest, P sends out its own Sync Interest.

In some cases, due to network delays and simultaneous data production, P may not receive a single up-to-date Sync Interest, but may receive multiple state vectors that together would correct the missing state in I_r. In this case, one can expect that these Sync Interests also updated the state of the sender of I_r. To account for this possibility, P merges all incoming state vectors into a *merged* state vector while in suppression state. When the timer expires, if the merged state vector is identical to P's latest known state, P suppresses its outgoing Sync Interest and resets the timer to the periodic timeout value.

The Sync Interest suppression mechanism prevents a flood of Sync Interests that may be triggered in response to a single outdated Sync Interest sent to the group. Fig. 3.3 illustrates this process with a flow chart. The performance of SVS on metrics such as synchronization delay and protocol overhead are significantly affected by the exact values that the loss recovery timer assumes. We describe various optimizations for these timer values below.

3.4.1 Tuning SVS Timers

Choosing SVS timeout values represents a trade-off between synchronization delay and protocol overhead. Setting the periodic timeout to a shorter value can correct outdated state sooner, at the cost of a larger number of periodic Sync Interests. Similarly, a shorter suppression timeout would correct the state of the sender of an outdated Sync Interest faster, with the risk of generating more redundant Sync Interests, since a Sync Interest generated by another member of the group may still be in transit in the network.

First, we describe the randomization of the timeout values used by each timer. For the periodic timer, approximately one Sync Interest should be generated for every period by the entire group. In this case, the timer value is randomly chosen using a jitter around the periodic timeout value. For example, a group may calculate the periodic timeout value as $30\text{s} \pm 10\%$. As a result, on receiving a Sync Interest in steady state, all participants will set a random periodic timer that is approximately 30s. When the timer expires at the node with the shortest timeout, the cycle resets for approximately another period.

We note that a shorter period is more likely to correct inconsistent state among the participants, at the cost of higher packet overhead. However, if the frequency of data publication in a Sync group is high, the periodic timer would likely get reset before its expiration. A shorter period could also greatly benefit SVS participants in ad-hoc networks with poor connectivity. If a group member can communicate with another group member via ad hoc connectivity, a shorter period increases the likelihood of a state exchange during this transient connectivity period, improving dataset consistency. Under dynamic conditions, it may be possible to further optimize the timer range depending on factors such as the current data production rate in the group.

The requirements for a good suppression timer value differ from those for the periodic timer. In this case, it is desirable if *exactly one* Sync participant R sets a very short timeout (ideally zero) in response to receiving an outdated Sync Interest, while all other participants

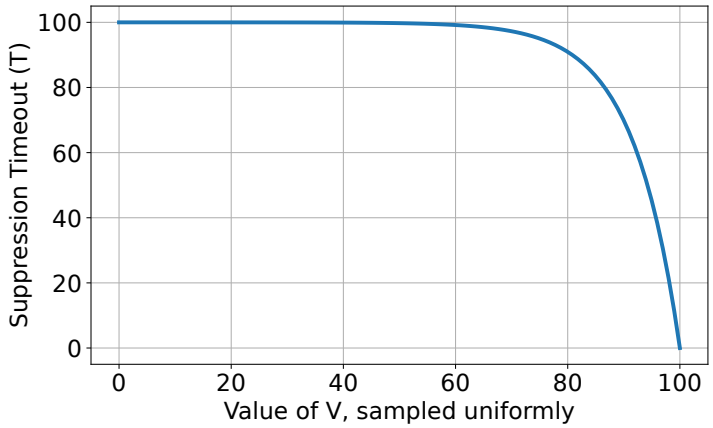

Figure 3.4: Suppression Timeout sampling in SVS

set a much longer timeout. This would ensure that P_a's outgoing Sync Interest will be received by all or most other participants, which in turn would suppress their redundant outgoing Sync Interest. To achieve this effect, we suggest using the following exponential curve to randomly sample the suppression timeout value.

$$T = C \times (1 - e^{\frac{V-C}{C/F}})$$

where:

T is the calculated suppression timer value

V is a uniform random value in $(0, C)$

C is a constant *Suppression Period*

F is a constant *Decay Factor*

Fig. 3.4 illustrates a plot of the curve with $F = 12$ and $C = 100$. The two constant parameters must be approximated depending on network conditions. C is the maximum suppression time that any node may choose, and must be equal to at least the network latency diameter. This allows for a Sync Interest generated by P_a to reach all other group

members in time. F must be positively correlated with and can be set equal to the number of active participants in the Sync group. This value ensures that only a small number of members, ideally exactly one, picks a short suppression timeout value while others pick relatively long values. Future SVS research may focus on effectively learning these constants from network conditions, using active or passive measurements, or a combination of both.

Finally, we note one simple yet effective optimization. Some producers may produce data at a very high rate, so that their sequence numbers can change rapidly. If one Sync Interest is generated for each change in the sequence number, one could burst out too many Sync Interests. To counter this, members producing data at a high rate should upper-bound the frequency of their Sync Interests by setting the SVS timer to a short period T_h on any dataset change, and withhold Sync Interest generation during this *throttle state*. When the timer expires, SVS will send out an Sync Interest with the latest sequence number, informing others in the group about all data production during this time with a single message.

3.5 Evaluation

We conducted an evaluation of SVS [Nam25b] using its C++ implementation, and compared the results with three other Sync protocol implementations, namely ChronoSync, PSync, and syncps. In our evaluation setting, we emulated the 45-node GÉANT [GÉ18] network topology using MiniNDN [Min25]. Each node was set to have unlimited bandwidth and a 10ms propagation delay. In each experiment run, we randomly selected 20 nodes to be participants in a Sync group. To observe each Sync protocol's performance under different conditions, we varied the loss rate of each link from zero to 20%, and the publishing rate of each Sync group member from 1 data item every 15 seconds to 2 data items per second. In the evaluation plots, the X-axis indicates the data publication rate of the *entire* group.

We experimented with setting the Sync Interest lifetime of ChronoSync, PSync, and syncps, the Sync protocols that deploy *long-lived Sync Interests*, to 1 sec, 4 sec and 10 sec;

the results reported use 1 sec as the Sync Interest Lifetime if not specified. For other configurable parameters such as various timer values, data freshness periods, IBF size etc., we used the suggested default configuration values in individual protocols' implementations. Our evaluation metrics include,

1. **Sync latency**: the time period between a data item's generation time and the time its notification reaches another member. We calculated the percentiles for individual values which are reported in the results.

2. **Sync protocol overhead**: for each published data item, we measured the total number of Sync Interest and reply Data packets received at every NDN forwarder including all end nodes

3. **Reliability**: the percentage of participants that successfully received new publication notifications by the end of the given run.

Each evaluation run lasted approximately 150 seconds. Error bars in the figures denote the 95% confidence interval from 10 runs of every emulation setting. We first describe the evaluation results from each individual protocol and then compare them.

Fig. 3.5 shows ChronoSync's performance measured by the group synchronization latency. In the absence of packet losses, ChronoSync performs well at low publication rate; as publishing rate increases, simultaneous publications become more likely, which lead to unrecognizable state digests. This leads to a rapid increase in latency, which stabilizes quickly since a large fraction of Sync cycles fall back to recovery Interests. Compounding simultaneous publications with packet losses further deteriorates performance.

Fig. 3.6 shows the evaluation results for syncps. Thanks to its use of IBF in namespace encoding, syncps significantly improves the Sync latency as compared to ChronoSync: with 7 data items per second and without losses, syncps's 90-percentile Sync latency is around

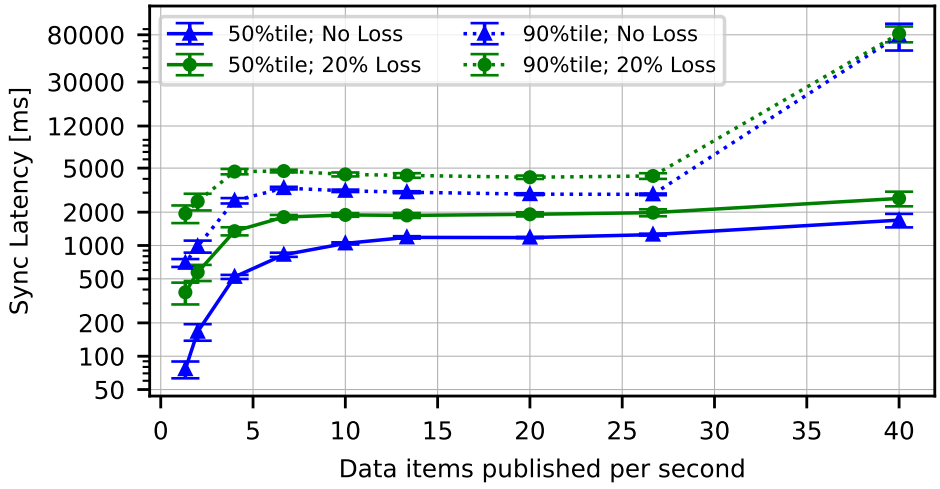

Figure 3.5: Sync Latency of ChronoSync

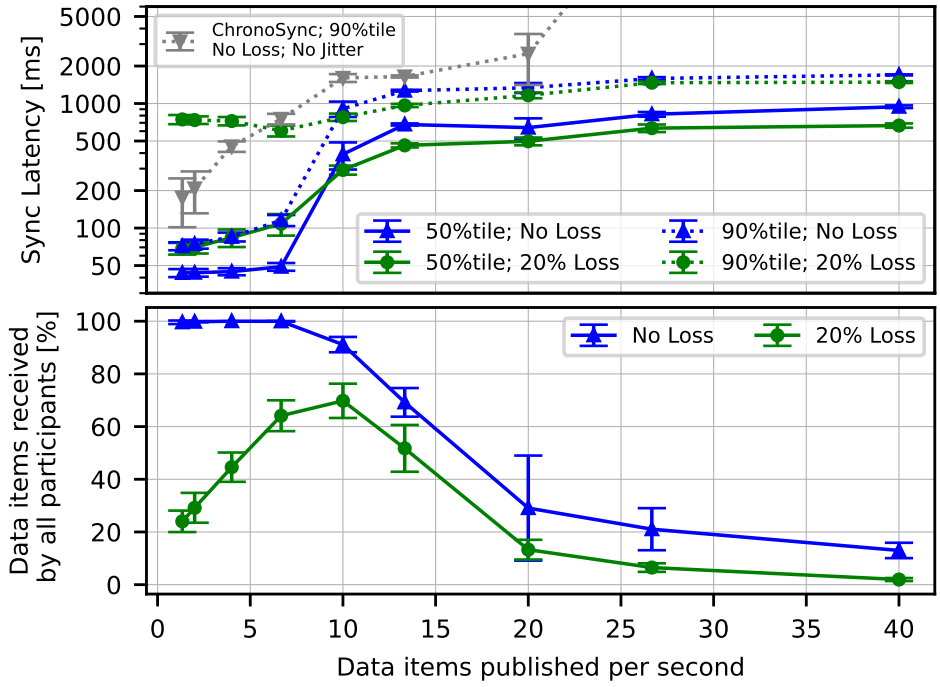

Figure 3.6: Sync Latency and reliability of syncps

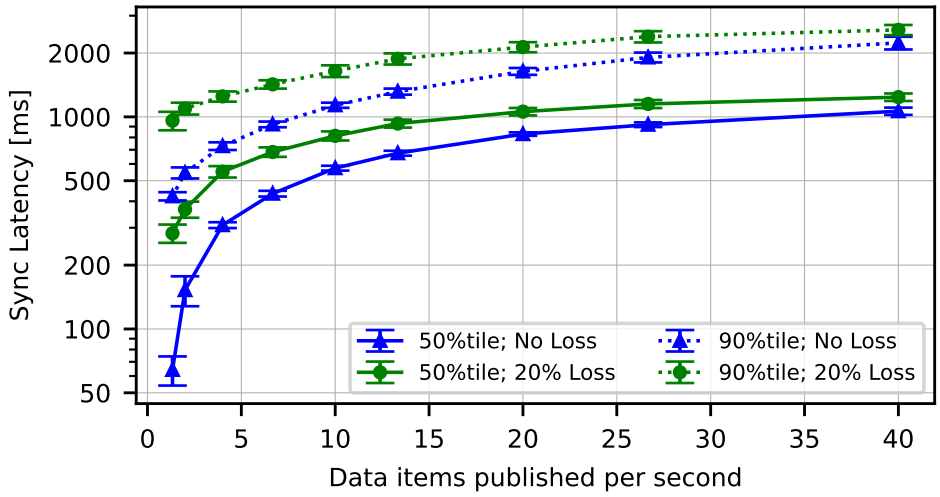

Figure 3.7: Sync Latency of PSync

100 msec while the same measure for ChronoSync is almost 1 sec[4]. However, as described in §3.2, syncps drops publications after reaching a predefined lifetime[5]. This results in a decrease in reliability of synchronization as the data publication rate increases[6].

Our results for PSync in Fig. 3.7 show a higher observed latency at lower publication rates compared to syncps, but the observed delays were similar as the publication rate increased. However, we must note here that PSync achieved 100% reliability at the end of the experiment, unlike syncps. This change can be attributed to PSync's usage of sequential naming.

Unlike the other Sync protocols, the evaluation in Fig. 3.8 show that data publishing rate has no impact on SVS Sync latency, and adding packet losses only has a small impact. Increasing the publication rate leads to more frequent event-driven Sync Interests, better compensating packet losses, thus making the Sync latency drop. Conversely, Sync latency gets worse when publications are infrequent in the presence of losses, since not enough

[4]To enable this direct comparison, we disabled jitters in the ChronoSync implementation.

[5]Our evaluation used the default publication lifetime of 2 sec

[6]For all other protocols except syncps, 100% reliability was observed indicating eventual consistency at the end of the experiment.

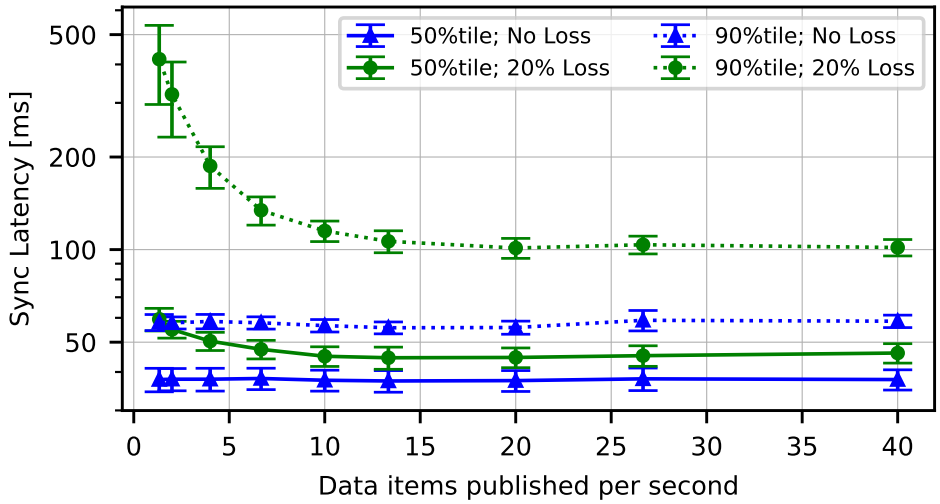

Figure 3.8: Sync Latency of SVS

redundant Sync Interests are generated to overcome the packet loss. Thus, SVS reverses the correlation observed between Sync latency and the data publication rate when compared with other Sync protocols.

The comparative evaluation results illustrated in Fig. 3.9 demonstrate the drastic reduction in Sync latency achieved by SVS when compared to previous NDN Sync designs. We also observe, as shown in Fig. 3.10, that SVS continues to have lower overhead measured in terms of the number of packets exchanged for every publication. We attribute this improvement in the performance of SVS to three important design choices.

1. Since SVS does not solicit replies to multicast Sync Interests, it does not suffer from the problems arising from simultaneous data publication. These issues become more common as the publication rate increases. Since each event-driven Sync Interest in SVS is independent of any other in the network, we observe a very low latency of synchronization (effectively equal to the average round-trip delay in the group in the absence of packet losses) that approximately remains constant regardless of the data publication rate.

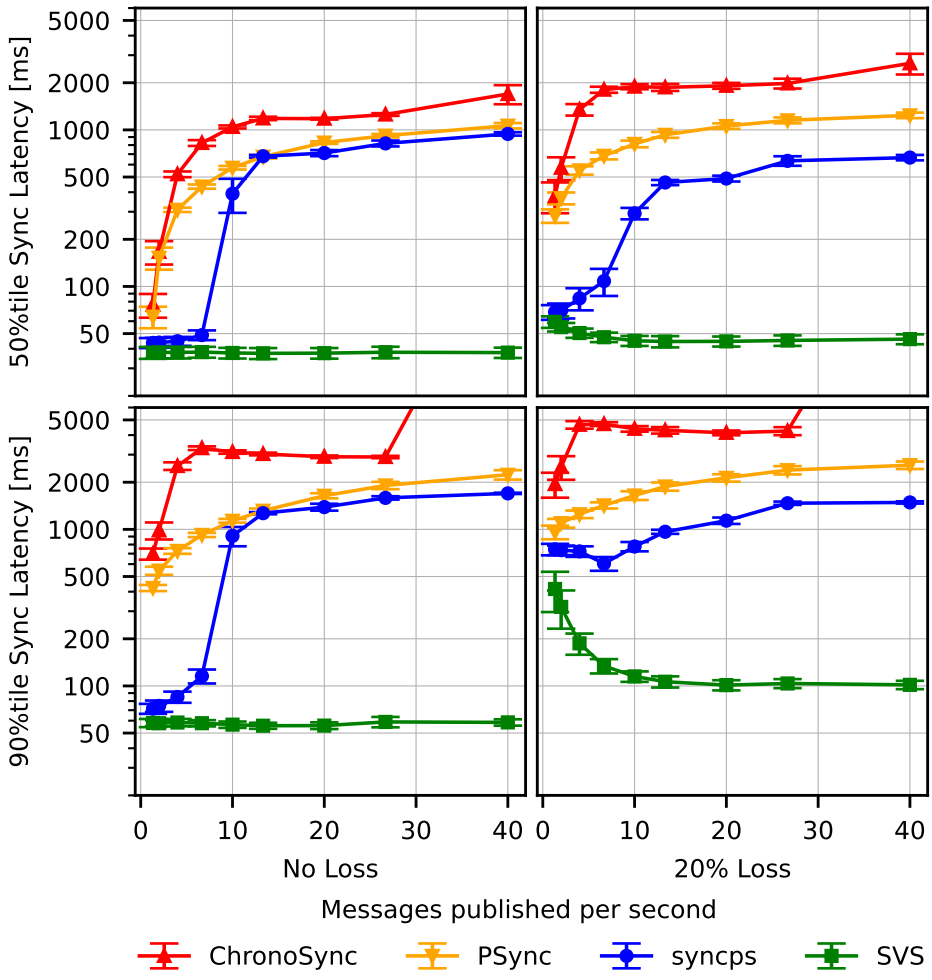

Figure 3.9: Sync Latency Comparison

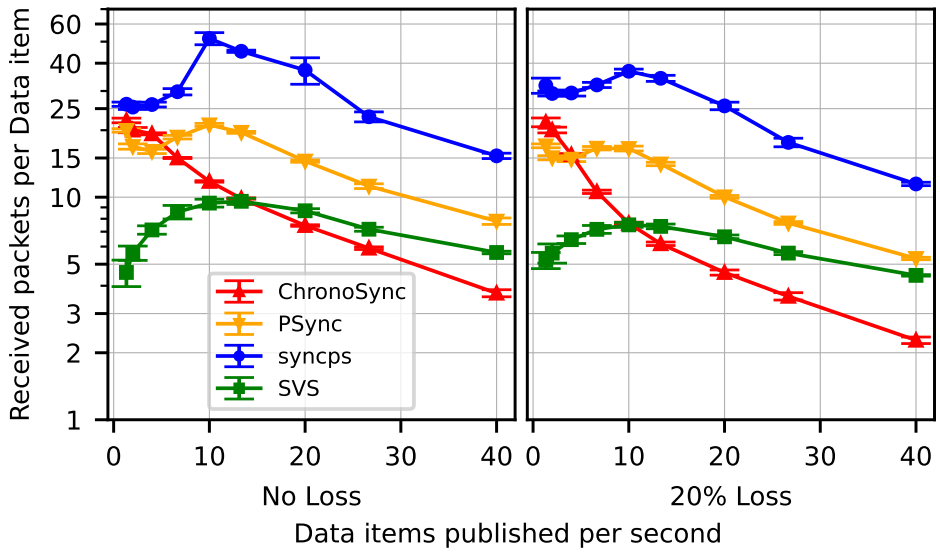

Figure 3.10: Sync Overhead Comparison

2. SVS Sync Interests overcome losses more easily, especially with an increase in the publication rate. This is due to the increase in redundant information exchanged on the network, which in turn is a result of SVS carrying the entire raw state in each Sync Interest. Receiving a single message informs a receiver of the entire state of the sender without requiring further loss-prone round trips, making SVS more resilient to packet losses.

