UNIVERSITY OF CALIFORNIA

Los Angeles

Towards Decentralized Applications

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Xinyu Ma

2024

ABSTRACT OF THE DISSERTATION

Towards Decentralized Applications

by

Xinyu Ma
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Lixia Zhang, Chair

Internet consolidation has become a major concern. Centralization causes concern among users, especially social platform users, about privacy, service failure, and censorship. In recent years, vast efforts have been put into Internet decentralization, including new protocol designs (such as ActivityPub and Nostr), new platforms (such as IPFS), and the development of decentralized applications (such as Mastodon and Bluesky). Majority of these efforts take the existing TCP/IP network protocol stack as given, and build new solutions as an overlay of it. As for security, some of them keep the existing server-centric security model, but try to increase the number of servers; some utilize cryptographic keys, and distributes signed contents over existing web. An exception is the development efforts of decentralization built on Named Data Networking (NDN), which is a new way to networking, by naming and securing data directly, enabling end users to exchange secured data by names. This new way of networking removes dependencies on centralized servers.

In this dissertation, I find that the fundamental issue with centralization is the centralization of control power, i.e., cloud servers control the user identities and host the authentic versions of user-generated data. In this case, users lost the ability to securely communicate with each other without depending on the cloud. I analyze decentralized applications, especially those with a large amount of user-generated data, such as social applications and collaborative applications. By inspecting the basic factors that led to server's centralized control power and the functionalities the cloud provides, I identify the two main tasks of decentralizing security control power — decentralized name space and data dissemination. I classify and analyze decentralized applications based on these two tasks.

Then, based on the lessons learnt, I present NDN Workspace, a decentralized collaborative editing application over NDN. I discuss the key design decisions I made in NDN Workspace, and demonstrate the prototype I implemented. I evaluate the effectiveness of NDN Workspace in terms of functionality and performance. At last, I present other research contributions I has made to advance the NDN ecosystem to support the deployment of NDN Workspace and decentralized applications.

The dissertation of Xinyu Ma is approved.

Jeffrey Aaron Burke

Jingsheng Jason Cong

Songwu Lu

George Varghese

Lixia Zhang, Committee Chair

University of California, Los Angeles

2024

“You asked ‘who is this who hides counsel without knowledge?’

Therefore I have uttered what I did not understand,

Things too wonderful for me, which I did not know.

Listen, please, and let me speak;

. . .

Therefore I abhor myself,

And repent in dust and ashes.”

— *Job*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Appreciate the mentorship given by my supervisor, Professor Lixia Zhang, whose wisdom and extraordinary experience in computer networking guided my path towards this degree. Thank my current and previous colleagues including Tianyuan Yu, Varun Patil and Junxiao Shi, who contributed a great deal in revising the design and implementing our software. Appreciate Zhiyi Zhang, Hongcheng Xie, Yu Guan, Yekta Kocaoğullar, Zhaoning Kong, Eric Newberry, Prof. Alexander Afanasyev, Prof. Jeff Burke, Prof. Phlipp Moll, and others, who collaborated with me on different research topics.

This dissertation consists of published works by the author in part or in whole, whose copyright are reserved by the original publisher. These works are

- \$ 4 is adopted from "Exploring the Design of Collaborative Applications via the Lens of NDN Workspace.", which is in the preceeding of IEEE MetaCom 2024.
- \$ 6.2 adopts part of "A New API in Support of NDN Trust Schema" (`https://dl.acm.org/doi/10.1145/3623565.3623709`).
- \$ 6.4 adopts "Sovereign: Self-contained smart home with data-centric network and security" (`https://doi.org/10.1109/JIOT.2022.3144980`) and "YaNFD: Yet another named data networking forwarding daemon" (`https://doi.org/10.1145/3460417.3482969`).
- \$ 6.6 adopts "PythonRepo: Persistent In-Network Storage for Named Data Networking" (`https://doi.org/10.1109/ICNC59896.2024.10556243`).

VITA

2013–2017	B.S. (Software Engineering), Beijing Institute of Technology.
2018–2018	Enrolled in M.S. program in Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University.
2019–2020	Teaching Assistant, Computer Science Department, UCLA. Taught sections of Computer Science 131 (programming languages) under direction of Professor Paul Eggert
2022–2024	Teaching Assistant, Computer Science Department, UCLA. Taught sections of Computer Science 118 (computer networking) under direction of Professor Lixia Zhang.
2018–2024	Graduate Student Researcher, Internet Research Lab, Computer Science Department, UCLA.

PUBLICATIONS

Zhiyi Zhang, Yu Guan, Xinyu Ma, and Lixia Zhang. “AuditShare: Sensitive Data Sharing with Reliable Leaker Identification.” *arXiv preprint arXiv:1907.11833*, 2019.

Xinyu Ma, Eric Newberry, and Lixia Zhang. “NDN Forwarder Manager: Improving the Usability of NDN Forwarders.” *Named Data Networking, Tech. Rep. NDN-0070*, 2021.