3. Owing to its simple design and effective Sync Interest suppression, SVS can achieve much lower overhead compared to other Sync protocols. Suppression also becomes more effective with an increase in the publication rate, since duplicate Sync Interests may also be suppressed by an unrelated data production from another Sync participant. We note here that we only measured the total number of *packets* exchanged on the network; however each SVS Sync Interest may be larger compared to other Sync protocols, and grows with the size of the Sync group. We discuss this scalability of the SVS Sync Interest in further detail in §3.6.

We also note that unlike all other Sync protocols, syncps pushes application data to

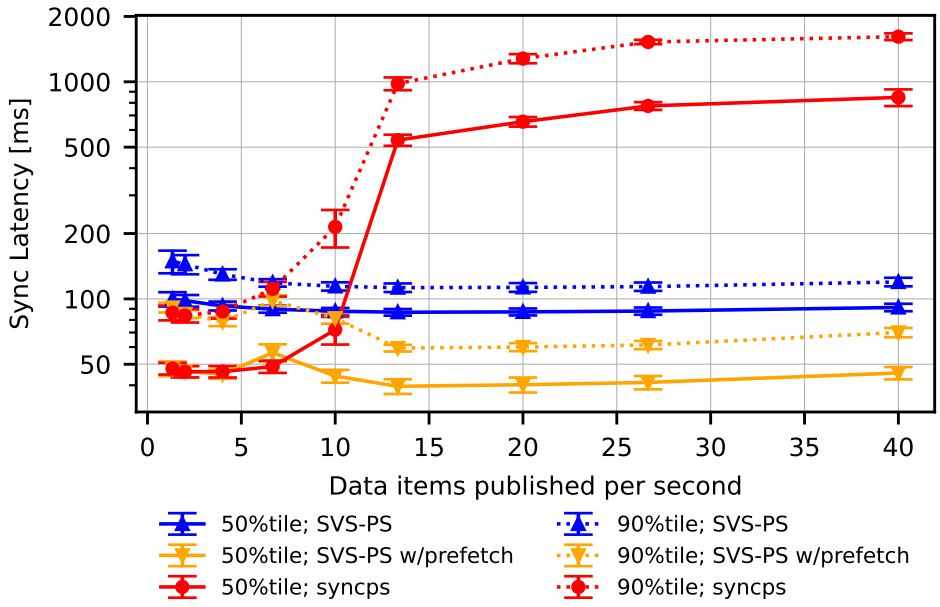

Figure 3.11: Sync Latency of SVS-PS vs syncps

all Sync participants rather than just synchronizing the namespace dataset. To ensure a fair comparison, we compared the Sync latency of syncps with SVS-PS, a publish-subscribe layer built over SVS further described in §3.7. SVS-PS can run in two distinct modes. In normal operation, SVS first synchronizes the namespace dataset and the application then decides which data to fetch. In a second low-latency mode, all data is *prefetched* by the Sync participant by sending Interests to fetch the data in anticipation before it is actually produced. Fig. 3.11 shows the comparison in latency between syncps and SVS-PS in each mode of operation. As we can observe, SVS-PS continues to perform comparably at lower data publication rate and provides a large improvement in latency as the data publication rate increases.

Our evaluation results demonstrate that SVS improves over the previous Sync design in both performance metrics of latency and overhead, while retaining reliability and eventual consistency. In the next few sections, we provide further discussion on topics such as scalability and application data naming.

3.6 Scaling to Large Groups

SVS Sync Interests carry raw encoded dataset state directly, instead of using compressed encoding such as a digest or IBF. While this allows SVS to perform better and be more resilient in the face of losses, it also raises concerns about the size of each Sync Interest. The state vector, and thus the Sync Interest, grows linearly with the number of producers in the Sync group. This means the overhead of SVS in terms of the number of bytes transmitted would also increase linearly in this regard. Further, the number of producers would be upper-bound by the network MTU size, since each Sync Interest is a single NDN packet.

To enable SVS to scale to a large number of producers, we first note that each tuple of [producer name, seq#] in the dataset state has no dependency on any other tuples in the state vector. This means one may include a *partial* set of tuples in any Sync Interest, and this set would still be directly usable. Such a design, named p-SVS [PSX22], converts the problem of SVS scalability to the question of how to choose an ideal subset of state vector to send in each Sync Interest. On observing the nature of applications using Sync, we can identify two conflicting goals in the creation of a Sync Interest carrying a partial state vector.

1. In real distributed applications, not all producers are expected to actively produce data at all times. Therefore, one may choose to only include tuples of producers that have produced data most recently in the outgoing Sync Interest. This design improves the efficiency of Sync by carrying less redundant information which is likely already synchronized. In the p-SVS design, this approach is referred to as the *Recent* strategy.

2. However, we must take into account packet losses and network partitions, which may cause participants to miss some Sync Interests. If Sync Interests carry only the recently updated producers, the state of some producers may never be propagated if the initial few Sync Interests were lost after they last produced data. This means that the *Recent* strategy does not provide any time-bound on eventual consistency in the group. Thus, a second proposed strategy to speed up consistency in these cases is to randomly

select the subset of producers to include in each Sync Interest. This approach provides a statistical guarantee that each producer's state will eventually and periodically be included in a Sync Interest. This approach is referred to as the *Random* strategy.

3. We can now combine the two strategies above as a middle ground, and include the states of both a subset of recently updated producers and a subset of randomly selected producers in the Sync Interest. We call this the Random-Recent or *RandRec* strategy of building a p-SVS state vector.

We evaluated these strategies together with SVS carrying the entire state vector, and a segmented SVS implementation which breaks up the entire state vector into multiple Sync Interests, but sends these Sync Interests in quick succession. We set up a grid topology of 8×8 nodes with all 64 nodes participating in a p-SVS Sync group. Each node randomly produced data every 1 sec, while a smaller number of *fast* producers produced data every 100 msec. We varied the aggregate group publication rate by varying the number of *fast* producers from 0 to 38. Each network link was set to have a 10 msec propagation delay and a 50% packet loss rate[7].

We used the 95%tile latency of synchronization and the total network traffic as metrics for our comparison. The 95%tile latency is defined as the delay for 95% of participants to learn about a given data production. An MTU of 30% of the size of the entire state vector was artificially simulated when testing partial state vectors. For the baseline, we also evaluated SVS sending the entire state vector (Base), and SVS with segmented Sync Interests (SegBase), where the complete state vector is segmented into 4 Sync Interests and these segments are sent in quick succession.

Fig. 3.12 compares Sync latency between the different strategies. For all strategies, the latency reduces with an increase in publication rate, as discussed previously in §3.5. As may be expected, the Random strategy has the worst performance, with the gap getting

[7]We use a very high loss rate since a grid topology provides rich connectivity.

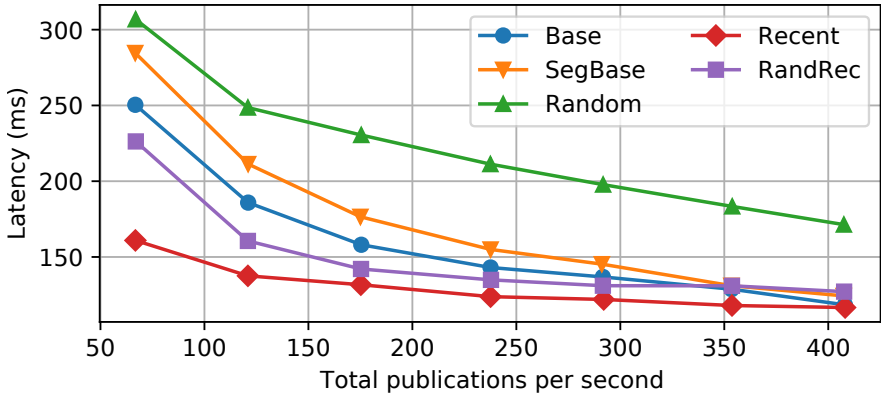

Figure 3.12: Sync Latency of p-SVS strategies

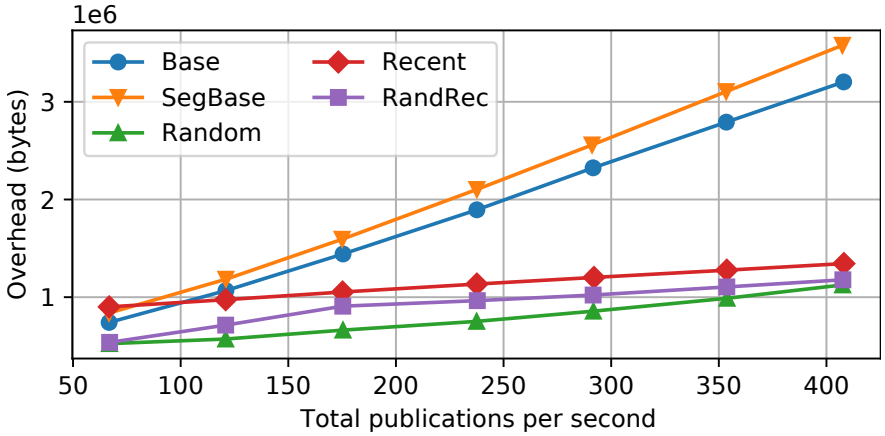

Figure 3.13: Network overhead of p-SVS strategies

worse with an increase in the frequency of publication. As the publication rate increases, the Random strategy is more likely to miss some of the latest data production, leading to this effect. The latency of the Random-Recent strategy closely follows baseline SVS, and is lower at a low publication rate likely due to a reduced packet size. The Recent strategy has the lowest Sync latency; however this strategy does not necessarily provide eventual consistency.

Fig. 3.13 shows a comparison of the network overhead of p-SVS calculated as the total number of bytes transmitted. The naive segmentation of the state vector has the worst performance here due to the added protocol overhead while still sending the entire state vector. The overhead of all other strategies is lower than the baseline, owing to the reduction in redundant information exchanged on the network.

These results demonstrate that using partial state vectors can be a promising direction for scaling SVS to large groups. We note that this is not an exhaustive list of the possible partitioning strategies, and encourage future research to further explore this direction. Further study is also required to understand the interaction between partial state vectors and Sync Interest suppression.

3.7 Use of Sequential Naming In SVS Design

The use of sequence numbers is essential in transport protocol designs. Without sequence numbers, each dataset change must be separately and reliably propagated to every member in the group. The use of sequence numbers avoids enumerating all data names; a single number informs others how many pieces of data have been produced, and simultaneously recovers from losses of previous data production notifications.

In the SVS design, if one or more Sync Interests are dropped due to packet losses, some members may have outdated dataset state. In this case, receiving a single Sync Interest with the latest dataset state is sufficient to update the state at these members to the latest known state in the group, without the need for any further exchanges. Thus, a single packet can

correct the loss of multiple packets, increasing Sync efficiency.

Sequential naming is essential to enable the transport layer to be resilient and efficient. On the other hand, applications may desire to utilize and synchronize semantic application layer data names. SVS fulfills this requirement by building higher layer protocols over Sync. The SVS Pub/Sub (SVS-PS) [PMZ21] protocol provides the mapping between a producer's sequence numbers and the actual application data names, hiding the transport layer's use of sequence numbers from application developers. We briefly describe the SVS-PS design below.

3.7.1 SVS Pub/Sub

SVS-PS provides a mapping between the sequence numbers of Sync and the corresponding application data names. Such a mechanism is required since applications must know the semantic data name for the data that is available *before* initiating a request to fetch said data, depending on local requirements. In SVS-PS, this requirement is fulfilled using a name-mapping Interest-Data exchange. On receiving a new sequence number, an SVS-PS consumer sends an Interest to the producer of the data asking for the name of the data. The producer replies with the name of the corresponding encapsulated data[8]. Individual consumer applications can then determine whether they want to fetch the newly produced data.

We optimize this process to remove the extra RTT introduced for fetching in the name mapping with a best-effort approach. SVS sends an event-driven Sync Interest whenever a given participant produces new data. We can piggyback the name of the encapsulated data on this Sync Interest, in a best-effort attempt to inform all recipients about the application layer name of the newly produced object. Any consumers that receive this Sync Interest need not separately fetch the name mapping, saving a round trip to the producer.

[8]This process is efficient since the Data reply containing the encapsulated name is multicast to all consumers in the Sync group.

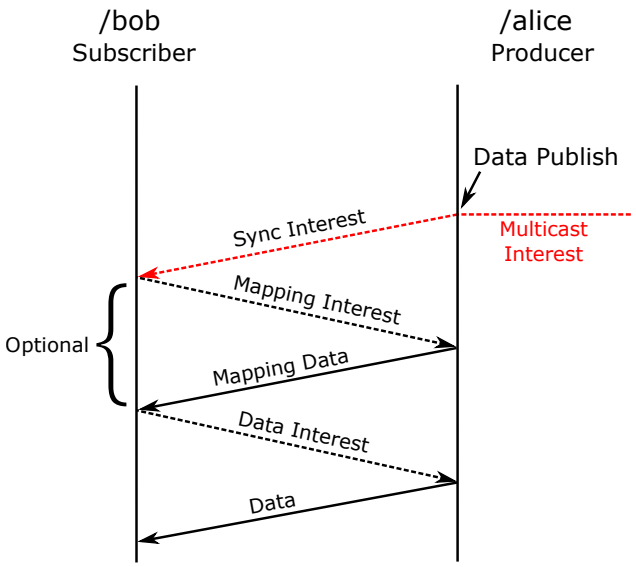

Figure 3.14: Name mapping in SVS-PS

The SVS-PS protocol exchanges are illustrated in Fig. 3.14. When a Sync participant Bob receives a Sync Interest, Bob fetches the name mapping for any newly received sequence numbers, and fetches any desired data after consulting with the application for these names. If this is the original Sync Interest generated by the producer Alice, the step of fetching the mapping can be optimized away.

Besides using semantic names, applications also desire high-level APIs to work with data. SVS-PS implementations, as the name suggests, provide such a high-level publish-subscribe API that applications can use to directly publish named data and subscribe to name prefixes for a given Sync group. In these implementations, transport and network layer details, such as sequential naming and encapsulation, segmentation, and security aspects, including data signing and verification, are automatically handled by the SVS-PS library implementation [Nam25b, Nam25d].

3.8 Security and Group Membership

Different from other Sync protocols, an SVS Sync Interest directly updates the dataset state of its recipients with the state vector. This requires that all state vectors be signed before being accepted. The application parameters field of an SVS Sync Interest carries a signed NDN Data object, which is secured similarly to all other data in a given application, using the application instance's identity and security policies. Similarly, the state vector in a received Sync Interest is verified like any other piece of data in the particular application. Notably, SVS decouples the name of the Sync Interest and the name of the state change itself, i.e. the name of the Data in the application parameters of the Interest. This design provides greater flexibility for applications to decide how to secure the state changes.

Group membership is another aspect that the application controls. Deciding whether to accept the Sync Interest from a specific entity requires information that is only available at the application layer. Higher-level libraries, such as SVS-PS implementations, may provide support for automating certain aspects of using standing NDN security primitives, such as trust schemas with SVS. We encourage such integrations to improve the developer experience.

3.9 Future Work

SVS continues to evolve, and we point out several directions for future research.

1. The p-SVS design (§3.6) is still in early stages and needs further evaluation, along with a comprehensive analysis of the interaction between partial state vectors and Sync Interest suppression. We continue to perform experiments to further evaluate this direction.

2. The timer values used in SVS have a significant impact on performance, as described in §3.4.1. Ideal timer values may depend on network conditions, the topology and the application, and may change dynamically during the lifetime of an application instance.

We envision that it may be possible in the future to learn these timer values by making passive or active measurements from the network. In the same direction, it may be possible to implement *localized* suppression – participants in physical proximity to the sender of an outdated Sync Interest should be more likely to respond with the correct state.

3. Network multicast routing support for Sync Interest multicast. While this is a network layer function, the functionality of SVS directly depends on Sync Interest multicast. Therefore, developing effective network multicast routing support should be considered imperative in supporting SVS functionality.

CHAPTER 4

NDN Distance Vector Routing

Data consumers in NDN networks request named Data by sending Interest packets that carry the names of the desired Data. The network forwards these Interests towards the Data producers, and forwards the Data back to the interested consumers. Unlike IP packets, Interest packets do not carry a destination address that they must be forwarded to; instead, they are forwarded according to the names they carry, which indicate what Data the consumers desire to fetch.

Packet forwarders in an NDN network forward Interests using a Forwarding Information Base (FIB) and a forwarding Strategy. The FIB is a table containing reachability information, indicating one or more outgoing interfaces of the forwarder that can be used to forward a given name prefix, along with additional information such as a cost value. When a forwarder receives an Interest packet, it uses the name carried in the Interest tp perform a longest prefix match on the FIB[1]. If multiple matching FIB entries are found, they are then passed to the forwarding strategy, which makes a decision on which interface(s) the Interest packet should be forwarded to. Forwarding strategies may store and utilize additional information in the forwarder to assist this decision.

The role of the routing plane in NDN is to fill in the FIB at each forwarder to enable name-based forwarding in the network. Application instances in an NDN network announce the name prefixes that they use to produce Data. An NDN routing protocol is then responsible

[1]An NDN name is made of one or more *components*. Network forwarding treats each component as a binary blob and is matched as a single unit when doing a prefix match.

for efficiently and scalably propagating application name prefixes to all forwarders in the network and computing the FIB at each individual forwarder in a distributed manner. NDN routing protocols occupy a position in the network stack similar to BGP or OSPF in TCP/IP, but distribute Data name prefix reachability instead of IP address prefix reachability.

While several routing protocols have been proposed for NDN in the past [KAH22], there has not yet been an extensive investigation into the relationship between the NDN routing and forwarding planes. Several of the existing routing designs closely follow the footsteps of TCP/IP routing protocols, without fully exploiting NDN's unique capabilities, mainly the stateful forwarding plane. In this chapter, we present the design of the NDN Distance Vector Routing Protocol (ndn-dv), and our study of the effects and implications of a stateful forwarding plane for routing protocols. Our evaluations demonstrate ndn-dv's ability to utilize a very simple design and still be highly effective, owing primarily to NDN's stateful forwarding.

4.1 Background

In this section, we provide a brief introduction to Distance Vector routing along with some general notes on lessons learned from past routing designs.

4.1.1 Distance Vector Routing

Distance Vector (DV) routing protocols represent a foundational class of distributed algorithms used to determine the best paths within a network. The core principle of DV routing relies on each router maintaining a *distance vector*, essentially a list containing its current estimate of the shortest path cost to known destinations in the network. This information is computed and maintained locally, based primarily on information received directly from neighboring routers. Routers periodically, or upon topological changes, exchange their entire distance vectors with their immediate neighbors. Upon receiving a vector from a

neighbor, a router updates its own vector by applying a distributed version of the Bellman-Ford [Bel58, For56] algorithm, comparing the cost to reach a destination through that specific neighbor with its currently held best cost. This iterative process of local computation and neighborly exchange allows routing information to propagate throughout the network, enabling routers to converge on the shortest loop-free paths based on the cost metric.

In contrast to link-state protocols such as OSPF [Moy98], which endeavor to build and maintain a complete topological map of the network domain within each router, Distance Vector protocols operate on a more localized principle, relying solely on information exchanged with direct neighbors. This fundamental difference has several distinct advantages, primarily in terms of simplicity and reduced resource utilization. DV protocols are less computationally intensive and have a smaller memory footprint, as they avoid the need to store a comprehensive link-state database and execute complex algorithms across the entire topology. Since DV protocols do not need a view of the entire network, updates in one corner of the network need not be flooded to the entire network. Instead, topological changes are propagated only as far as relevant, thereby significantly reducing protocol overhead.

Routing Information Protocol (RIP) [Mal98] was the first DV protocol to be standardized and widely deployed in operational networks. Experience with RIP deployments highlighted several issues with distance vector routing, some of which were subsequently mitigated. Two of these issues are briefly described below.