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*, pp. 30–41, 2021.

Zhiyi Zhang, Tianyuan Yu, Xinyu Ma, Yu Guan, Philipp Moll, and Lixia Zhang. "Sovereign: Self-Contained Smart Home with Data-Centric Network and Security." *IEEE Internet of Things Journal*, 9(15):13808–13822, 2022.

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*, pp. 91–102, 2022.

Tianyuan Yu, Hongcheng Xie, Siqi Liu, Xinyu Ma, Varun Patil, and Lixia Zhang. "CertRevoke: a Certificate Revocation Framework for Named Data Networking." In *Proceedings of the 9th ACM Conference on Information-Centric Networking,*, pp. 80–90, 2022.

Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Yekta Kocaoğullar, and Lixia Zhang. "Intertrust: Establishing Inter-Zone Trust Relationships." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pp. 180–182, 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 *ICC 2023-IEEE International Conference on Communications*, pp. 5091–5096. IEEE, 2023.

Tianyuan Yu, Xinyu Ma, Varun Patil, Yekta Kocaoğullar, 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.

CHAPTER 1

Introduction

The topic of decentralizing the Internet has been attracting increasing attention in recent years, and there have been a plethora of proposed solutions that go in various different directions. However, there has not been a commonly agreed upon definition of what is Internet centralization, and more importantly, what are the leading factors that have driven the Internet to today's highly consolidated state. A good understanding of these factors is critically important in deriving effective solutions to decentralize the Internet.

In this dissertation, without nailing down a specific definition of Internet centralization, we want to focus on the control power residing in user-generated data, which is a more fundamental question regarding Internet centralization compared to resource usage (traffic, storage, computation, etc.) If data are controlled by a single party, the economies of scale will drive resources to be centralized. Therefore, in this dissertation, we refer to *decentralization* by *decentralizing the control of user data*, and the fundamental goal of this dissertation research is to investigate solutions that enable *direct* user-to-user applications that are fully controlled by end users including all the resulting data, and that have no reliance on cloud servers for any critical functions, including naming, security, and rendezvous, which are hallmarks of today's most popular Internet applications.

To be more specific, we are developing solutions to the following questions:

- How can users obtain names that are independent of a specific platform or application provider?

- How can users establish secure peer-to-peer communication among themselves?
- Given the distributed nature of Internet applications in general, how can end users synchronize application state and data in the absence of servers?
- Given that user devices may go online and offline all the time, how can user data be made available all the time to support asynchronous communication among users, without reliance on specific cloud providers?

The rest of this dissertation is organized as follows. Chapter 2 analyzes several representative ongoing efforts towards decentralized applications, and summarizes the lessons we can learn from these efforts. Chapter 3 describes the design and implementation of *NDN Workspace*, a prototype application that supports direct user-to-user applications, without reliance on cloud services. Chapter 4 introduces the programming implementation of NDN Workspace and evaluates it in terms of functionality and performance. Chapter 5 gives discussion on related topics of NDN workspace, including the consistency model, limitations and future work. Chapter 6 presents other contributions the author has made to the overall ecosystem of NDN networking and applications, together with the lessons learnt from these efforts. Chapter 7 concludes this dissertation.

CHAPTER 2

Existing Solutions for Internet Decentralization

In this chapter, we first review several existing efforts in developing decentralization solutions, which are dominated by social networking applications. We then compare several representative design approaches of different types and summarize the lessons we extract from these efforts.

2.1 Problems With Centralization

Today's Internet applications heavily rely on cloud services in general. Collaborative applications, such as shared document editors (e.g. Google Doc, Overleaf), use cloud servers as a place to (i) connect users as a rendezvous point, (ii) perform security verification, and (iii) serve the authoritative copy of all user produced data, which enables asynchronous communication.

We recognize that today's observed application and service centralization is initiated by a few basic factors.

1. Due to the shortage of IPv4 address space, the majority of enterprise networks and end users no longer possess public IP addresses; instead they are behind Network Address Translators and not directly reachable.

2. The point-to-point, connection-based TCP/IP network model resulted in early adoption of client-server application model. In this model, clients and servers are not equal — a server has a public IP address and a DNS name, but it is difficult for a client to

obtain them.

3. The necessity of securing communications resulted in a quick fix of securing point-to-point connections between clients and servers. The setup of a secured connection always requires some verifiable out-of-band knowledge. More specifically, a certificate issued by a public CA in TLS.

4. The economy of scale resulted in favoring consolidation in mitigating all the above three factors: offering services by a small number of application platform providers, assigning servers with public IP addresses to make them reachable by all clients, assign DNS names, then cryptographic certificates to these servers to enable secure connections to them.