1. DV routers announce distances to reachable destinations to neighbors. Consider a network three routers $A - B - C$ connected linearly in that order. B announces a distance of 1 to reach C; this is learned by A, which subsequently announces a distance of 2 to reach C. In this situation, if the link $B - C$ fails, B will detect the local link failure and attempt to find an alternative path to reach C. Since A will continue to announce a distance of 2 to reach C, B will now *incorrectly* infer that it can reach C through A with a distance of 3. To mitigate this error in route computation, RIP uses the split horizon and poison reverse mechanisms. With the poison reverse design, A

informs B that it cannot reach C since B is along the path A uses to reach C. As a result, when the link $B - C$ breaks, B can quickly infer that no valid path now exists to reach C.

2. Even with poison reverse, a group of routers can engage in mutual deception under certain circumstances, when the network contains topological loops. In this case, the distance to a given destination continues to rise continuously, similar to the previous example without poison reverse. To break this *count to infinity*, RIP sets the maximum distance to 16 hops, and any destination with a larger distance is assumed to be unreachable.

In the second case above, a transient routing loop may exist in the network while the routers count to infinity. While such loops may be short-lived, they directly translate to packet looping in the data plane, which can significantly degrade network availability and waste resources. While several solutions have been proposed to fix the "count to infinity" problem for RIP [SS02, MJ16], concerns about packet looping remain as one of the key reasons for the lack of widespread deployment of DV routing protocols.

In later sections of this chapter, we demonstrate how NDN networks with a stateful forwarding plane can automatically break data plane loops, thus accounting for some degree of deficiencies in the routing plane. This ability enables us to develop an effective DV routing protocol for NDN, offering the aforementioned advantages without concerns about looping.

4.1.2 On Prefix Reachability

Past experience from routing protocol design and deployment has shown that topological connectivity among routers and destination prefix reachability are two separable concerns. The number of destination prefixes in a network can be orders of magnitude larger than the number of routers in the network topology, and the frequency of router topological connectivity changes may be much higher than the frequency of prefixes changing their

attached routers. Taking a look at deployed routing protocols, we make the following three observations.

1. As a link-state routing protocol, OSPF defines two different types of Link State Advertisements (LSAs) to represent topological connectivity and destination prefix information. *Router LSAs* carry information about the topology, such as which routers are directly connected to which other routers. Separate *Network LSAs* carry information about the prefixes attached to individual routers.

2. Distance Vector and Path Vector protocols such as RIP and BGP do not directly concern themselves with router topological connectivity. Instead, they only propagate prefix reachability information. Consequently, past BGP measurements have shown that a single link failure between two eBGP routers can result in hundreds or even thousands of prefixes changing routes [WZP02].

3. In large operational networks, intra-domain IP packet forwarding often takes a 2-step approach. An Interior Gateway Protocol (IGP) is used to establish router reachability within an autonomous system, and iBGP is used to establish reachability to destination prefixes. For each IP packet arriving at an ingress router R_i of an Autonomous System (AS), R_i first looks up the prefix table to find the AS exit router R_e. Then the router *separately* looks up the IGP table to find the next hop along the best path to reach R_e and forwards the packet to this next hop. It should be noted here that two separate protocols are used: one to build the lookup tables for prefix to router mapping, and one to learn the router to best route mapping.

Scalability considerations suggest that topological connectivity among routers and destination prefix reachability are best handled separately. As described in §4.2, the ndn-dv design adopts a two-step approach to minimize overhead and improve scalability. ndn-dv is designed to establish and maintain router reachability among all routers in a network, and a separate prefix-to-router mapping table is used to match NDN name prefixes to routers.

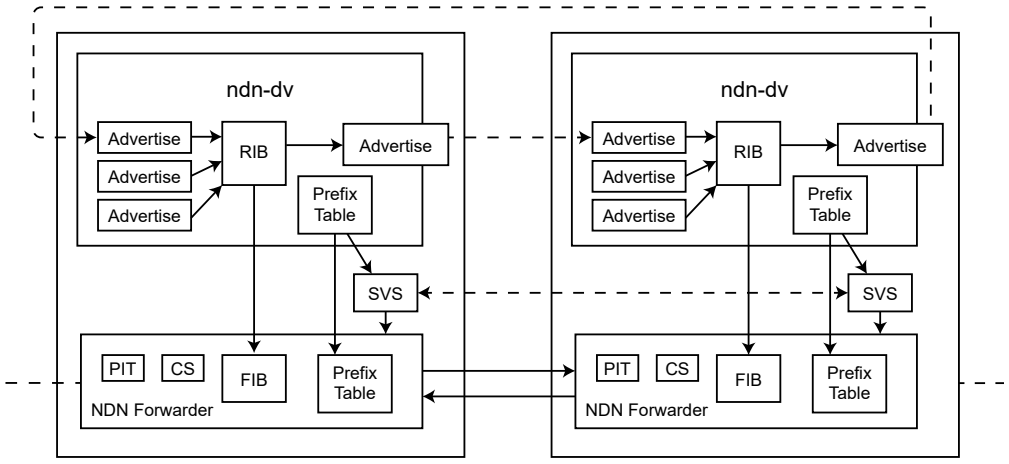

Figure 4.1: Design overview of ndn-dv

4.2 Design of ndn-dv

In this section, we present the design of the NDN Distance Vector (ndn-dv) routing protocol. ndn-dv is built with five basic components; we briefly describe each of these below and expand further in a dedicated subsection.

1. **Routing Information Base (RIB):** Routers running a distance vector routing protocol must maintain a table with the distance to every other router in the network. The RIB of ndn-dv serves the equivalent function with some additional information, as described in §4.2.1

2. **Advertisements:** DV routers exchange their distance vector with the neighbors of a router. Advertisements, as described in §4.2.2 are the equivalent messages exchanged by ndn-dv between neighboring routers periodically and on RIB updates.

3. **Update Processing:** Advertisements received from neighboring routers are processed whenever they are received to update the RIB. The algorithm used to make these updates is described in §4.2.3.

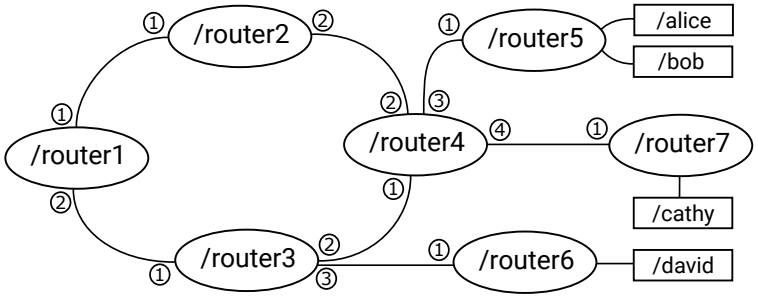

Figure 4.2: Example topology for ndn-dv illustrations

4. **Prefix Table:** ndn-dv separates router and prefix reachability using a dedicated prefix mapping table. The network can ideally utilize a two-step forwarding plane; however, extra steps may be required to maintain backward compatibility with existing NDN forwarders. We describe prefix table synchronization and the FIB computation steps in §4.2.4.

5. **Security:** All updates generated by ndn-dv are signed by the source router and authenticated upon receipt. We provide a description of the routing security model in §4.2.5.

Fig. 4.1 illustrates an overview of the various components of ndn-dv. Each component is described in detail in the following sections. For all the illustrations and examples that follow, we use the simple example topology shown in Fig. 4.2. In the referenced figure, routers are represented by ellipses and data producers are represented by boxes. Numbers in circles identify interfaces at individual routers

4.2.1 Routing Information Base (RIB)

The RIB in an ndn-dv router is a table of all reachable destination *routers* in the network, with the cost (distance) to reach the destination through *each* outgoing interface of the router. Each destination router in an NDN network is identified with a preconfigured name, and this reachability information is stored as a map of the destination name to a list of

Destination	Intf (1)	Intf (2)	Intf (3)
/router1	1	3	∞
/router2	2	2	∞
/router4	3	1	∞
/router5	4	2	∞
/router6	∞	∞	1
/router7	4	2	∞

Table 4.1: RIB at router 3 in the example

candidate interfaces and their corresponding costs to that destination. Table 4.1 illustrates the contents of the RIB at one of the routers in the example topology[2]. Two distinct features of this RIB design are notable.

1. Traditional DV protocols such as RIP compute a single best-cost path to each destination in the network, thus identifying a single distance value and the next hop for this path for a given destination. Each entry in the ndn-dv RIB provides *multiple* next hop candidates and their associated costs to reach the destination through the given next hop. This extra information can be utilized by NDN's stateful forwarding plane for multi-path forwarding and failure recovery, as described in later sections.

2. The RIB only stores reachability to *routers*, not reachability to data name prefixes announced by applications attached to the network. Since the number of interfaces at a router can be safely assumed to be a relatively small constant, the size of the RIB table is linearly proportional to the number of routers in the network. External actions such as a new data producer joining the network and announcing prefixes have no effect on the contents of the RIB.

[2]The referenced table serves the purpose of illustration only, and does not indicate the actual computation results of the ndn-dv protocol.

ndn-dv routers learn the contents of the RIB by exchanging information in the RIB with their direct neighbors. We describe the mechanism of these exchanges next.

4.2.2 Routing Advertisements

Traditional routing protocols generally have a *push*-based design, wherein each router *tells* its neighbors (e.g. RIP or BGP) or all other routers in the network (e.g. OSPF) what destinations the router is able to reach. ndn-dv reverses this design to a *pull*-based routing update dissemination paradigm. An ndn-dv router merely advertises what destinations it is able to reach, and interested neighbors then *fetch* this *Advertisement* in their quest to find optimal paths in the network. This design has an important implication: routing updates produced by an ndn-dv router are not tailored separately for each neighbor. However, while all neighbors fetch identical updates, their interpretations of these updates may differ.

Each ndn-dv router generates an *Advertisement* whenever its local RIB changes. The Advertisement is the only type of message used for routing, and only a single Advertisement is valid for a given router at any given point in time. Each Advertisement contains the following information.

1. A list of destination routers in the advertising router's RIB. The advertiser itself is added as a special entry.

2. For each destination, the Advertisement includes the next hop router name along the lowest-cost path to that destination, and the associated cost value. A cost of zero is used to indicate the advertising router itself. Any ties in selecting the lowest-cost path are broken using a consistent hash of the router name.

3. If the RIB contains multiple possible candidates for the next hop, a third entry of *other cost* is used to indicate the cost if the *second-best* path is used reach the destination.

For our continuing example, Table 4.2 illustrates the contents of the advertisement gen-

Destination	Next Hop	Cost	Other
/router1	/router1	1	3
/router2	/router1	2	2
/router3	/router3	0	∞
/router4	/router4	1	3
/router5	/router4	2	4
/router6	/router6	1	∞
/router7	/router4	2	4

Table 4.2: Advertisement generated by router 3 in the example

erated from the contents of the example RIB in Table 4.1.

ndn-dv routers use the State Vector Sync protocol in a simple local-broadcast manner to notify neighbors about the existence of new or updated Advertisements. Each router broadcasts a Sync Interest with a well-known prefix that all routers in the network subscribe to. This Sync Interest contains the broadcaster's name along with a sequence number identifying the latest version of the Advertisement. When an interested neighboring router receives this broadcast Sync Interest, it may fetch the corresponding Advertisement using this name and sequence number, and update its local RIB to create new routing paths towards the advertising router.

Similar to SVS, the broadcast Sync Interests are generated whenever the Advertisement has a new version (event-driven) and periodically (time-driven). Since these messages are sent periodically, they are also used for failure detection. If no broadcast is received from a neighboring router for a pre-configured number of time periods, the link to the neighbor is assumed to be dead and the neighbor's Advertisement is dropped from future RIB computations. If a broadcast Sync Interest is subsequently received, the latest Advertisement is fetched again, and any routes through the neighbor are restored.

As discussed earlier, unlike RIP, each ndn-dv router produces an identical Advertisement

for all neighbors. As a result, the process of updating the RIB from this information also differs significantly. The algorithm for processing these updates is described in the next section.

4.2.3 Update Processing

Each ndn-dv router fetches advertisements from neighboring routers to recompute its local RIB; if the local RIB changes, it produces a new routing advertisement. We describe the update processing algorithm below. A pseudocode implementation of the algorithm is illustrated in Algorithm 1.

1. The cost of reaching each destination through a given interface is set to 1 more than the cost advertised by the neighbor connected to that interface. This design may easily be expanded to accommodate per-link costs, by adding the cost of the link to the advertised cost instead.

2. If the cost to a destination is not advertised or is infinity, the cost in the RIB to the destination through the corresponding interface is set to infinity. Any advertised costs that are greater than a preconfigured maximum hop count or cost are assumed to be infinite.

3. If the next-hop router for a destination is the router processing the update, then the cost from the *other* field is used instead, if present. If no *other* cost is specified, the cost is set to infinity. This condition serves a function similar to poison reverse in RIP, while finding multiple possible paths to a given destination.

This algorithm is used to recompute the RIB whenever any neighbor's Advertisement changes. This may happen either after an updated Advertisement is fetched following a newly received sequence number, or on detecting a local link failure that invalidates the neighbor's Advertisement.

Algorithm 1 Updating RIB from Advertisements

```
function ComputeRIB(neighbors)
    rib                                                    ▷ return value
    for each n in neighbors do
        if n.advert is not null then
            for each entry in n.advert do
                cost ← entry.cost + 1
                if entry.nexthop is self then
                    if entry.other is not null then
                        cost ← entry.other + 1
                    else
                        continue
                    end if
                end if
                if cost ≥ max then
                    continue
                end if
                rib[entry.dest][n.intf] ← cost
            end for
        end if
    end for
end function
```

An important feature of this update processing algorithm is the ability to keep multiple paths to the same destination with stable cost values, without exchanging the entire RIB with neighbors. The presence of an *other* cost in the Advertisement indicates the existence of a secondary path, even when the processing router is along the lowest-cost path from the advertising router to a destination. We note that while the algorithm ensures that the lowest-cost path is always valid, secondary paths with higher costs might have loops with certain topologies. However, these loops are benign and do not affect forwarding.

ndn-dv routers compute the RIB using Advertisements fetched from neighbors, establishing reachability to all routers in the network. In the data plane, however, NDN forwarders must forward Interest packets using *application name prefixes* rather than router names. A dedicated prefix mapping table performs this translation, as described in the next section.

4.2.4 Prefix Table

One lesson learned from BGP routing is that using a routing protocol to propagate prefix reachability can lead to scalability issues. The same is true in an NDN network, and can be much worse because an NDN network may need to establish reachability to a much larger number of name prefixes. To scale routing update overhead linearly with the number of routers in the network instead of the number of prefixes, ndn-dv establishes reachability to routers only. A separate prefix table is then used to map name prefixes to their attached routers in the network to support packet forwarding using Data names. We note that the design of the prefix table presented below is not specific to ndn-dv, and can be generally utilized by other NDN routing protocols.

The prefix table is a globally synchronized table that maps external application data prefixes to their attachment points in the network. This mapping table is maintained by all routers and is synchronized using an NDN Sync group. An illustration of the contents of the global prefix table in our example topology is shown in Table 4.3.

Name Prefix	Exit Router
/alice	/router5
/bob	/router5
/cathy	/router7
/david	/router6

Table 4.3: Prefix table in the example

In a network running an ndn-dv network, all routers join a common prefix table Sync group. In our implementation, we use the State Vector Sync Pub/Sub (SVS-PS) [PMZ21] framework to distribute any updates to the prefix table. SVS has been shown to be resilient to dynamic network conditions such as link failures and packet losses, which enables ndn-dv routers to quickly converge on a consistent prefix table. Since entries in the prefix table corresponding to an exit router are governed solely by that router, the prefix table is conflict-free. As a result, all prefix table operations can be performed with the three primitives described below.

1. A *Reset* operation indicates that all prefix entries for the router should be cleared from the prefix table. Routers produce this operation during initialization after a fresh start, to prevent any conflicts with previous instances of the same router name.

2. An *Add* operation indicates a new prefix has been attached to the router, and must be added to the prefix table. This operation may carry a cost, indicating the cost to reach the prefix from the exit router. Subsequent Add operations may be used by routers to update this cost in the prefix table.

3. A *Remove* operation indicates that a prefix has been detached from the router, and must be removed from the prefix table.

In the SVS group used to synchronize the prefix table, each data object is named sequentially, indicating one of the operations indicated above. These operations are fetched and

applied at all routers in order, leading to a consistent dataset state. Since the Data carrying these updates is multicast to all other routers in the network, this process is very efficient and has low overhead.

A network may suffer from prolonged network partitions, and separately, routers might join a network that has already been operational for a long time. In both cases, a long queue of updates to the prefix table may have accumulated that routers now need to fetch in-order, greatly increasing the convergence time of the prefix table. To mitigate this possibility and to make bootstrapping efficient, SVS-PS provides a snapshot mechanism to prevent fetching a large number of historical updates. Every router produces a snapshot of its entire local prefix table state using the same primitives as above, i.e. one *Reset* operation followed by one *Add* operation for each entry in the prefix table for that router. Such a snapshot is produced periodically, every time a preconfigured number of updates are generated. After a network partition resolves or a new router joins the network, it may choose to fetch this snapshot first, instead of fetching a large number of individual updates, thereby speeding up convergence of the prefix table.

The prefix table indicates which specific router(s) can reach a given name prefix (a multihomed prefix can be reached through multiple routers). As a result, the table size scales linearly with the number of prefixes announced to the network. The contents of the prefix table are unaffected by link failures or other topological changes, and the table is updated only when an external name prefix announcement is received by or withdrawn from an ndn-dv router. This property is orthogonal to updates to the RIB, which are affected only by topological changes and not by applications changing their attachment points to the network. Thus, this design creates a clear separation of concerns between these two problems. Since all the changes to the prefix table can be propagated efficiently in the network via NDN's native multicast delivery, this design greatly improves the routing system scalability.

Ideally, NDN forwarders should perform a two-step lookup when forwarding an Interest packet. When an Interest is received, forwarders must first perform a longest prefix match

Name Prefix	Next Hops
/alice	intf=2 (cost=2), intf=4 (cost=4)
/bob	intf=2 (cost=2), intf=4 (cost=4)
/cathy	intf=2 (cost=2), intf=4 (cost=4)
/david	intf=3 (cost=1)

Table 4.4: Flattened FIB at router 3 in the example

on the prefix table to find a matching exit router(s). Then the forwarder must look up the FIB to find the interface(s) that can be used to reach the exit router(s) and the cost of each. This information can then be used by the forwarding Strategy to make a forwarding decision.

The current ndn-dv implementation deviates from the above to a certain degree. The existing NDN forwarder implementation, NFD [ASZ21], uses a Forwarding Information Base (FIB) that directly maps application name prefixes to output interfaces and the corresponding costs. The FIB is used to perform a single-step longest prefix match to find all possible faces to reach the prefix. To allow ndn-dv to work with the existing forwarder implementation, the prefix table and RIB are currently joined and flattened to output a FIB, which in turn has to be updated continuously whenever either the RIB or prefix table changes. Table 4.4 illustrates the contents of the flattened FIB at one of the routers in our example topology.

4.2.5 Security

We can decompose routing security into two separate tasks. First, only authorized entities should be able to announce name prefixes to the network. Second, the internal update messages exchanged between routers must themselves be secured. The NDN security model can handle these tasks directly, providing security to the ndn-dv protocol in a way similar to any other NDN application, using a process of security bootstrapping and a schematized trust policy [YAC15].

NDN views a network as a collection of entities, each identified by its semantic name and with trust relations with others. Each router in the network is bootstrapped with an identity (i.e. a unique name) and a certificate for this identity. Being NDN Data packets, all routing updates in the network are signed using each individual router's private key. Other routers can verify these updates using the router's certificate, tracing the chain of trust back to the locally configured trust anchor. Authorization is performed using a schematized trust policy; we use the LightVerSec [YMX23b] language to specify the trust policy in the ndn-dv implementation. Thus, the routing updates are secured in the same way as any other NDN application, using the security primitives that already exist. Notably, this design does not depend on the Web PKI, allowing the usage of a trust anchor controlled by the operator of the network.

External prefix announcements are secured similarly. Each announcement is signed by the customer or autonomous system announcing the prefix, and can then be verified based on the pre-established real-world trust relations between the two entities by configuring a trust policy. NDN security primitives allow the creation of scalable name-based trust policies through schematization, which is key enabler in this regard.

4.3 Implementation and Evaluation

We initially implemented ndn-dv in Typescript using the NDNts [Jun24] library for experimentation and measurements. A production-ready version of routing daemon was subsequently implemented in Go as a part of the NDNd [Nam25d] project, including a security implementation and optimizations such as prefix table snapshots. The evaluation results shown below reflect the initial Typescript implementation, but were subsequently validated with the more mature implementation in NDNd.

To evaluate ndn-dv, we emulated a 52-node Sprint PoP network topology [SMW04] with a maximum diameter of 8 hops, using MiniNDN [Min25]. Each link was set to have a

relatively high 50ms one-way latency to study the effects of slow routing convergence. In our evaluation scenario, each router runs an NDN forwarder (NFD [ASZ21]), the ndn-dv routing daemon and a simple Data producer that produces random content. At the start of each evaluation, we randomly chose 80 pairs of routers in the network to set up data flows. For each pair, a consumer connected to the first router would continuously fetch data from the second router, sending one Interest every 10ms. Each consumer would log whether it was successful in retrieving the Data packet, serving as a test of reachability to a given name prefix. The flows in the network were determined at the start of the experiment and were fixed for each experiment.

During the experiment, we randomly failed links with a probabilistic model. We specified a Mean-Time-To-Failure (MTTF) and a Mean-Time-To-Recovery (MTTR) for each run. A shorter MTTF indicates that links fail more frequently, while a shorter MTTR indicates that links recover faster after failures. The specified MTTF values are utilized independently for *each* link; i.e. an MTTF of 100 sec indicates any given link would fail at an average of once every 100 seconds.

In our evaluation, we planned to study both the performance of ndn-dv as well as the effect of the stateful forwarding plane on the overall network performance. To study this interaction, we ran three sets of experiments with the same parameters.

1. In our baseline test, consumers transmit each Interest only once. We use NFD's default Best Route strategy, which always chooses the face with the lowest cost to forward any Interests that are not retransmissions. This ensures that only the lowest-cost path is used for Interest forwarding. In other words, the forwarding plane always directly utilizes information provided by the routing plane without attempting to find alternative paths in the data plane. As a result, this case serves a good baseline to directly measure the convergence time of the ndn-dv routing protocol.

2. In our second experiment, we continue to utilize the same Best Route strategy, but

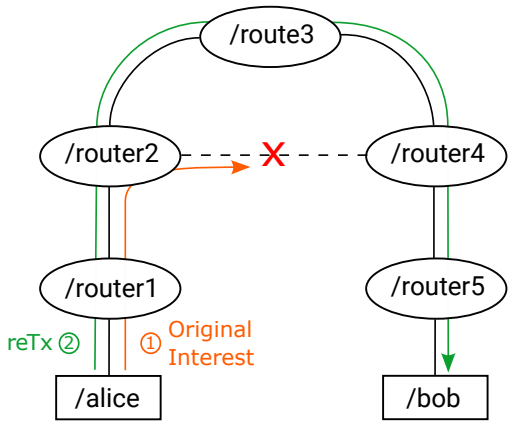

Figure 4.3: Failure recovery with Interest retransmissions

enable automatic Interest retransmission in the client library used at the consumer application. We allow up to two retransmissions, at least 500ms apart. This means that if the consumer does not receive the requested data within 500ms, the same Interest packet is retransmitted, and the process is repeated after another period. On receiving retransmitted Interests, the Best Route strategy explores alternative paths, forwarding Interests to matching interfaces with higher costs. This enables the network to discover alternative paths when the lowest-cost path has failed or leads to a transient data plane loop. This mode of failure recovery is illustrated in Fig. 4.3.

3. For our third experiment, we implemented an alternative forwarding strategy in NFD called the Best-Two-Routes (B2R) strategy. On receiving an Interest packet, this strategy first forwards the Interest to the matching lowest-cost face. If this Interest fails to retrieve the data in time or triggers a NACK, the *forwarder* retransmits the Interest using the next-best face. Thus, loss recovery is *network-driven* when using such a strategy, unlike the *consumer-driven* loss recovery described above. Further, the forwarder keeps track of which links have not received any packets for some time and marks these links as potentially broken. A smaller timeout value can then be used if forwarding Interests to these links, since they are less likely to work. If any packets are subsequently received from the broken link, it is marked as usable again. Fig. 4.4

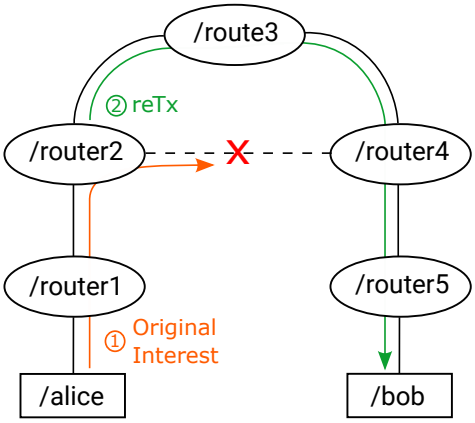

Figure 4.4: Failure recovery with Best-Two-Routes strategy

illustrates this mode of loss recovery using the B2R strategy.

We ran each emulation scenario for a duration of 300 seconds, and measured the fraction of unsatistied Interests as experienced by the consumer application, indicating the fraction of time for which the data flows set up in the network were disrupted. For the link failures, we used a fixed value of 120 seconds for the MTTR, and varied the MTTF from 4000 to 300 seconds. Each experiment was run thrice with a different random seed value to create variations in the set of data flows and the sequence of link failures over the duration of the experiment.

During our evaluation, we noticed that some failed links may create network partitions, which would make Interests for some flows impossible to satisfy for the duration of the partition, even in the presence of a perfect routing protocol. To account for these partitions, we analytically calculated the number of Interests that are impossible to satisfy for each run of the experiment. This number shown in the graph in Fig. 4.5 was subtracted from the loss results of the actual experiments as a zero correction. In rare cases, this may cause an over-correction, since Interests sent during the network partition may be satisfied by any retransmissions after the partition recovers. This case shows up in the results of a few evaluation runs as a small negative loss value observed after this correction.

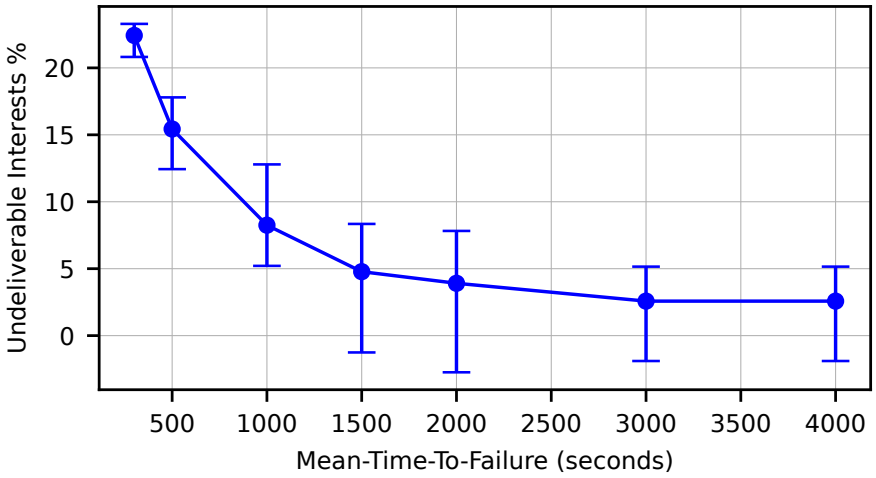

Figure 4.5: Unsatisfiable Interests due to network partitions

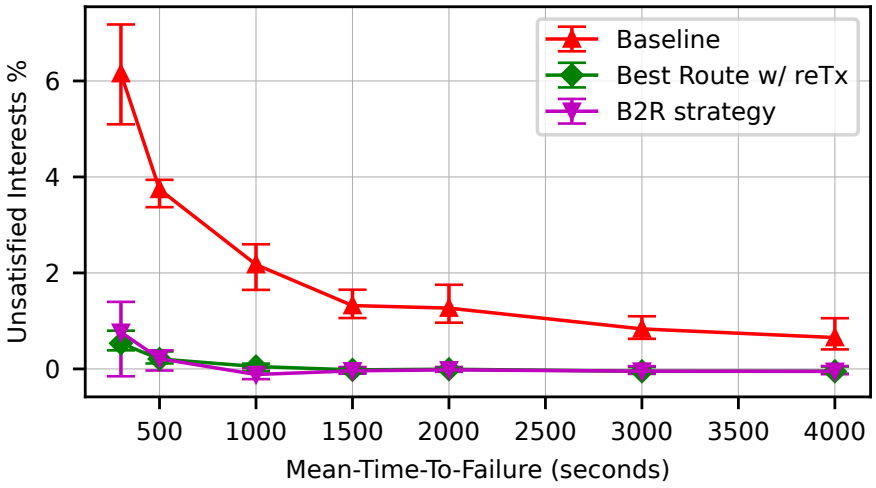

Figure 4.6: Fraction of unsatisfied Interests with ndn-dv

The comparative results for the three sets of runs from our experiments are presented in Fig. 4.6. As link failures become more frequent (i.e. for shorter MTTF values), the lowest-cost path becomes increasingly likely to fail the data flows in the network. In our baseline evaluation, these flows are restricted to using a single path only, similar to IP forwarding. Consequently, an increasing number of Interests are lost, as our results demonstrate. However, losses in the network remain low at approximately 6% at an MTTF of 300s, which indicates a very high rate of link failures. This indicates that ndn-dv routing can quickly converge to loop-free paths in the network upon detecting failures.

When consumer-driven Interest retransmission is enabled, or when the B2R strategy is used, forwarders can use alternate next hops for retransmitted Interests. As a result, Interests may follow a different path if the lowest-cost path is broken and the initial Interest is lost or encounters a loop. This path discovery is reflected in the nearly-zero fraction of unsatisfied Interests observed for both retransmission strategies. We note that with a relatively longer MTTF, the forwarding plane is able to satisfy *all* Interests despite link failures, indicating availability is completely unaffected by slightly inconsistent routing state. As the MTTF decreases, multiple paths may be failed simultaneously, and the subsequent retransmissions of Interests are more likely to also encounter failed paths. This translates to the observed increase in the number of unsatisfied Interests; however it remains an order of magnitude smaller compared to the baseline. We observed that allowing more retransmissions was effective in satisfying more Interests, since it allows the data plane to effectively explore more alternative paths to reach the same destination.

We also compared the convergence performance of ndn-dv with NLSR [WLH18], a link-state routing protocol for NDN. We used the baseline strategy only for this comparison, since it directly reflects routing plane performance without help from the stateful forwarding plane. NLSR utilizes NDN Sync to distribute LSAs containing prefix announcements to all routers in the network, and each router computes the entire network topology to find the best path to reach destination prefixes. Both ndn-dv and NLSR were configured similarly

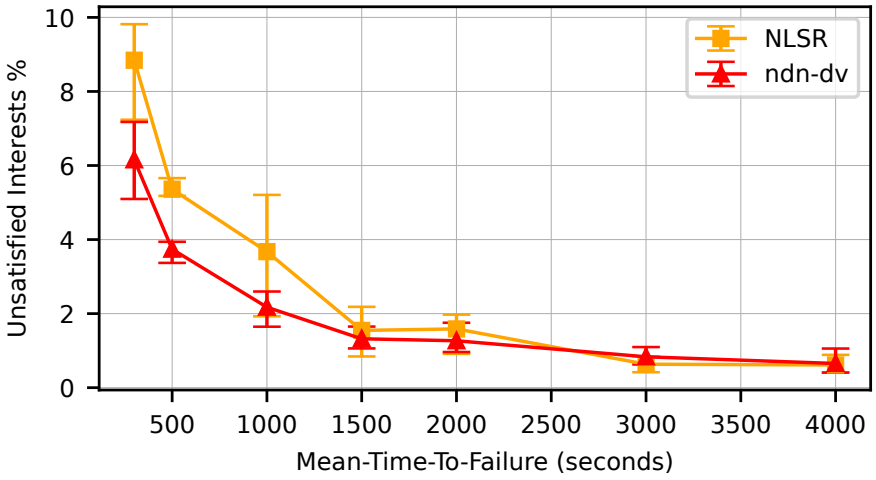

Figure 4.7: Baseline comparison of ndn-dv with link-state

for parameters such as link failure detection timeouts.

Our evaluation found that ndn-dv performance is comparable to that of NLSR in terms of routing convergence, as shown in Fig. 4.7. These results may be considered surprising, since NLSR floods topological updates to the entire network, theoretically enabling much faster convergence at the cost of higher routing overhead. We note, however, that these results serve only as a rough comparison, as the NLSR implementation performs various optimizations to reduce protocol overhead, which may have affected its ability to converge more quickly.

We draw two inferences from the results of our experimental evaluations. First, the ndn-dv design can effectively and speedily converge to find *multiple* paths to reach name prefixes in an NDN network. In the presence of link failures, the protocol is able to adapt quickly and find alternative paths. Second, the intelligent forwarding plane of NDN is capable of automatically working with inconsistent routing state in a realistic topology. NDN forwarders can successfully find alternative working paths, using additional multi-path information obtained from the routing protocol. Since the performance perceived by end users is shielded from inconsistencies in routing inside the network, we can significantly

relax delay and consistency requirements on the routing protocol. This, in turn, allows the effective use of a simple distance vector protocol like ndn-dv, without compromising the availability of data in the network.

4.4 Discussion

On experimentation with link-state routing over NDN, Yi et al. [YAA14] concluded that "because the forwarding plane is capable of detecting and recovering from failures quickly, routing no longer needs to handle short-term churns in the network. Freeing routing protocols from short-term churns can greatly improve their scalability and stability".

Our evaluation of ndn-dv also shows how NDN's stateful forwarding plane can help relax routing requirements, but in a different context: NDN can effectively utilize a simpler distance-vector routing protocol. Usage of DV protocols remains relatively rare in larger TCP/IP networks due to concerns about looping. The NDN forwarding plane breaks packet loops, improving data availability and preventing inefficiencies.

In the rest of this section, we provide further discussions on the insights we gained from our experimentation with ndn-dv.

4.4.1 Multipath Forwarding

Our evaluation results demonstrate that NDN's forwarding plane can utilize alternative paths during the period when routing has not yet converged. On close inspection, we note two key enablers for multipath forwarding that NDN utilizes.

1. **Per-packet state:** In §4.3, we noted how NFD's best route strategy explores alternative paths for retransmitted Interests. This is possible since NFD maintains state for each outgoing packet in the Pending Interest Table (PIT). Since each Interest packet carries a unique nonce, the forwarder can detect that a newly received Interest is a

retransmission, and thus decide to forward it along a different path.

2. **Symmetric packet flows:** In NDN, Data packets follow the same path that the Interest took to the producer in reverse order, using a breadcrumb trail that the Interest leaves at intermediate routers. Symmetric flows create a feedback loop which enables NDN forwarders to categorize Interests as satisfied, timed out, or explicitly rejected (NACK'ed) by the next hop. This allows forwarders to infer producer reachability through the tried interface directly; for example, if all Interests for a given prefix time out when sent along the lowest-cost path, the forwarder can select a different path to forward future Interests of the same prefix. IP forwarders, on the other hand, cannot make such measurements since IP packets are one-way flow only; although TCP connections have 2-way packet exchanges, they may not take symmetric paths and they should not be visible to network layer, so the forwarder may never know for sure whether the next hop that it has forwarded packets to still works or has failed.

While our evaluation demonstrated NDN's multipath forwarding capabilities only in the presence of failures, the same model can be extended to improve performance in various situations. For example, the forwarding strategy may be extended to provide in-network congestion control [SZ22], and be allowed to use multiple paths when upstream links are congested.

4.4.2 Breaking Loops with Stateful Forwarding

ndn-dv does not guarantee that paths in the FIB are loop free; temporary loops may be formed in the forwarding plane while routing converges after link failures. NDN's stateful forwarding enables it to detect and immediately break these loops, using the mechanisms described below.

1. Every Interest in NDN contains a randomly generated nonce that uniquely identifies the Interest packet. When an Interest is stored in the PIT, the nonce is also stored as

part of the PIT state. If an Interest loops back to an NDN forwarder, it can detect that the same packet has been received previously, indicating a looped Interest. The forwarder reacts by dropping the incoming Interest and sending a NACK to indicate a loop to the downstream forwarder.

2. Under certain circumstances, Interests may loop back to a forwarder after the PIT entry has already been dropped. In this case, a separate Dead Nonce List (DNL) is used by the forwarder to track the recently received name-nonce pairs, and this list is used to detect the loop [MPA23].

This ability of NDN's forwarding plane to break loops helps prevent packet looping during routing protocols' convergence time, removing the requirement for loop-free protocol or fast routing convergence. Even if higher-cost paths with loops exist in the FIB, these are probed on failure of the primary path and quickly discarded upon loop detection. As a result, these loops have little impact on packet forwarding performance.

CHAPTER 5

Ownly Workspace

The centralization of control power on the Internet has been steadily gaining attention in recent years. Today's applications rely heavily on cloud services for functions such as communication, storage, and security; in particular, they simply cannot function without a (often implicitly) trusted third party, i.e. the cloud service provider. Economics of scale dictate that this third party is inevitably centralized to a few dominant players; the damaging effects of such centralization have been widely documented [Not23, Ma24].

The recognition of Internet centralization has led to efforts towards building a new class of *decentralized* applications. Decentralized applications and protocols aim to reduce or eliminate central control points such as cloud service providers. This process can be broken down into two distinct goals. First, users should retain full ownership and control over their data, including the ability to determine who can access it and for what purpose. Second, users should not rely on centrally controlled infrastructure for secure communication and storage, as such infrastructure itself is a point of centralized control.

Several decentralized applications have been developed in recent years. The most prominent of these include federated social platforms such as BlueSky [KFG24], Mastodon [Mas25] and Nostr [fia24]. These systems aim to enable users to interact within a distributed system where no single entity has overarching control. More general-purpose platforms such as Solid [SMH16] directly advocate for user-centric data ownership, allowing individuals to store and manage their own data in "pods" and to grant selective access to other applications. For public data, IPFS [WTP24] aims to provide a decentralized storage layer for

applications to build upon. Web3 applications aim to leverage blockchains to build financial systems and enforce digital ownership through cryptographic primitives [MKC23]. Unlike federated models, Web3 places strong emphasis on immutable ledgers, smart contracts, and token-based economies to replace intermediaries.

However, on closer scrutiny of the deployments of some of these designs, we notice that they continue to tend towards centralized control. Recent studies on IPFS [WTP24] and BlueSky [BSA24] have shown an increasing dependency on centralized third parties. Most users of Mastodon continue to be concentrated at a single instance [the25]. Distributed blockchains that rely on proof of work or stake are also vulnerable to centralization [BS15].

Ma [Ma24] identified the need for semantic naming and end-to-end security as a lesson learned from the study of the efforts towards decentralized applications, and described the design of the *NDN Workspace* application, which highlights a new path for building decentralized applications using Named Data Networking. In this chapter, we present the design of *Ownly*, the spiritual successor of NDN Workspace. The Ownly design builds on lessons learned from the NDN Workspace's development and deployment. More specifically, Ownly developed new solutions for resilient data reliability in the absence of persistent user storage without assigning users a new identity, formalizing the structure of a user workspace and identifying its relation with the NDN transport service, developing a new solution to application snapshotting, extending NDN's trust schema to support dynamic security policies which significantly expands the capabilities and scalability of NDN trust schema, and last but not least, improving the user interface to make Ownly truly usable by all without awareness of the underlying NDN substrate that provides resilient security and service. As we discuss later, Ownly's design, empowered by NDN's name-based data-centric security, eliminates all central control points.

5.1 Design Goals

As an introduction, we first describe the functionality of Ownly, and then lay out the goals and non-goals of the Ownly design explicitly.

At a high level, Ownly aims to enable real-time collaboration between users. A group of users may desire to share and jointly modify a set of documents, diagrams, or other types of files[1]. To start such a collaboration, one of the users would create a *workspace* and invite others to collaborate with them. Invited users can then securely and privately collaborate within the workspace, creating and editing files in real-time, uploading external files like images, and engaging in online discussions with other users.

We identify the high-level design goals of Ownly below, highlighting how these goals distinct Ownly from today's commercial cloud-based services. These goals also reflect our general view on the key objectives for any decentralized application.

1. In today's collaborative applications, all users connect to a single cloud server[2]. The server maintains the "true" copy of all documents and files. Users can update the documents by submitting a series of requests to the server, and learn about the edits made by their peers *through* the cloud server. The server physically possesses the documents, and all users trust the server to provide a faithful and the latest version of them. Eliminating such a central controller is the most important design goal of Ownly. Users of Ownly should be able to collaboratively develop documents with no reliance on any trusted third party.

2. In today's applications, all users can communicate securely through the cloud, which in turn means no communication is possible when connectivity to the cloud is unavailable

[1]This functionality closely resembles and is in parallel with commercial cloud-based services such as Notion, Google Docs, and Dropbox.

[2]Although multiple servers may be used in reality for fault tolerance, the key point is that all servers are centrally controlled and managed by a single party.

or during infrastructure outages. The second design goal of Ownly is to enable users in the same collaboration to communicate securely via any available physical connectivity, with no reliance on connections to a central rendezvous server.

3. Finally, Ownly considers usability as an important design goal. Users should be able to reap the benefits of decentralization without needing to deploy their own infrastructure or understand the system's internals. Furthermore, the implementation should strive to provide a broad set of functionality that end users may need in such a collaborative application.

We emphasize that our design goals for Ownly *do not* discourage the usage of cloud *infrastructure* as resources. Taking storage resources as an example, using cloud-based storage could be more cost-effective and offer higher availability due to economies of scale and resource sharing. However, unlike existing cloud-based applications, the infrastructure must not have control over user applications or the ability to access user data without permission. Furthermore, the overall system must provide users with the freedom to utilize alternative infrastructure if they so desire.

5.2 Application Layer

In describing the Ownly design, we start at the top, the application itself, then work our way down the stack to explain the functions of each layer.

5.2.1 Naming

Naming is the cornerstone of any Named Data Networking application design. Assuming a number of connected entities participate in a running Ownly application, each of these entities and the data they produce must be uniquely named using well-defined naming conventions. We describe these naming conventions below.

DNS Name	NDN Name
alice.cs.ucla.edu	/edu/ucla/cs/alice
alice.cs.ucla.edu	/cs.ucla.edu/alice
alice@cs.ucla.edu	/cs.ucla.edu/@alice

Table 5.1: User name conversion

1. **Users:** Each user in an application must have a *unique identifier*. This identifier should ideally be semantically meaningful, allowing other users to correlate it with real-world relationships easily. To obtain meaningful and unique identifiers, we use DNS names to assign identifiers to users and other entities within the Ownly design. Ownership of a DNS namespace can be easily proven using existing tools, enabling users to bring their existing identifiers into the Ownly namespace. It should be noted that utilizing DNS namespace ownership for assigning identifiers does not introduce a dependency on the DNS *infrastructure*; this design only uses the DNS namespace to ensure name uniqueness.

   In practice, not all users may own a DNS namespace. We treat email addresses, which should also be globally unique, as an extension of the DNS namespace, thereby allowing them to be used as user identifiers. To bring these identifiers into the Ownly namespace, they must first be converted to the NDN naming format. Table 5.1 illustrates some ways to perform this conversion. We note that multiple name conversions are possible[3] for each user identifier; the exact method is left up to the implementation, with the requirement of consistency.

2. **Workspaces:** Each workspace that a user creates or joins must itself have a unique name. In the Ownly design, workspaces are named under the namespace of the "owner", i.e. the user that created the workspace. Thus, the name of a workspace

[3]Having more name components in the converted NDN name provides extra semantic information across all layers, but comes with a cost of increasing the complexity in the forwarding and data verification.

is the owner's NDN name, followed by one or more name components uniquely identifying the particular workspace instance. For example, the user "/cs.ucla.edu/@alice" may create a workspace named "/cs.ucla.edu/@alice/lab".

In our data model for Ownly, the owner of a workspace controls the authenticity and authorization rules for all data generated within that particular workspace. This relationship is defined by naming the workspace under the owner's namespace. Participants in a workspace are also referred to in a similar manner. When a user "/arizona.edu/@bob" joins Alice's workspace, Bob is assigned an application-scoped identifier "/cs.ucla.edu/@alice/lab/arizona.edu/@bob" which is used in this particular workspace[4]. This naming convention is crucial for defining data ownership, as described next.

3. **Workspace Data:** Users in a workspace produce data by uploading or modifying files in a workspace. This data is named in a semantically meaningful way to enable systematic trust policy definitions. In Ownly, all data produced by a user is named under that user's *application-scoped* identifier as described above. We also make two application design decisions here. First, a single workspace can have multiple *projects*, representing mutually independent sets of files. Second, data under each project is identified sequentially[5] to facilitate the use of Sync directly. Thus, in our continuing example, a piece of data produced by Bob in Alice's workspace can be named as "/cs.ucla.edu/@alice/lab/project-1/arizona.edu/@bob/seq=5". This naming scheme for data is illustrated in Fig. 5.1. In accordance with our data ownership model, this naming convention highlights Alice's ownership of the data by virtue of being the owner of the workspace. At the same time, the name also indicates Bob's identity as the producer of this specific piece of data. We directly utilize this information for data

[4]A *application-scoped* user identifier is highly desirable from the view of defining security policies, since it directly restricts what kind of data an application participant can produce.

[5]Sequential naming is a natural fit for this application due to reasons described in §5.2.2

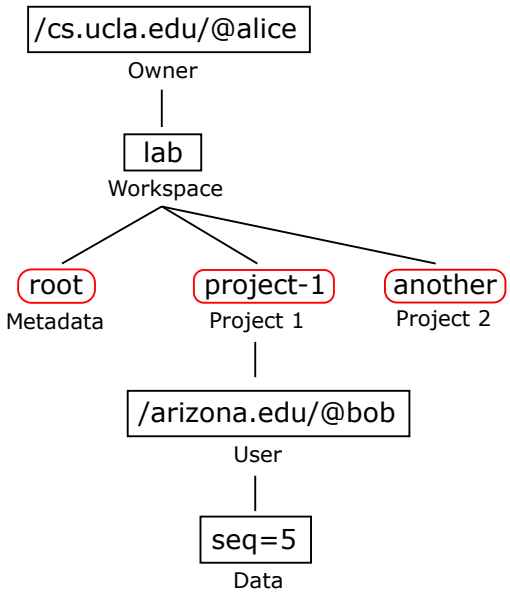

Figure 5.1: Data naming in Ownly

provenance and security, as described later in §5.4.

5.2.2 Conflict Resolution

One application requirement of Ownly is the ability for users to collaborate in real-time with automatic conflict resolution. In cloud-based applications, this requirement is easily satisfied by having the single cloud instance serve as the source of truth, allowing it to resolve any conflicts authoritatively. In a decentralized application design, we intend that no central control point should exist; thus, we must satisfy this requirement in a distributed manner.

For this purpose, we make use of an existing solution dubbed "Conflict-free Replicated Data Types (CRDTs)". CRDT encodes changes to a document, such as a text file or a JSON object, as implementation-specific *deltas*. These deltas must then be propagated to all other members of the group who are collaboratively editing the same document. CRDT provides eventual consistency with the assumption that these deltas are reliably propagated to all group members.

Applications using CRDT are a natural fit for the NDN Sync transport. NDN Sync promptly and reliably notifies all participants in a group whenever a new delta is produced by another group member. The reliability of Sync is crucial here, because CRDT does not have loss recovery capability. Also, it is usually more efficient to apply CRDT deltas in the order they are produced. This makes it convenient for each producer to name CRDT updates sequentially, allowing the application to learn the names of produced data from Sync and fetch and apply them to the local CRDT in order.

As first introduced in §5.2.1, an Ownly workspace is split to several projects. Each project in Ownly represents a single CRDT, which may cover one or more files. When users make edits to a file in a particular project, they generate a CRDT delta update for that project's CRDT, which is named sequentially within that project. This update is then propagated to all other workspace members working on *that specific project*, providing eventual consistency for the edited file. The transport mechanism for propagating these updates via NDN Sync is described further in §5.3.

5.3 Sync Transport & Storage

Users of Ownly produce named data as described in §5.2, which must be synchronized to all interested workspace members. In this section, we describe the storage and synchronization mechanisms, and how this design helps further our goal of decentralization.

5.3.1 Local-First Storage

Ownly follows the steps of the local-first [KWH19] software principles. The dataset state and data of every workspace a user is participating in are stored locally on the user's personal computer. When the user U produces data, for example by modifying the contents of a file, U first stores the generated update locally and then sends an SVS Sync Interest to inform others in the group, who will fetch the data after receiving the notification by the Sync

Interest. Updates from other users are propagated in the same way, which U uses to update its local state of the workspace. Since the user always stores an up-to-date copy of all data locally, Ownly eliminates the reliance on a remote service provider when a particular piece of data is needed.

A naive local-first approach has obvious disadvantages in terms of efficiency and local resource availability. A user of a workspace may not need or be able to store locally all data ever produced over very long periods. To improve efficiency, Ownly stores all CRDT data locally, but can also fetch large binary files from the network on demand. As described earlier, each project in a workspace is a separate CRDT. This design allows users only to synchronize and store CRDT data for the projects they are actively working on. Any other data can be fetched whenever needed.

5.3.2 Sync with SVS

Ownly uses State Vector Sync to synchronize updates between participants of a workspace. Each workspace uses one Sync group for metadata management (called the "root" Sync group) and one Sync group for each project in the workspace. When joining a workspace, a user automatically joins the metadata group using a naming convention, which in turn brings to the user the information about the list of projects in the workspace that the user may choose to participate. The root Sync group itself also uses a CRDT to synchronize its metadata. Sync groups (i.e. Sync prefixes) in the Ownly naming design are illustrated as red boxes in Fig. 5.1.

The Sync groups for each project are used to synchronize the CRDT updates for that project. Users join a project's Sync group when they desire to collaborate on that project. It is possible that the project dataset state already contains items when a user first joins, i.e. other users may have produced data earlier. In this case, the user first fetches all existing data in the project to construct the current state of the group. After this, whenever any participant of the group produces a new CRDT update, SVS informs others about the

existence of this update. Users that have joined the project then fetch this update and apply it to their local state. If a user is no longer interested in collaborating on a particular project, it may simply leave the Sync group; any existing data produced by the leaving user will continue to exist and be synchronized within the group.

The granularity and number of Sync groups represent an important design decision with some trade-offs. A larger number of Sync groups can provide a higher degree of granularity, allowing users to determine which datasets to synchronize. In a workspace with a large number of projects, for example, having one Sync group for each project reduces the number of dataset updates each user needs to process. The user also does not need to store any state for projects that they are not actively working on. However, a large number of Sync groups puts stress on the network routing system, since each group requires its own multicast forwarding state at routers. Furthermore, if a user joins a large number of projects, it would correspondingly receive a large number of Sync Interests, one for each project, thereby increasing network overhead. Due to this trade-off consideration, Ownly leaves the decision of the granularity of Sync groups to specific application designs.

5.3.3 Data Availability via In-Network Storage

When a participant of an Ownly project produces a new update, the named update data is stored locally at the producer. Sync informs other group members about this data production, and they fetch the data from the producer. However, the participants may not always be online. If the producer P_1 goes offline after data production, when another participant P_2 comes online later and cannot find a cached copy within the network, P_2 will fail to fetch the data until P_1 comes online again.

Cloud-based applications are available at all times because the application servers are always online. Being a data-centric architecture, NDN has a different definition of availability – the data itself must be available in the network at all times. NDN's Sync transport provides namespace synchronization, i.e. it informs the application of what data has been produced.

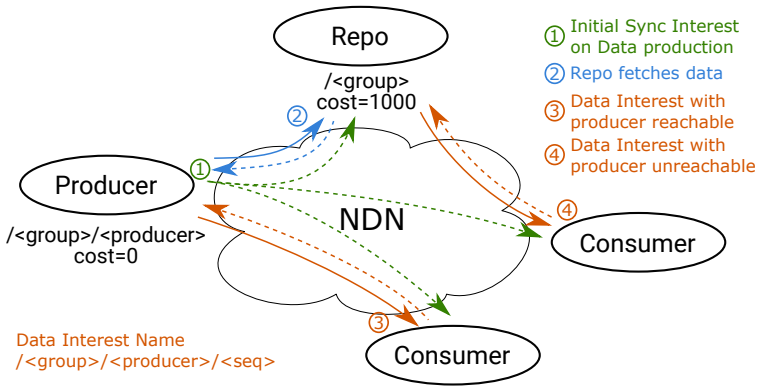

Figure 5.2: Illustration of Sync Repo

However, to enable participants to retrieve and use all the produced data, some storage service must store all the data and respond to any data requests. NDN Data Repository, or Repo [Zha20, YKM24], provides a storage service to satisfy this requirement of data availability.

Repo is an *in-network* data store that runs as part of the network infrastructure to ensure data availability for applications. When producers in an application generate new data, this data is stored in the Repo's storage. If the producer is subsequently unreachable when a consumer desires to fetch this data, Repo provides this data to the consumer. Thus, Repo enables asynchrony in NDN applications by making sure data is always available regardless of whether the original producer is reachable.

Various mechanisms have been designed for the exact process of storing data in a Repo. In earlier designs of Repo [YKM24] and other NDN storage systems [PDZ22], applications explicitly instruct the Repo to fetch specific pieces of data. Ownly uses the oblivious Sync Repo [YZM24] design, which makes Repo functionality transparent to the application. With Sync Repo, an Ownly instance instructs the Repo to join one or more Sync groups. In this case, the network Repo joins the metadata Sync group and each project's Sync group. The Sync Repo then passively listens to data production on all registered Sync groups, fetches data produced by the groups, and stores it in the Repo storage.

Sync Repo enables transparency by announcing a shorter, higher-cost data prefix for each group to the network. When the producer of a given piece of data is online, the producer announces a longer low-cost prefix, and all Interests for matching data are forwarded to the producer directly, including Interests generated by Repo to fetch the data from the producer. When a producer goes offline, Interests for its data are automatically forwarded to the Repo via routers's longest prefix match in forwarding. As a result, consumers are unaware of the location of the data or the state of the producer to retrieve it. Instead, in line with the spirit of NDN, the network takes responsibility for finding matching data for the Interest, either from the producer or a Repo instance. The functioning of Sync Repo is illustrated in Fig. 5.2.

In addition to storing data, Sync Repo also stores the latest dataset state of each Sync group it joins, helping users who are newly coming online catch up with the latest data production of their group. Since Repo itself is not an active member of the application, it does not generate Sync Interests because it cannot sign SVS state vectors. Instead, Sync Repo buffers the most recently received state vectors, and replays these state vectors in response to receiving outdated Sync Interests sent by participants who just come online and are falling behind the group's data production. As a result, members of a Sync group can learn about the latest dataset state of the group even when no other group member is online. Note that Repo does not send Sync Interests periodically to avoid unnecessary overhead.

It is important to note that Repo only stores signed and encrypted data that is already available on the network. Repo does not understand application semantics, nor can it produce application data or decrypt any data it stores. Instead, Repo offers storage resources in a fully transparent way to applications, working in the background to ensure data availability at all times. This design ensures that Repo has no control over user applications or user data, and makes it easy for users to switch to different Repo services if desired. As a result, the task of storing data can be safely offloaded to storage service providers who can benefit from scale, without locking up users or making users lose control over their data.

The transparent nature of Repo also clears the path for improvements to the design of Repo itself. Since the application is largely oblivious to the existence of Repo, the internal functioning of Repo can change freely. This may include any future distributed and fault-tolerant designs of Repo, in which Interests for data from the application may be routed to the nearest storage replica via anycast forwarding, similar to the design of Kua [PDZ22].

5.3.4 Snapshots

Over time, workspaces may accumulate large amounts of data. Since CRDTs exchange updates as deltas, this results in a large number of updates being produced. When a new user U_{new} joins a workspace[6], U_{new} must first fetch all previously produced updates one-by-one and process all these deltas. This process is highly inefficient, both computationally and in terms of network overhead. It also makes the process of bootstrapping a new user very slow, leading to a significant impact on application usability.

CRDT implementations typically support the ability to combine multiple updates into a single update, which is more efficient to process. This functionality can also be extended to combine *all* updates of a CRDT into a single update, directly and efficiently representing the current state of the CRDT. We utilize these abilities of CRDTs to produce "snapshots" that greatly reduce the number of data pieces that a new user needs to fetch.

We considered and implemented two different designs for snapshots in our collaborative application. In the first design, a random member of the group periodically takes a snapshot of the *entire state* of the group (Fig. 5.3). This snapshot is stored at the snapshot producer, and optionally at all other group members and in Repo. When a new user U_{new} bootstraps into the group, U_{new} first fetches the latest available snapshot, and then fetches any updates that were produced after the snapshot was taken. This design of a *group snapshot* allows new users to quickly bootstrap into the Sync group. Each snapshot uses the state vector itself

[6]This effect is also observed when a user synchronizes a workspace project after being inactive for a while.

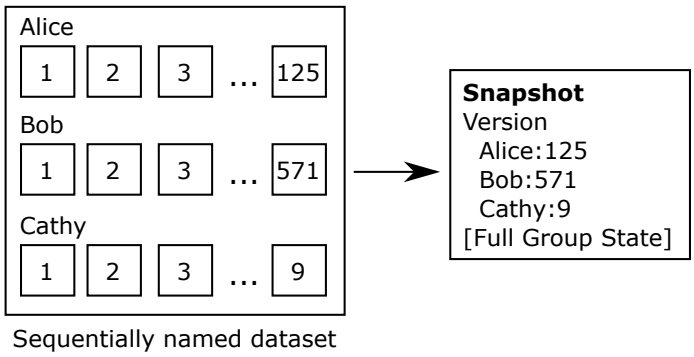

Figure 5.3: Group Snapshots

as the version number, as it directly represents the entire state at that point in time. Upon learning the snapshot version and the latest state vector, a new user can directly compute the difference to determine the remaining updates it needs to fetch.

The group snapshot design has the downside of high storage and network usage costs. Each snapshot is relatively large, since it contains the updates of the entire group over time. Storing the snapshot in Repo requires the transfer of the entire update set to the Repo every time a new snapshot is created. Our experience with the trial deployment of group snapshot shows that, even if a user P already has a partial state of the group, P still needs to fetch either the entire snapshot (which may be wasteful) or fall back to fetching each update individually (which is slow).

To avoid the above-identified issues, Ownly developed a different snapshot design. In Ownly, each producer periodically merges and publishes a snapshot of *its own* updates *since its last snapshot*. This snapshot is stored locally at the producer and in Repo. In this design, when a user U bootstraps into the workspace, it needs to fetch multiple snapshots from each participant in the group before it reaches the latest dataset state of the group. Each snapshot is essentially a compressed version of all updates within a range of sequence numbers, and can be fetched as a single data object. A versioned snapshot index tracks the existence of each snapshot. When U desires to fetch snapshots, U first fetches the index for each other user and identifies the list of snapshots it can fetch to speed up the process of reaching the

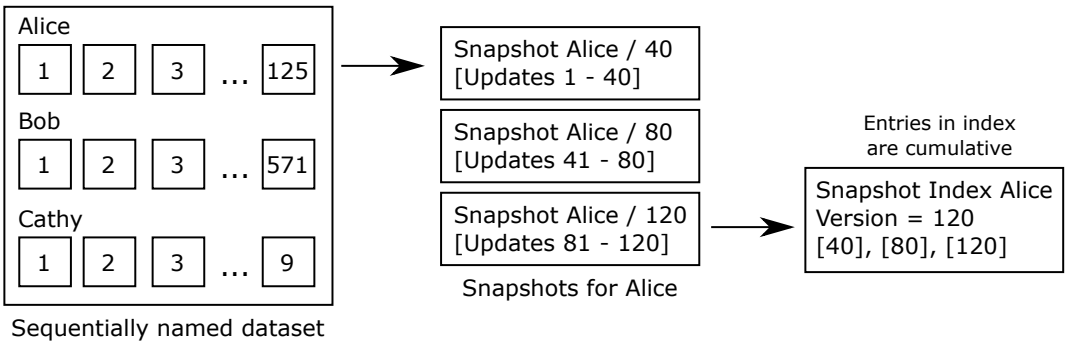

Figure 5.4: Producer Snapshots

latest state.

This *producer snapshots* design (Fig. 5.4) has the downside of being more complex and requiring users to fetch multiple snapshots to reach the latest state. However, unlike group snapshots, users no longer have to store and transfer the entire state of the group every time a snapshot is created. Instead, only a compressed version of the last few updates for a particular producer must be stored and transferred. This design also allows a consumer P with partial state to cherry pick the snapshots it desires to fetch, allowing P to converge to the latest state quickly and efficiently. We note that this snapshot design may be further extended for another layer of indirection, where a larger number of very old updates are bundled together. These "cold" publications are likely to be fetched only when a new user bootstraps into a project group, and bundling them together helps reduce the number of "warm" snapshots that users need to fetch and store.

The snapshot design represents a trade-off between bootstrapping time and overhead. In the Ownly design, we use producer snapshots, but keep doors open for better snapshot designs in the future.

5.3.5 Offline Support

NDN Sync protocols utilize any available connectivity to propagate dataset changes to other members of an application group. Since Ownly uses Sync for all communication, it directly inherits the ability to work well in scenarios with network partitions. The Ownly application does not distinguish "online" and "offline" states; instead, Sync Interests are forwarded on any available connectivity. Updates produced by a user are stored locally regardless of the status of connectivity. When disconnected from the network, Sync continues to attempt to send periodic Sync Interests, but these are simply dropped locally due to the absence of usable outgoing faces. As soon as the user re-establishes connectivity, Sync automatically propagates all previously produced updates to the group, and vice versa.

From the above description, we note how NDN Sync enables offline support transparently, without introducing any additional complexity at the application layer. This ability of Sync is in sharp contrast to client-server applications in TCP/IP, which typically need to implement additional logic to separately store and synchronize the effects of any actions a user takes while disconnected from the server. This functionality in turn is enabled by NDN's ability to make use of any available connectivity, because its namespace is independent from topological connectivity. This is in contrast to IP addresses which refer to attachment points in the network, meaning that offline instances have no meaningful identifiers in the IP namespace.

5.4 Security

In an Ownly workspace, only authorized users are allowed to read and modify the contents of projects in the workspace. In this section, we describe the security model of Ownly.

5.4.1 Bootstrapping

Every entity in an NDN network must undergo a process of security bootstrapping [YMX23a], which provisions the entity with a trust anchor, an identity certificate, and a set of trust policies. In the Ownly design, we assume that users learn one or more trust anchors for the system out-of-band, and all certificates in the system are traced back to one or more of these trust anchors. To simplify bootstrapping using DNS names as described in §5.2.1, we assume the existence of one or more identity verifiers. These verifiers provide users the ability to obtain a certificate of the ownership of their unique identifier, which in turn is derived from a DNS name or email address. In our implementation, we utilize the existing deployment of a set of NDNCERT [ZAZ17] CAs on the global NDN Testbed [Nam25f] to serve as identity verifiers.

When a new user N wants to bootstrap into an Ownly workspace, N first generates a new signing key pair, and requests an identity certificate from a verifier. The chosen verifier must be trusted by all members of the workspace. The verifier may ask the user to perform one or more challenges to prove ownership of a particular identifier before issuing this certificate. The user may now use this key pair to sign any newly produced data.

5.4.2 Trust Policy

Data signatures allow consumers to verify the authenticity of the data producer and the integrity of the data. However, signatures alone do not validate whether the producer is authorized to produce data with a particular name. This function is achieved by using a trust policy. In NDN, trust policies define the relations between semantic data names and the names of the key used to sign the data, and thus can be schematized [YAC15] for automated enforcement. In Ownly, the application bundles such a schematized trust policy which can be extended at runtime.

At a high level, a static trust policy in Ownly is quite straightforward. Fig. 5.5 shows

```
#user_cert: #owner/wksp/#user/#KEY <= #owner_cert
#owner_cert: #owner/wksp/#owner/#KEY <= #owner_id_cert
#owner_id_cert: #owner/#KEY <= #verifier_site_cert | #verifier_root_cert
#user_id_cert: #user/#KEY <= #verifier_site_cert | #verifier_root_cert

#verifier_site_cert: /"ndn"/_/_/#KEY <= #verifier_site_cert
#verifier_root_cert: /"ndn"/#KEY

#proj: #owner/wksp/proj
#proj_data: #proj/#user/_/_ <= #user_cert
#proj_blob: #proj/#user/_/"32=blob"/_ <= #user_cert
```

Figure 5.5: LightVerSec for static trust schema

a subset of the trust policy rules written in LightVerSec [YMX23b]. These rules are briefly described below.

1. Users who possess a valid *scoped* user certificate (`user_cert`, see §5.2.1) are allowed to produce data for projects (`proj_data`) in the workspace.

2. Only the owner is allowed to sign scoped user certificates (`user_cert`), and it must present the scoped owner certificate (`owner_cert`) to be authorized to do so.

3. The scoped owner certificate (`owner_cert`) must be signed by the owner's identity certificate (`owner_id_cert`). Notably, scoped certificates for other users cannot be signed by their identity certificates, or the owner's identity certificate.

4. Any user's identity certificate including the owner[7] must be signed by a trusted verifier.

[7]The separate rules shown for the two types of identity certificates are redundant in the LightVerSec syntax, and are only shown for clarity.

Some verifiers ("sites") may in turn have their certificate signed by the trust anchor ("root").

The trust policy defined by this schema enforces that only users that are explicitly authorized by the workspace owner are allowed to publish data in the namespace of the workspace, which in turn lies under the namespace of the owner. This provides the owner with full control of data under the workspace without requiring a third party to manage access control. When a participant P produces data within the group, all other members in the workspace can consult this trust policy and determine whether P is authorized to produce the given piece of data. This in turn will be true only if P has been issued a certificate by the owner of the workspace.

5.4.3 Invitations

Ideally, the owner of a workspace (the "inviter") can directly authenticate a user ("invitee") they wish to authorize for data production in their workspace, and issue the invitee a scoped user certificate that would satisfy the trust policy. In this design, workspaces can have their own trust anchors, thereby reducing the attack surface and providing a workspace owner with full control.

In practice, however, this design may be inconvenient for users. When an invitee wants to bootstrap into the workspace, it must wait for the inviter to issue a certificate before being able to produce or access data in the workspace. To avoid this wait, the owner must either be always online or run additional infrastructure such as an NDNCERT CA instance for each workspace. Such requirements may be seen as detrimental to usability of the application.

To allow for a greater degree of asynchrony, the Ownly design provides the ability for workspace owners to authorize users to issue their own scoped user certificates. When the owner wishes to invite a user N into their workspace, the owner creates a new trust policy rule that allows N's identity certificate (`user_id_cert`) to sign the scoped identity of N

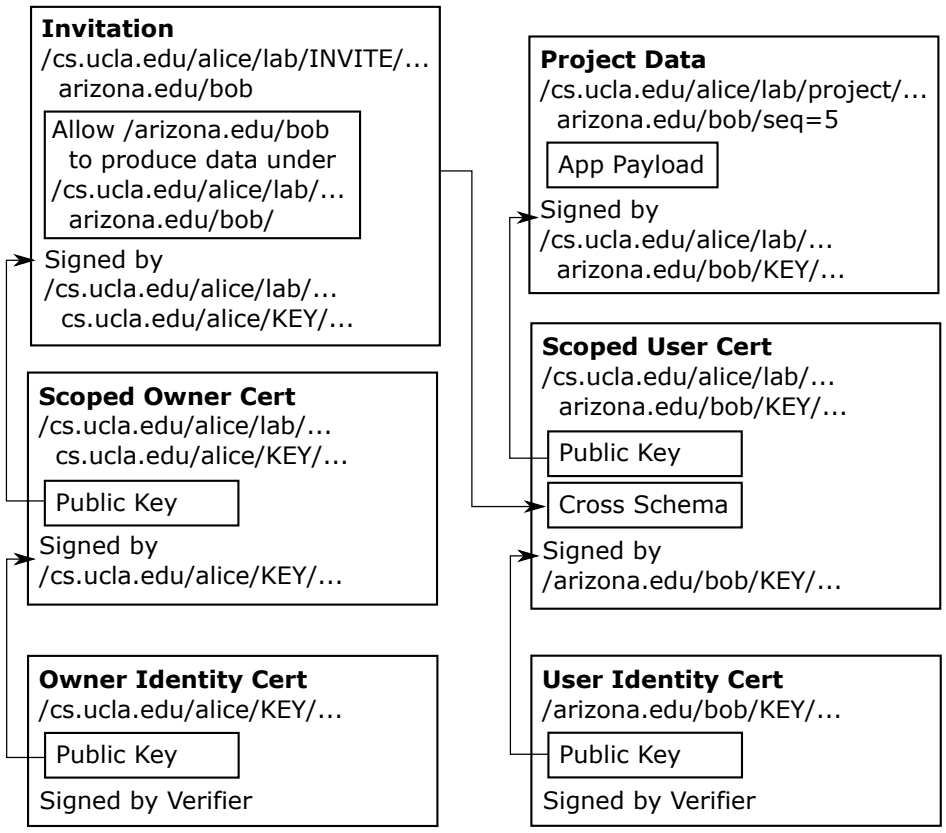

Figure 5.6: Invitations with CrossSchema

(`user_cert`). A policy rule signed by the workspace owner, called an *invitation*, allows the invitee to self-bootstrap into the workspace.

When a user N with a valid invitation bootstraps into a workspace, N signs a scoped user certificate for itself using N's identity certificate. The signed policy rule (i.e. the invitation) that allows N to perform this action is then stapled to the newly issued certificate. When a consumer receives this certificate during data verification, it takes the stapled rule into account when deciding whether N is authorized to sign such certificate. The policy rule in turn is also verified, and the consumer validates whether the signer of the rule is allowed to create such a new policy rule. This process is illustrated with an example in Fig. 5.6.

Dynamic policy rules, dubbed *CrossSchema*, allow the application to update the trust policy at runtime. This powerful mechanism may be further extended to expand Ownly's

simple trust model. For instance, CrossSchema may be used by the owner to authorize a particular user to invite other users into the workspace, or to perform other administrative tasks. Another type of rule may serve as an "open" invitation, allowing any user with a verified identity to join a given workspace. The introduction of this new primitive significantly expands the capabilities and scalability of NDN trust schema.

5.4.4 Data Confidentiality

Previous work suggests using Name-based Access Control [ZYR18] to provide confidentiality in NDN applications through encryption, with automated key management. As of the time of this writing, Ownly's design of data encryption continues to be a work in progress. The description below reflects the preliminary design and implementation.

Data in a Sync group in Ownly is shared among all participants in the group. Thus, it may be possible to use a simple group key that is regularly rotated to provide the group with confidentiality. This key must be provisioned at each participant in the group as part of the security bootstrapping process, allowing the participant to decrypt current and past data produced by the group.

Ownly uses two shared secrets to create this encryption key. A *Pre-Shared* Key (PSK) is exchanged between the users of the workspace out-of-band, for example with the URL inviting a user to the workspace. A *Dynamic* Shared Key (DSK) is managed within the group and may be rotated over time. When a user N bootstraps into the workspace, N requests the DSK on the metadata Sync group of the workspace. Any other user may then securely provision N with the DSK using an asynchronous Diffie-Hellman exchange[8]. On receiving the DSK, N combines the PSK and DSK using a suitable Key Derivation Function (KDF) to obtain the group key for the workspace[9].

[8]The initial request for the DSK and the response contain the required parameters for the exchange.

[9]Our current design assumes a single key for the entire workspace; it is straightforward to extend this design to have keys specific to projects that are provisioned only to users authorized by the trust policy.

Notably, this design protects against two types of attacks by splitting the shared secret. First, a compromised PSK alone does not allow an attacker to decrypt contents of a workspace, since the DSK is only provisioned to authorized invited users, who in turn are verified by a trusted identity verifier. Second, a malicious identity verifier cannot decrypt the workspace contents without knowing the PSK, which in turn is only shared out-of-band with authorized users. Further development of the encryption design, such as to incorporate key rotation, should be considered as future work in the development of Ownly.

5.5 Implementation

We implemented [IRL25] Ownly as a browser-based application written primarily in Go and TypeScript, using the NDNd [Nam25d] standard library and Vue. Our deployment[10] of Ownly utilizes the NDN Testbed [Nam25f] for NDN connectivity and uses an instance of NDNd Sync Repo deployed on the testbed to provide in-network storage.

We made a deliberate decision to build Ownly as a web application, recognizing the web as a uniquely powerful and accessible platform. Web applications offer a seamless user experience with zero installation, run across all major vendor devices, and support modern capabilities required for local-first applications. Ownly in particular utilizes offline support, WebAssembly, IndexedDB storage, and the Origin Private File System (OPFS). These features enable performant, local-first applications with native-like responsiveness and robust offline support, making the web the ideal choice for balancing usability and capability.

We continue to use and improve the design and implementation of Ownly. Ongoing work includes improving the encryption design of Ownly, enabling the use of multiple connectivity and identity verification options, and expanding the application's functionality beyond collaborative editing.

[10]The deployed web application can be accessed at `https://ownly.work`

5.6 Discussion

In this section, we present some additional discussion and insights gained from the development of Ownly.

5.6.1 Enabling Decentralized Apps

The Internet started as a decentralized network, but the rise of service providers led to gradual centralization over the last few decades. In addition to economic issues, we attribute this trend to a few important technological factors.

1. The rise of NAT to combat IPv4 address space exhaustion marked a fundamental shift in the network model of the Internet. While IP itself is symmetric, NAT created an inherent asymmetry in communication; clients behind NAT can initiate outbound connections, but cannot easily accept inbound ones. This asymmetry relegated most consumer devices to "client-only" roles, while a small number of publicly reachable servers, often hosted by large data centers or cloud providers, assumed the role of service providers. This reinforced the centralization trend, as deploying a globally accessible service increasingly required a public IP address and infrastructure outside the reach of everyday users. NAT also complicates peer-to-peer protocols, pushing many applications toward centralized coordination points or relays (e.g., STUN/TURN servers for WebRTC, matchmaking servers for games).

   Thus, what began as a pragmatic solution to mitigating IP address scarcity became a key architectural inflection point, steering the Internet toward a hub-and-spoke model dominated by large-scale service providers, eroding the universality and symmetry that the Internet was originally built upon.

2. A foundational shift in the architecture of Internet applications came from the lack of verifiable identities for individual users and devices. IP addresses identify network

locations; they do not provide any information about *who* is communicating or *what* they are allowed to say. This absence of cryptographically secure user-level identity made it extremely difficult to build secure peer-to-peer applications without relying on third parties. Application developers offloaded the burden of establishing trust relation to centralized identity providers (IdPs), first in enterprise settings and later in the consumer web.

This model has effectively made the cloud not just a convenience, but a requirement for secure communication. If two users want to securely message, game, or collaborate, they typically authenticate through a trusted third party that vouches for their identity and mediates communication. Cloud-hosted IdPs enable this at global scale, assigning stable, universally recognized identities (like email addresses or OAuth tokens) that developers can depend on. A key shortcoming of these cloud-based identities is non-portability; they are typically tied to specific platforms and subject to unilateral control, directly leading to the negative consequences of centralization.

3. The lack of network multicast support in IP networks makes scaling content delivery inherently challenging and capital-intensive. This directly led to the rise of centralized content delivery networks (CDNs). CDNs represent a solution to the inherent mismatch between IP's connection-based communication model and applications' requirements for content dissemination.

NDN addresses these issues directly by shifting the network to a data-centric paradigm. First, we note that building decentralized applications remains challenging even with the availability of end-to-end connectivity. With IPv6, for example, user devices should be able to accept inbound connections from other devices on the Internet directly. However, the initiator of the connection still needs to know the IP address of the intended destination, which may not be stable, especially with user mobility. This reintroduces the need for a central controller with a well-known stable address as a rendezvous point to allow users to

meet peers.

NDN eliminates the need for this indirection by directly using semantic application layer data identifiers (i.e. names) in network layer communications. With NDN, knowing the name of the desired data is sufficient for applications to obtain a copy of the data, without the need to find the *location* of the desired data. This data-centric design better aligns with application requirements than IP's host-centric communication; applications typically care only about the data itself, not the address of the data containers.

By securing data directly, NDN also eliminates the need for cloud providers to act as intermediaries for secure communication. NDN provides the primitives required to bootstrap a user or device with an identity trusted within the application; this identity is then directly verifiable by others in the network, without requiring a third party. We note that NDN applications still require a unique identifier for each user or device, and we use the well-established Internet namespace, DNS, for this purpose. The key feature that NDN provides is the ability for end users to *directly* use this identifier for secure communication, without reliance on centralized servers to meet peers.

Data-centric security is also the key on why NDN applications need not depend on centralized CDNs. Early attempts at scaling content delivery on the Internet focused on caching HTTP proxies; however, this approach became unusable with the introduction of encrypted SSL pipes. CDNs stepped around this issue by assuming centralized control of all caches in the network and impersonating the identities of CDN customers in good faith, to deliver content to users over encrypted channels. In NDN, data is secured directly and thus can be cached anywhere in the network without the need for a centralized trustworthy gatekeeper, i.e. the CDN operator. This design makes multicast and content delivery accessible[11] to everyone in the network, and furthermore significantly improves the security properties of the system by eliminating middlemen and restoring the end-to-end principle.

[11]Besides being control points, commercial CDNs also only serve data for *paying* customers.

5.6.2 The Role of Transport Service

The transport layer plays an important role in the Internet centralization, it is more so due to the tight coupling of security with transport protocols.

One noteworthy issue is that new transport protocols face serious deployment barriers. Because of widespread reliance on NATs, firewalls, and traffic-shaping middleboxes. Any protocol that deviates from the well-known TCP or UDP port numbers tends to get blocked or degraded. This has effectively frozen the transport layer, making innovation extremely difficult. This resistance to new transports has led developers to move up the stack and rely on centralized infrastructure – cloud services, CDNs, and HTTPS-based APIs, where they have more control over performance and reliability.

Moreover, even within TCP and UDP, the need to traverse NATs has led to architectural workarounds like TURN relays and HTTP tunneling, which often require trusted intermediaries. Direct user-to-user communication becomes fragile or infeasible without a third party mediating the connection. As a result, many applications, even those that could theoretically be peer-to-peer, default to a client-server model hosted in the cloud, where connectivity and security issues are abstracted away.

Transport Layer Security (TLS) also contributes to centralization by reinforcing a server-centric trust model. In the common case, only servers possess DNS names and certificates issued by a third party certificate authority (CA), while the client ends remain anonymous below the application layer. Although mutual TLS (mTLS) allows both parties to authenticate each other with certificates, it is rarely used outside of tightly controlled enterprise environments, primarily due to the lack of standardized user-specific and application-independent identities. Today's reality is that authentication of server identities relies on a small set of centralized Certificate Authorities (CAs), while users have no app-independent identity. This makes it easy for large platforms to secure connections at scale, but difficult for decentralized, peer-to-peer systems to establish connectivity and trust without relying on these

same authorities. Even though TLS is symmetric in theory, in practice, it enforces a model where users can only authenticate to the cloud, not to each other.

The communication model of cloud infrastructure with channel-based communication has also shaped application designs to inadvertently depend on transport properties. Protocols like TCP are stateful, requiring a persistent connection. This works well in data centers, where uptime is high and network connectivity is stable; however, it is poorly suited to decentralized systems, where nodes may be ephemeral or intermittently connected. The transport layer also historically lacks support for mobility (changing IP addresses) or multihoming (using multiple interfaces/IPs concurrently), which are common in user-facing, decentralized scenarios. As a result, today's applications are largely designed with assumptions of a constantly available centralized service provider; this inherent bias further complicates decentralization.

NDN introduces the idea of channel-independent communication and security, eliminating most of these issues directly at the network layer. NDN enables secure peer-to-peer communication, allowing each entity to directly use a unique identifier that is independent of any service provider. Since NDN does not rely on addresses for communication, it also directly supports mobility and multihoming, By securing data directly, it does not rely on stateful connections that TCP/IP uses for security, and that may hinder ad-hoc communication.

Instead of being a reliable data pipe, the Sync transport provides a new type of service which enables application participants to discover what data has been produced. As demonstrated by the design of Ownly, this crucial function enables users to work together and exchange data in real time, without the need of a centralized server for coordination. Our design emphasizes the need for efficient and reliable NDN Sync protocols, which are fundamental to our design of State Vector Sync, as described in Chapter 3.

5.6.3 Future Work

During our implementation and trial deployment of NDN Workspace, we identified end-user usability as one of the key factors preventing widespread usage of decentralized applications. We plan to further focus on this aspect in future work on Ownly, with the goal of improving the practical aspects of security. Some future directions are briefly described below.

1. Tooling to manage keys and certificates would greatly benefit and simplify decentralization, allowing users to maintain a stable cryptographic identity. This requirement has been previously identified and implemented in various forms, such as PGP, but end-user usability continues to be a major concern, preventing widespread adoption.

2. Our implementation of Ownly uses a simple trust model to enforce data authentication and authorization in a group. We encourage further exploration of more flexible name-based trust models that can be easily defined by schematization. Such models can be more effective in demonstrating improvements in scalability and trust management for large, distributed systems, especially when compared to traditional approaches, such as centrally managed access control lists.

3. Ownly's design and implementation of access control and encryption is currently in nascent stages. We plan to further iterate these aspects of the design, incorporating key rotation and access revocation. Preserving usability while allowing effective membership management without a central, always-available controller is considered particularly challenging.

4. To enable Ownly to be truly decentralized, the network infrastructure itself should also be sufficiently decentralized. Further research on protocols to enable secure peering between independent autonomous systems running NDN networks would be crucial in this regard.

We also intend to further explore the decentralization of other types of applications running over NDN, with the prudent note that collaborative applications such as Ownly are relatively low-hanging fruit in the story of Internet decentralization. For example, applications that require infrastructure to perform resource-intensive computation introduce additional challenges to decentralization, which are not addressed by our present design of Ownly.

CHAPTER 6

Other Contributions to NDN

This chapter is a brief overview of some of the author's contributions to the NDN ecosystem not described previously, including implementation, tooling and deployment.

6.1 Named Data Networking Daemon

Over time, several software packages have been developed within the NDN ecosystem, including forwarders, routing daemons and application libraries. These libraries and infrastructure components have evolved significantly over time through a process of iteration. We built the Named Data Networking Daemon (NDNd) package [Nam25d], a new *consolidated* implementation of the entire NDN stack. Our implementation of NDNd represents a new set of priorities based on the lessons learned from previous NDN implementations, which are briefly described below.

1. **Supporting modern languages:** NDNd is implemented in pure Golang, providing cross-platform support, automatic memory management and a broad set of available libraries. Several previous NDN implementations were written in lower-level languages such as C++. The use of Go allows NDNd and application developers to focus on functionality rather than low-level details.

2. **High-level APIs:** Application developers typically use high-level APIs such as RPC or publish-subscribe. To enable developers to build NDN applications with ease, we focus on building similar or equivalent high level APIs in NDNd. Some of these APIs

are opinionated by design, providing developers with a well-tested starting point for building their applications.

3. **Ease of deployment:** A recurring operational challenge for network deployment is the complexity of configuration. One of the goals of NDNd is to ease the process of deploying an NDN network or overlay, by providing the required tools for autoconfiguration and relevant documentation. NDNd also aims to reduce the number of separate components required to deploy an NDN network.

The NDNd implementation is divided into several components described below. NDNd is built as a single statically linked binary; while all components are bundled in this binary, each component can be utilized independently using a command line interface. We also make prebuilt binaries available for various platforms and architectures for the convenience of network operators. This methodology of packaging and software delivery is an important step towards our goal of easing the deployment of NDN networks. Previous implementations of NDN often required operators to install various dependencies and toolchains to compile from source. With NDNd, operators may simply download and directly deploy the appropriate binary for their platform.

6.1.1 Standard Library

At the core of NDNd, the `ndnd/std` package provides a standard library for developers to build NDN applications. All other packages in NDNd itself are built using the standard library. We briefly describe various components and their functionality within the standard library below.

1. The `encoding` package provides a high performance implementation of NDN TLV encoding and decoding, first described by Ma et al. [MAZ22]. NDNd allows application developers to semantically define TLV structures using Golang structs; a code generator

then emits optimized functions for encoding and decoding these structs with strict typing. This package also provides utilities for working with NDN names, including optimized hash functions and pattern matching.

2. The `engine` package is an implementation of low-level network APIs, such as Interest-Data exchanges. The engine implementation serves as the local NDN face of an application, and is responsible for matching and dispatching Interest and Data packets to the attached hooks. This API may be considered too low-level, and is not recommended for direct usage by applications.

3. The `ndn` package contains wire format definitions for various standardized NDN protocols, including the NDN packet specification [Nam25a].

4. The `object` package provides an API that allows applications to produce and consume signed NDN data objects. This high-level API hides details such as segmentation, loss recovery and data signing from application developers, providing a simple interface that may seem vaguely similar to HTTP. The library also provides additional functionality such as congestion control and various data storage implementations.

   Application developers may use this API by instantiating a client with a security configuration, consisting of a set of trust anchors, a keychain[1] and a security policy. The client API then provides two important functions.

   (a) `produce(name, content)`: this method segments the provided content depending on the configured MTU, and produces signed NDN Data with the selected name. The name argument is used to determine a suitable key from the keychain by consulting with the security policy. An error is returned if no matching key is found, otherwise, the generated Data packets are stored in a local data store of the developer's choice. This method is intended to be used for the production of

[1]The keychain is a database that stores the private keys that are owned by a given NDN application instance, along with any corresponding or generally useful certificates.

a single Application Data Unit (ADU), which may then be fetched by another application instance.

(b) `consume(name, callback)`: this asynchronous method attempts to reliably fetch the Data object with a given name, verifies the received content against the configured security policy, and returns the content blob to the application. An error is returned if fetching one or more segments fails, or if the object producer cannot be verified. If consume is called without a versioned name, it first attempts to discover the latest version of the object using a discovery Interest[2] for the metadata of the requested object; this API eases the process of using versioned objects.

5. The `schema` package provides an implementation of Name-Tree Schema [Ma24], an alternative high-level NDN API.

6. The `security` package implements various NDN security primitives. These include signing and verification of NDN Data using various cryptographic techniques[3], key and certificate encoding and storage, and the implementation of a LightVerSec [YMX23b] validator. The high-level APIs provided by this package enable applications to define well-structured name-based security policies, which are then directly and automatically executed by the aforementioned network APIs.

7. The `sync` package contains implementations of NDN Sync protocols. Currently, NDNd provides high-level APIs for State Vector Sync and SVS Pub/Sub; these libraries automate various functions including the execution of applications' security policies.

The standard library also comes with various examples and documentation for using the packages described above. We note that along with support for most native platforms, this

[2]Discovery Interests carry the `MustBeFresh` and `CanBePrefix` flags, so as to encourage matching only the latest version of the metadata at the producer.

[3]As of writing this, NDNd supports signatures with RSA, ECDSA w/ NIST curves, and Ed25519

package also supports WebAssembly; this target is used by the implementation of Ownly, as described in Chapter 5.

6.1.2 Packet Forwarder

The `ndnd/fw` package provides an NDN packet forwarder based on the YaNFD [NMZ21] implementation. The NDNd forwarder is a multithreaded implementation that runs entirely in userspace. The forwarder approximately follows the design of NFD's forwarding pipeline, as described in the developer's guide [ASZ21], and is compatible with the NFD management protocol. The NDNd forwarder implementation incorporates various optimizations and improvements to the original YaNFD implementation. Some of these are described below.

1. We replaced the TLV processing primitives in YaNFD with the high-performance implementation of `ndnd/std/encoding`, significantly reducing latency in the forwarding pipeline.

2. For most forwarder functions, it may be unnecessary to decode all fields of an NDN packet, an example being the `SignatureInfo` carried with all NDN Data. We optimized the forwarder to decode only the sections of packets essential for forwarding.

3. We simplified the implementation of the forwarding pipeline and packet dispatch, minimizing the number of copies required. We note that the current implementation continues to copy each packet more than once; this may be further optimized.

4. Garbage collection represented a significant bottleneck in previous evaluations of YaNFD. To alleviate this issue, we systematically reduced the number of allocations performed in the forwarding pipeline using Golang's profiling tools.

The NDNd forwarder represents a significant evolution in the implementation of userspace NDN forwarders. We conducted some preliminary experiments using an Ubuntu 22.04 machine with an AMD EPYC 7702P (64 core, 1.5 GHz) and 256GB memory, comparing NDNd

with NFD and the older YaNFD implementation. We connected four local producers and four local consumers to each forwarder, and restricted the multithreaded forwarders to 8 forwarding threads each. Testing with various payload sizes, NFD and YaNFD provided a sustained goodput of 1.5 to 3 gbps, while the NDNd forwarder consistently provided upwards of 10 gbps. We aim to further improve the performance of NDNd in the future and expand the suite of benchmarks with which it is tested. We also plan to implement the two-step forwarding process described in Chapter 4 in the NDNd forwarding pipeline, allowing the usage of this more efficient forwarder design.

6.1.3 Routing Daemon

The reference implementation of the NDN Distance Vector Routing (ndn-dv) (Chapter 4) Daemon can be found in the `ndnd/dv` package. The ndn-dv implementation is designed to be easy to configure and has security support out-of-the-box. We note that this implementation is currently also compatible and can be directly used with the NFD forwarder.

To further ease the deployment process, the NDNd `daemon` also allows operators to configure and run the NDNd forwarder and ndn-dv together, as a single entity.

6.1.4 Tooling

`ndnd/tools` provides several useful tools that are useful in debugging NDN applications and networks. Some of these are briefly described below.

1. The *ping* utilities allow operators to test connectivity within an NDN network. A ping server acts as a tiny data producer that announces a given prefix to the network. The ping client generates Interests for the given name, which a ping server may then respond to.

2. The *chunks* utilities work in a manner similar to ping, but produce segmented data

objects rather than dummy echo data. Besides connectivity, these utilities can be useful for testing throughput and congestion within the network.

3. Several keychain and security utilities allow developers to generate keys and certificates for testing their applications. The NDNCERT client allows developers or users to manually request an identity certificate from an NDNCERT CA running on the network.

4. This package also contains implementations for the control plane utilities that can be used to manage the NDNd forwarder using the NFD management protocol, and likewise for the ndn-dv routing daemon.

6.1.5 Testing Framework

NDN software and applications are typically distributed in nature. Past experience has shown that unit and integration testing may not be sufficient for such software, and comprehensive end-to-end tests may be considered necessary for robustness. NDNd uses a testing framework built over MiniNDN [Min25] for automated end-to-end testing. This framework allows developers to easily define topologies and scenarios involving node and link failures. In the future, we plan to expand this framework to add more capabilities, such as automating the collection of metrics such as network usage and latency.

6.2 Containerizing the NDN Testbed

Testing infrastructure is one of the key requirements of developing network protocols; as a future Internet architecture, the NDN shares this requirement. The NDN Testbed is a publicly usable NDN network running as an overlay over the existing IP infrastructure, with routers at volunteering universities and institutes around the world. The Testbed provides a platform for developers to connect to a real NDN network to run and deploy their

applications.

First deployed over 10 years ago, the legacy Testbed infrastructure became increasingly harder to maintain reliably over time, with more failures, inconsistencies and the need for frequent operator intervention. To alleviate the maintenance overhead of the Testbed, we built an NDN container package [Nam25c] that can easily scale to a large number of routers. While the package currently has elements bespoke to the Testbed, we believe it can be generally usable by operators who wish to deploy their own production NDN network.

Our containerization process was two-fold. First, we built container images for all NDN software, and then set up router-specific orchestration using Docker Compose. All images are currently built using automated CI/CD pipelines for each named git tag, along with a biweekly build that is tagged with the build date. We use reusable GitHub Actions workflows to enable lower maintenance overhead and consistency across repositories.

To automate configuration, we built a small configuration rendering engine using Jinja2, allowing us to directly port the legacy Ansible configuration to the new system. During deployment, each host is configured with a node name. A cron container runs on each node, periodically pulling changes to the authoritative Testbed GitHub repository. Whenever there are any changes, the rendered configuration is updated by the cron container, and any affected containers are restarted using the Docker API[4].

For monitoring the Testbed, we also designed a new status page [Nam25g]. Unlike the legacy version, the new status page itself operates entirely over NDN. Each router periodically collects prefix reachability and latency data to all other routers on the Testbed, and registers a prefix to serve this data. When an operator opens the status page, the browser connects to the geographically closest Testbed router using the NDN-FCH service over an NDNts WebSocket face. We then directly fetch the latest status data from each router from the browser and present it to the operator.

[4]We mount the Docker socket from the host to the cron container to enable it to manage containers.

Our deployment of the containerized NDN Testbed has significantly reduced maintenance overhead and has been running in production for over a year. In the future, we plan to make the container package more generally usable for other operators that wish to deploy their own Testbed network, improving the documentation and providing more security configuration options.

6.3 NDN-Play

Despite significant advances in recent years, developer tooling for NDN applications remains relatively scarce. This scarcity presents a major roadblock for building new NDN applications. To address this problem, we built *NDN-Play*[5], a versatile web-based tool that addresses a variety of developer tooling needs. We briefly describe some use cases of NDN-Play below.

1. NDN-Play was originally conceived as an educational tool, allowing users to simulate an NDN network entirely in the browser. This functionality allows the demonstration of NDN concepts such as Data multicast, security and Sync, running real NDNts [Jun24] code in a sandboxed environment. This playground allows users to define and dynamically modify the topology of the simulated network including parameters such as latency and the packet loss rate. The playground supports writing and running realistic NDN applications in the user's browser, bundling advanced functionality such as code autocompletion.

2. Applications may use complex TLV structures for encoding data. NDN-Play allows the visualization of encoded TLV blobs, displaying the internal structure of the encoded blocks. Users may use a meta language based on TypeScript to define complex TLV typing systems, aiding and improving this visualization. NDN-Play also includes a

[5]https://github.com/pulsejet/ndn-play

web-based visualizer for the VerSec [Nic21] trust schema language. This functionality is also provided in the form of a Visual Studio Code extension based on NDN-Play.

3. For debugging web-based applications, the NDN-Play browser extension provides the ability to capture and visualize NDN traffic from a given browser tab. This functionality is provided as a browser console tool that snoops on the application's WebSocket transport that is connected to the NDN forwarder. This extension also allows capturing packet traces that can later be loaded into the web interface for further analysis.

4. NDN-Play can also be used as a graphical interface for MiniNDN [Min25]. In this mode, MiniNDN emulates an NDN network using Linux namespaces, which can run any NDN software in a distributed setting. NDN-Play can then be used to interactively update parameters in the network and observe application behavior. To ease debugging, NDN-Play also provides interactive consoles on one or more emulated nodes of the developer's choice, which can be treated and used like separate virtual machines for the purposes of application development.

We continue to develop and expand the capabilities of NDN-Play. In the future, we hope to build more advanced visualization and debugging capabilities to assist developers, such as the ability to visualize data name trees and their security relationships, and to monitor data flows in NDN applications.

CHAPTER 7

Conclusion

Developing decentralized Internet applications encounters several fundamental conflicts with the existing TCP/IP architecture, stemming from factors such as the lack of user identities for communication and the development of security solutions, which leads to the inability of users to communicate directly, and the need for overlay infrastructures to support efficient data dissemination. Named Data Networking shows the promise of being the next-generation platform for building decentralized applications effectively.

We presented several insights on the fundamental reasons leading to the centralization of the Internet and advances towards our goal of enabling decentralized applications. We designed State Vector Sync, a new efficient and resilient transport protocol for NDN applications. SVS uses raw state encoding and dataset state change notifications to achieve better performance compared to previous Sync protocol designs. We also designed ndn-dv, a new distance vector routing protocol for named data networks, and demonstrated how the stateful data plane of NDN can make intelligent forwarding decisions to improve performance.

Building on previous research results, we designed and implemented Ownly, a new decentralized collaborative application that utilizes NDN networking and security primitives. Ownly's design prioritizes usability, allowing end-users to benefit from NDN's security and Ownly's decentralized design without introducing new constraints. Our design of Ownly will continue to evolve, providing further insights into the design and requirements for building sustainable, decentralized applications of the future.

7.1 Insights into Internet Centralization

We summarize the insights gained from examining modern applications dominated by centralized service providers, alongside both past and emerging efforts toward decentralization.

1. The lack of support for secure peer-to-peer communication between end users on the Internet forces applications towards centralization. This state of the network can be attributed to the lack of a cloud-independent user identity, and the rise of restrictive middleboxes such as NATs and firewalls.

2. The point-to-point nature of TCP/IP and TLS is an improper fit for modern distributed applications, which are inherently multi-party in nature. This mismatch has led to the rise of content delivery networks to support these applications. Reliance on CDNs introduces new security concerns and reinforces centralization by primarily serving only paying enterprises.

3. Several existing efforts to develop decentralized applications operate within the constraints of the TCP/IP communication and security model, addressing all challenges at the application layer. This design approach significantly increases complexity and often leads to inevitable centralization of control across various components.

These observations underscore the necessity for a new design mindset when developing decentralized applications. Named Data Networking, with its data-centric approach, offers a more appropriate foundation for modern data-oriented systems.

7.2 Summary of Contributions

Beyond the conceptual insights, this work advances the design and implementation of key practical components to enable decentralized applications running over Named Data Networking. We summarize these contributions below.

State Vector Sync: The SVS protocol builds upon extensive research in distributed synchronization and IP multicast, incorporating several deliberate design choices. The use of raw state encoding enables fast and efficient reconciliation during loss recovery. By treating Sync Interests as pure notifications, SVS avoids the issue of multiple data replies encountered in earlier Sync protocols, and reduces the persistent state overhead caused by long-lived Interests in the network. Our evaluations demonstrate the enhanced performance of SVS, and we outline future directions for further improving its scalability and efficiency.

NDN Distance Vector Routing: ndn-dv represents a new comprehensive effort towards scalable name-based routing. Its design adopts a purely pull-based mechanism for disseminating routing updates, shifting responsibilities like poison reverse to the routers affected by them – where they belong. By separating prefix and router reachability for two-step forwarding and leveraging Sync to maintain the prefix table, the ndn-dv design improves scalability in name-based routing. Furthermore, it serves as a practical demonstration of NDN Sync's capabilities.

Ownly: The design and implementation of Ownly exemplify a real-world decentralized application built using core Named Data Networking primitives. Our approach leverages and adapts various NDN components developed over the years, and introduces a new perspective focused on end-user usability to guide future research directions. Ownly marks a concrete step toward realizing truly decentralized Internet applications that do not compromise on user experience.

7.3 Future Directions for Named Data Networking

The underlying technology of a network must be designed to support applications that use the network for communication. Therefore, to design an effective network, we must first understand what functionality applications desire. We strongly advocate for application-driven research as a guiding force in shaping the future of NDN and advancing the broader goal

of Internet decentralization in general. In these early stages of decentralization efforts, we also believe it is essential for researchers to develop usable applications whose usage will provide feedback on our understanding of the solution space, driving for further advancements towards the decentralization direction.

A comprehensive understanding of the economic models of future decentralized applications remains lacking. Such insights are crucial for promoting decentralization and illuminating the potential for novel application types that NDN may support. To address economic concerns related to the deployment of NDN infrastructure, we advocate for the broader adoption of NDN overlay, in a way similar to the initial TCP/IP rollout, where IP started as an overlay running on top of the existing telephone networks.

Finally, our experience shows that effective developer tooling and implementation usability play a crucial role in the adoption of any technology. We recommend focused improvements in the NDN ecosystem such as unifying software components, enhancing documentation, simplifying network configuration, and designing new primitives that prioritize enhancing application usability for end-users. These practical steps are essential towards making NDN a more accessible and widely adopted platform.

REFERENCES

[ABR18] Alex Afanasyev, Jeff Burke, Tamer Refaei, Lan Wang, Beichuan Zhang, and Lixia Zhang. "A Brief Introduction to Named Data Networking." In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pp. 1–6, 2018. https://doi.org/10.1109/MILCOM.2018.8599682.

[APD18] Hila Ben Abraham, Jyoti Parwatikar, John DeHart, Adam Drescher, and Patrick Crowley. "Decoupling Information and Connectivity via Information-Centric Transport." In *ICN 2018 - Proceedings of the 5th ACM Conference on Information-Centric Networking*, pp. 54–66, 2018. https://doi.org/10.1145/3267955.3267963.

[ARJ22] Waqas Ahmad, Aamir Rasool, Abdul Rehman Javed, Thar Baker, and Zunera Jalil. "Cyber Security in IoT-Based Cloud Computing: A Comprehensive Survey." *Electronics*, **11**(1), 2022.

[ASZ21] Alexander Afanasyev, Junxiao Shi, Beichuan Zhang, Lixia Zhang, Ilya Moiseenko, Yingdi Yu, Wentao Shang, Yanbiao Li, Spyridon Mastorakis, Yi Huang, Jerald Paul Abraham, Eric Newberry, Steve DiBenedetto, Chengyu Fan, Christos Papadopoulos, Davide Pesavento, Giulio Grassi, Giovanni Pau, Hang Zhang, Tian Song, Haowei Yuan, Hila Ben Abraham, Patrick Crowley, Syed Obaid Amin, Vince Lehman, Muktadir Chowdhury, and Lan Wang. "NFD Developer's Guide." Technical Report NDN-0021, Revision 11, NDN, February 2021. https://named-data.net/techreports.html.

[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 *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*, pp. 525–530, 2015. https://doi.org/10.1109/MASS.2015.59.

[Bel58] Richard Bellman. "On a routing problem." *Quarterly of Applied Mathematics*, **16**(1):87–90, 1958.

[BS15] Alireza Beikverdi and JooSeok Song. "Trend of centralization in Bitcoin's distributed network." In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 1–6, 2015. https://doi.org/10.1109/SNPD.2015.7176229.

[BSA24] Leonhard Balduf, Saidu Sokoto, Onur Ascigil, Gareth Tyson, Björn Scheuermann, Maciej Korczyński, Ignacio Castro, and Michal Król. "Looking AT the

Blue Skies of Bluesky." In *Proceedings of the 2024 ACM on Internet Measurement Conference*, IMC '24, p. 76–91, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3646547.3688407.

[CBR07] Ravi Chandra, Tony J. Bates, Yakov Rekhter, and Dave Katz. "Multiprotocol Extensions for BGP-4." RFC 4760, January 2007. https://www.rfc-editor.org/info/rfc4760.

[Cis23] Cisco. "Cisco Annual Internet Report (2018–2023)." White Paper, Cisco Systems, Inc., 2023. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[CK74] V. Cerf and R. Kahn. "A Protocol for Packet Network Intercommunication." *IEEE Transactions on Communications*, **22**(5):637–648, 1974.

[CT90] D. D. Clark and D. L. Tennenhouse. "Architectural considerations for a new generation of protocols." In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, SIGCOMM '90, p. 200–208, New York, NY, USA, 1990. Association for Computing Machinery. https://doi.org/10.1145/99508.99553.

[CV04] Luis Costa and Rolland Vida. "Multicast Listener Discovery Version 2 (MLDv2) for IPv6." RFC 3810, June 2004. https://www.rfc-editor.org/info/rfc3810.

[DAT22] Saurab Dulal, Nasir Ali, Adam Robert Thieme, Tianyuan Yu, Siqi Liu, Suravi Regmi, Lixia Zhang, and Lan Wang. "Building a secure mHealth data sharing infrastructure over NDN." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, ICN '22, p. 114–124, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3558091.

[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*, SIGCOMM '11, p. 218–229, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2018436.2018462.

[ES00] Carl Ellison and Bruce Schneier. "Ten Risks of PKI: What You Are Not Being Told About Public Key Infrastructure." *Computer Security Journal*, **XVI**(1):1–7, 2000.

[FAC15] Wenliang Fu, Hila Ben Abraham, and Patrick Crowley. "Synchronizing Namespaces with Invertible Bloom Filters." In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 123–134, 2015. https://doi.org/10.1109/ANCS.2015.7110126.

[Fen97] Bill Fenner. "Internet Group Management Protocol, Version 2." RFC 2236, November 1997. https://www.rfc-editor.org/info/rfc2236.

[FHH16] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)." RFC 7761, March 2016. https://www.rfc-editor.org/info/rfc7761.

[fia24] fiatjaf. "NIP-01: Basic protocol flow description.", 2024. https://github.com/nostr-protocol/nips/blob/master/01.md.

[FJL97] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." *IEEE/ACM Transactions on Networking*, **5**(6):784–803, 1997.

[For56] L. R. Ford. *Network Flow Theory.* RAND Corporation, Santa Monica, CA, 1956.

[GÉ18] GÉANT project. "GÉANT topology map.", 2018. accessed: 2025-05-15. https://network.geant.org/.

[Hof23] Paul E. Hoffman. "DNS Security Extensions (DNSSEC)." RFC 9364, February 2023. https://www.rfc-editor.org/info/rfc9364.

[HSW21] Yi Hu, Constantin Serban, Lan Wang, Alex Afanasyev, and Lixia Zhang. "PLI-Sync: Prefetch Loss-Insensitive Sync for NDN Group Streaming." In *ICC 2021 - IEEE International Conference on Communications*, pp. 1–6, 2021. https://doi.org/10.1109/ICC42927.2021.9500452.

[IRL25] UCLA IRL. "Ownly: A secure decentralized workspace that you own." https://github.com/pulsejet/ownly, 2025. Accessed: 2025-05-07.

[IT21] Jana Iyengar and Martin Thomson. "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021. https://www.rfc-editor.org/info/rfc9000.

[Jun24] Junxiao Shi. "NDNts: Named Data Networking libraries for the Modern Web.", 2024. https://yoursunny.com/p/NDNts/.

[KAH22] Farhan Ahmed Karim, Azana Hafizah Mohd Aman, Rosilah Hassan, Kashif Nisar, and Mueen Uddin. "Named Data Networking: A Survey on Routing Strategies." *IEEE Access*, **10**:90254–90270, 2022.

[KFG24] Martin Kleppmann, Paul Frazee, Jake Gold, Jay Graber, Daniel Holmgren, Devin Ivy, Jeromy Johnson, Bryan Newbold, and Jaz Volpert. "Bluesky and the AT Protocol: Usable Decentralized Social Media." In *Proceedings of the ACM Conext-2024 Workshop on the Decentralization of the Internet*, DIN '24, p. 1–7, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3694809.3700740.

[KS02] Angelos Keromytis and Jonathan Smith. "Requirements for Scalable Access Control and Security Management Architectures." *ACM Transactions on Internet Technology*, **7**, 06 2002.

[KWH19] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. "Local-first software: you own your data, in spite of the cloud." In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pp. 154–178, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3359591.3359737.

[LCC09] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. "A Brief History of the Internet." *SIGCOMM Computer Communication Review*, **39**(5):22–31, October 2009.

[LJD14] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. "When HTTPS Meets CDN: A Case of Authentication in Delegated Service." In *2014 IEEE Symposium on Security and Privacy*, pp. 67–82, 2014. https://doi.org/10.1109/SP.2014.12.

[LKM19] Tianxiang Li, Zhaoning Kong, Spyridon Mastorakis, and Lixia Zhang. "Distributed Dataset Synchronization in Disruptive Networks." In *16th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (IEEE MASS)*, p. 10. IEEE, November 2019.

[Ma24] Xinyu Ma. *Towards Decentralized Applications*. University of California, Los Angeles, 2024.

[Mal98] Gary S. Malkin. "RIP Version 2." RFC 2453, November 1998. https://www.rfc-editor.org/info/rfc2453.

[Mas25] Mastodon gGmbH. "Your self-hosted, globally interconnected microblogging community." https://github.com/mastodon/mastodon, 2025. Accessed: 2025-05-05.

[MAZ22] Xinyu Ma, Alexander Afanasyev, and Lixia Zhang. "A Type-Theoretic Model on NDN-TLV Encoding." In *Proceedings of the 9th ACM Conference on*

Information-Centric Networking, ICN '22, p. 91–102, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3558093.

[MBH09] Joseph P. Macker, Carsten Bormann, Mark J. Handley, and Brian Adamson. "NACK-Oriented Reliable Multicast (NORM) Transport Protocol." RFC 5740, November 2009. https://www.rfc-editor.org/info/rfc5740.

[Min25] Mini-NDN Authors. "Mini-NDN: A Mininet-based NDN emulator.", 2025. accessed: 2025-05-15. https://minindn.memphis.edu/.

[MJ16] Parth Mannan and T. Jayavignesh. "Alternate simplistic approach to solve count-to-infinity problem by introducing a new flag in the routing table." In *2016 Second International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pp. 291–295, 2016. https://doi.org/10.1109/ICRCICN.2016.7813673.

[MKC23] Alex Murray, Dennie Kim, and Jordan Combs. "The promise of a decentralized internet: What is Web3 and how can firms prepare?" *Business Horizons*, **66**(2):191–202, 2023.

[Mos14] Marc Mosko. "CCNx 1.0 Collection Synchronization." Technical report, Palo Alto Research Center, Apr 2014. https://doi.org/10.13140/RG.2.1.4510.1925.

[Moy94] John Moy. "Multicast Extensions to OSPF." RFC 1584, March 1994. https://www.rfc-editor.org/info/rfc1584.

[Moy98] John Moy. "OSPF Version 2." RFC 2328, April 1998. https://www.rfc-editor.org/info/rfc2328.

[MPA23] Philipp Moll, Varun Patil, Alex Afanasyev, Junxiao Shi, and Lixia Zhang. "Lessons Learned from Fixing the Dead Nonce List in NFD." Technical Report NDN-0077, Revision 1, NDN, 2023.

[MPS21] Philipp Moll, Varun Patil, Nishant Sabharwal, and Lixia Zhang. "A Brief Introduction to State Vector Sync." Technical report, NDN, 2021.

[MPW22] Philipp Moll, Varun Patil, Lan Wang, and Lixia Zhang. "SoK: The evolution of distributed dataset synchronization solutions in NDN." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, ICN '22, p. 33–44, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3558092.

[MPZ21] Philipp Moll, Varun Patil, Lixia Zhang, and Davide Pesavento. "Resilient Brokerless Publish-Subscribe over NDN." In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, p. 438–444. IEEE Press, 2021. https://doi.org/10.1109/MILCOM52596.2021.9652885.

[Nag01] S.V. Nagaraj. "Access Control in Distributed Object Systems: Problems With Access Control Lists." In *Proceedings Tenth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE 2001*, pp. 163–164, 2001. https://doi.org/10.1109/ENABL.2001.953407.

[Nam25a] Named Data Networking. "NDN Packet Specification.", 2025. https://named-data.net/doc/NDN-packet-spec/current/.

[Nam25b] Named Data Networking (NDN) Project. "ndn-svs: State Vector Sync (SVS) library for NDN." GitHub repository, 2025. Accessed: 2025-04-26. https://github.com/named-data/ndn-svs.

[Nam25c] Named Data Networking Project. "Containerized NDN testbed deployment.", 2025. accessed: 2025-05-08. https://github.com/named-data/testbed.

[Nam25d] Named Data Networking Project. "Named Data Networking Daemon (ndnd).", 2025. Accessed: 2025-04-23. https://github.com/named-data/ndnd.

[Nam25e] Named Data Networking Project. "NDN Packet Format Specification version 0.3: Type-Length-Value (TLV) Encoding.", 2025. accessed: 2025-05-15. https://docs.named-data.net/NDN-packet-spec/current/tlv.html.

[Nam25f] Named Data Networking Project. "NDN Testbed." https://named-data.net/ndn-testbed/, May 2025. Accessed: 2025-05-06.

[Nam25g] Named Data Networking Project. "NDN Testbed Status.", 2025. accessed: 2025-05-08. https://testbed-status.named-data.net/.

[Nam25h] Named Data Networking Project. "Spec and API description of the StateVectorSync (SVS).", 2025. https://named-data.github.io/StateVectorSync/.

[Nic19] Kathleen Nichols. "Lessons Learned Building a Secure Network Measurement Framework Using Basic NDN." In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ICN '19, p. 112–122, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3357150.3357397.

[Nic21] Kathleen Nichols. "Trust Schemas and ICN: Key to Secure Home IoT." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN '21, p. 95–106, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3460417.3482972.

[NMZ21] Eric Newberry, Xinyu Ma, and Lixia Zhang. "YaNFD: Yet another Named Data Networking Forwarding Daemon." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN '21, p. 30–41, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3460417.3482969.

[Not23] Mark Nottingham. "Centralization, Decentralization, and Internet Standards." RFC 9518, December 2023. https://www.rfc-editor.org/info/rfc9518.

[Par20] Chang-Seop Park. "Security Architecture for Secure Multicast CoAP Applications." *IEEE Internet of Things Journal*, **7**(4):3441–3452, 2020.

[PDZ22] Varun Patil, Hemil Desai, and Lixia Zhang. "Kua: A Distributed Object Store over Named Data Networking." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, ICN '22, p. 56–66, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3558083.

[PMZ21] Varun Patil, Philipp Moll, and Lixia Zhang. "Supporting Pub/Sub over NDN Sync." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN '21, pp. 133–135, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3460417.3483376.

[Pos77] Jon Postel. "Comments on Internet Protocols and TCP.", August 1977. https://www.rfc-editor.org/ien/ien2.txt.

[Pos81a] Jon Postel. "Internet Protocol." RFC 791, September 1981. https://www.rfc-editor.org/info/rfc791.

[Pos81b] Jon Postel. "Transmission Control Protocol." RFC 793, September 1981. https://www.rfc-editor.org/info/rfc793.

[Pro12] ProjectCCNx. "CCNx Synchronization Protocol." CCNx 0.8.2 documentation, 2012. https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt.

[PSX22] Varun Patil, Sichen Song, Guorui Xiao, and Lixia Zhang. "Scaling State Vector Sync." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, ICN '22, p. 168–170, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3559485.

[RD08] Eric Rescorla and Tim Dierks. "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246, August 2008. https://www.rfc-editor.org/info/rfc5246.

[SAZ18] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. "VectorSync: Distributed Dataset Synchronization over Named Data Networking." Technical Report NDN-0056, NDN, March 2018.

[SCG01] Tony Speakman, Jon Crowcroft, Jim Gemmell, Dino Farinacci, Steven Lin, Dan Leshchiner, Michael Luby, Todd L. Montgomery, Luigi Rizzo, Alex Tweedly, Nidhi Bhaskar, Richard Edmonstone, Rajitha Sumanasekera, and Lorenzo Vicisano. "PGM Reliable Transport Protocol Specification." RFC 3208, December 2001. `https://www.rfc-editor.org/info/rfc3208`.

[SEF97] Puneet Sharma, Deborah Estrin, Sally Floyd, and Lixia Zhang. "Scalable Session Messages in SRM." Technical Report 98-670, UCLA, August 1997.

[SHC19] Nicolas Serrano, Hilda Hadan, and L. Jean Camp. "A Complete Study of P.K.I. (PKI's Known Incidents)." *SSRN Electronic Journal*, **TPRC**(47), 2019.

[SK05] Karen Seo and Stephen Kent. "Security Architecture for the Internet Protocol." RFC 4301, December 2005. `https://www.rfc-editor.org/info/rfc4301`.

[SMH16] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitrij Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. "Solid: A Platform for Decentralized Social Applications Based on Linked Data." Technical report, Massachusetts Institute of Technology (MIT) Computer Science and Artificial Intelligence Laboratory (CSAIL) and Qatar Computing Research Institute (QCRI), 2016.

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

[SS02] Andreas Schmid and Christoph Steigner. "Avoiding Counting to Infinity in Distance Vector Routing." *Telecommunication Systems*, **19**(3):497–514, Mar 2002.

[SYW17] Wentao Shang, Yingdi Yu, Lijing Wang, Alexander Afanasyev, and Lixia Zhang. "A Survey of Distributed Dataset Synchronization in Named Data Networking." Technical Report NDN-0053, NDN, May 2017.

[SYZ16] Klaus Schneider, Cheng Yi, Beichuan Zhang, and Lixia Zhang. "A Practical Congestion Control Scheme for Named Data Networking." In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ACM-ICN '16, p. 21–30, New York, NY, USA, 2016. Association for Computing Machinery. `https://doi.org/10.1145/2984356.2984369`.

[SZ22] Sichen Song and Lixia Zhang. "Effective NDN Congestion Control Based on Queue Size Feedback." In *Proceedings of the 9th ACM Conference on*

Information-Centric Networking, ICN '22, p. 11–21, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3517212.3558088.

[the25] thekinrar. "Mastodon instances.", 2025. https://instances.social/.

[WLH18] L. Wang, V. Lehman, A. K. M. Mahmudul Hoque, B. Zhang, Y. Yu, and L. Zhang. "A Secure Link State Routing Protocol for NDN." *IEEE Access*, **6**:10470–10482, jan 2018.

[WPD88] D. Waitzman, C. Partridge, and S. E. Deering. "Distance Vector Multicast Routing Protocol." RFC 1075, November 1988. https://www.rfc-editor.org/info/rfc1075.

[WTP24] Yiluo Wei, Dennis Trautwein, Yiannis Psaras, Ignacio Castro, Will Scott, Aravindh Raman, and Gareth Tyson. "The Eternal Tussle: Exploring the Role of Centralization in IPFS." In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 441–454, Santa Clara, CA, April 2024. USENIX Association. https://www.usenix.org/conference/nsdi24/presentation/wei.

[WZP02] Lan Wang, Xiaoliang Zhao, Dan Pei, Randy Bush, Daniel Massey, Allison Mankin, S. Felix Wu, and Lixia Zhang. "Observation and analysis of BGP behavior under stress." In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, New York, NY, USA, 2002. Association for Computing Machinery. https://doi.org/10.1145/637201.637231.

[XZL18] Xin Xu, Haitao Zhang, Tianxiang Li, and Lixia Zhang. "Achieving Resilient Data Availability in Wireless Sensor Networks." In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, 2018. https://doi.org/10.1109/ICCW.2018.8403581.

[YAA14] Cheng Yi, Jerald Abraham, Alexander Afanasyev, Lan Wang, Beichuan Zhang, and Lixia Zhang. "On the Role of Routing in Named Data Networking." In *Proceedings of the 1st ACM Conference on Information-Centric Networking*, p. 27–36, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2660129.2660140.

[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 Conference on Information-Centric Networking*, ACM-ICN '15, p. 177–186, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2810156.2810170.

[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, 2013.

[YKM24] Tianyuan Yu, Zhaoning Kong, Xinyu Ma, Lan Wang, and Lixia Zhang. "PythonRepo: Persistent In-Network Storage for Named Data Networking." In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pp. 927–931, 2024. https://doi.org/10.1109/ICNC59896.2024.10556243.

[YMX23a] Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Xiaohua Jia, and Lixia Zhang. "On the Security Bootstrapping in Named Data Networking.", 2023. https://arxiv.org/abs/2308.06490.

[YMX23b] Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Yekta Kocaoğullar, and Lixia Zhang. "A New API in Support of NDN Trust Schema." In *Proceedings of the 10th ACM Conference on Information-Centric Networking*, ACM ICN '23, p. 46–54, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3623565.3623709.

[YZM24] Tianyuan Yu, Jacob Zhi, Xinyu Ma, Yekta Kocaogullar, Varun Patil, Ryuji Wakikawa, and Lixia Zhang. "Repo: Application Agnostic and Oblivious In-Network Data Store." In *2024 IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom)*, pp. 279–284, 2024. https://doi.org/10.1109/MetaCom62920.2024.00052.

[ZA13] Zhenkai Zhu and Alexander Afanasyev. "Let's ChronoSync: Decentralized Dataset State Synchronization in Named Data Networking." In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10, 2013. https://doi.org/10.1109/ICNP.2013.6733578.

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

[ZAZ17] Zhiyi Zhang, Alexander Afanasyev, and Lixia Zhang. "NDNCERT: Universal Usable Trust Management for NDN." In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pp. 178–179, September 2017. https://doi.org/10.1145/3125719.3132090.

[ZEB10] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D. Thornton, Diana K. Smetters, Beichuan Zhang, Gene Tsudik, kc claffy, Dmitri Krioukov, Dan Massey, Christos Papadopoulos, Tarek Abdelzaher, Lan Wang, Patrick

Crowley, and Edmund Yeh. "Named Data Networking (NDN) Project." Technical Report NDN-0001, Named Data Networking Project, October 2010.

[Zha20] Zhaoning Kong. "ndn-python-repo.", 2020. `https://ndn-python-repo.readthedocs.io/en/latest/`.

[ZLW17] Minsheng Zhang, Vince Lehman, and Lan Wang. "Scalable Name-based Data Synchronization for Named Data Networking." In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, 2017. `https://doi.org/10.1109/INFOCOM.2017.8057193`.

[ZMK15] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Nikhil Goyal, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. "The Cloud is Not Enough: Saving IoT from the Cloud." In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'15, p. 21, USA, 2015. USENIX Association.

[ZMM20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. "DECO: Liberating Web Data Using Decentralized Oracles for TLS." In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, p. 1919–1938, New York, NY, USA, 2020. Association for Computing Machinery. `https://doi.org/10.1145/3372297.3417239`.

[ZSR20] Behrouz Zolfaghari, Gautam Srivastava, Swapnoneel Roy, Hamid R. Nemati, Fatemeh Afghah, Takeshi Koshiba, Abolfazl Razi, Khodakhast Bibak, Pinaki Mitra, and Brijesh Kumar Rai. "Content Delivery Networks: State of the Art, Trends, and Future Roadmap." *ACM Comput. Surv.*, **53**(2), April 2020.

[ZYR18] Zhiyi Zhang, Yingdi Yu, Sanjeev Kaushik Ramani, Alex Afanasyev, and Lixia Zhang. "NAC: Automating Access Control via Named Data." In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pp. 626–633, 2018. `https://doi.org/10.1109/MILCOM.2018.8599774`.

[ZYZ18] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. "An Overview of Security Support in Named Data Networking." *IEEE Communications Magazine*, **56**(11):62–68, 2018.