In today's Internet, it is difficult for clients to obtain credentials, and thus peer-to-peer client connections are not feasible. Therefore, clients have to connect to a rendezvous point for communication, which leads to (i). Also, in the setup of the secured channel, the identity of server is authenticated, but the client remains anonymous. As a result, the server has to authenticate the client in the application layer using some pre-shared knowledge (such as passwords). Only servers have this kind of knowledge, which leads to (ii). (iii) comes from the fact that clients are not always online. In the channel based model, data objects are typically named using URL, which contains the server's address, implying that the method to securely obtain the data is to connect to the server. When servers become necessary to the Internet, infrastructures are be built to reduce the cost of servers, which form clouds. Clouds make servers cheaper and more reliable, which further strengthens servers, and applications depend more on servers.

There are several issues with centralization. First, the whole system's security is fully controlled by the cloud servers. Since users have no means to verify each other and blindly trust the server to provide secured user data. Although there has been no evidence that servers modify user data or impersonate end users, they do possess the power to do so, and there is no established systematic auditing solutions in user. Second, given servers' control power over user data, they can mine user data for multiple types of financial gains (e.g. targeted advertisements, or even selling user data). This leads to concern about user privacy. Third, data availability relies on the connectivity to the cloud server, which means local communication channels cannot be securely used. For example, two persons in a disconnected airplane cannot securely use the in-airplane WiFi or Bluetooth to communicate.[1]

Therefore, it is important to let user regain the control over their own data. More specifically, a user should be able to prove his identity to other users without relying the server to enable decentralized security, and unauthorized modification of user data must be detected. [2]

In this dissertation , we define decentalization as decentralization of control power. To achieve this goal, there are two tasks in a decentralizaed application design:

1. Naming, *i.e.,* how to name and authenticate users and user data, which is related to aforementioned (ii) and (iii).
2. Data dissemination, which is related to aforementioned (i) and (iii).

Solving these two tasks can reduce the reliance on the cloud servers.

2.2 Categorization of Existing Efforts

Naming plays an important role in Internet-scale applications. A name serves three functionalities:

1. Assuring (user and data) name uniqueness. For example, two users with the same

[1]AirDrop and similar works allow users to transfer files, there is no systematic solution to a generic application.

[2]A user also needs to control the access of the data generated by him, but this is out of the scope of this dissertation.

nickname need to be distinguished by globally unique user names (if they communicate with others in a global scope).

2. Relationship of user and the data they produce via names. For example, a post on Threads is identified by its URL, which includes the name of its author. This naming design associates posts with their authors.

3. Security. If an application uses URLs as object names and uses HTTPS to access resources, then the security of the objects, i.e. that authenticity and confidentiality, depends on the Web certificate of the host the URL points to as well as the TLS connection between the user device and the web server. [3]

The above three points suggest that user names should well organized into *meaningful* structures, such as the hierarchical structure used in DNS and URL.

The importance of semantically meaningful names is also well recognized by applications developers, who expose names to non-technical users. Popular applications, such as Threads, Youtube, and Telegram, display the user names in visible places.

Given the important role names play in application designs, we categorize existing decentralized applications by the name space they use. We have identified three types

- *Semantic naming*, which organizes names into a hierarchical structure.
- *Flat naming*, which treats names as structure-less octet strings.
- *Hybrid naming*, where every entity has more than one names.

Figure 2.1 roughly shows the classification. Although not every application platform may fit into this chart, We emphasize that all applications need a shared namespace with Internet-

[3]Alternatively, if web objects are signed using JSON web signatures [JBS15] or HTTP message signatures [BRS24], then security of the objects will be decoupled from TLS, but instead coupled with securely obtaining the signing key via the key names.

scale, which is directly tied to a number of other functions, among them the most outstanding one is security solutions.

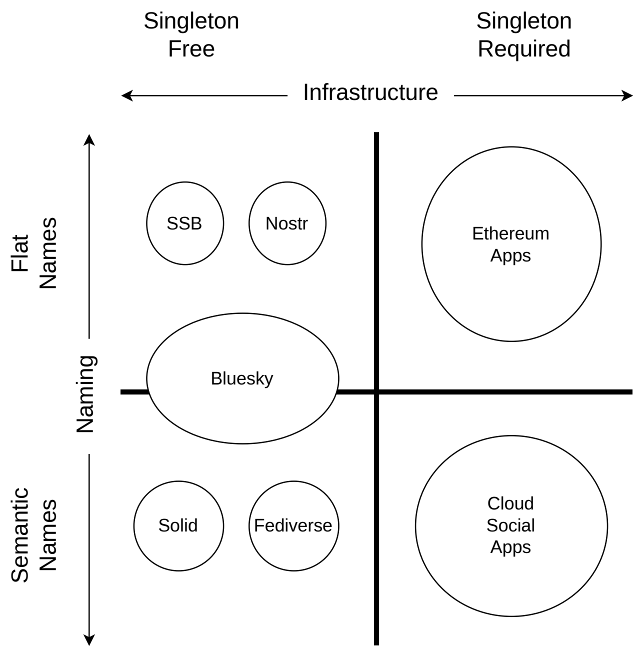


Figure 2.1: Coordinate of platforms

Offsets in a quadrant does not matter. Note that not every platform can fit into this chart.

2.3 Semantic Naming

As we mentioned earlier, we view human-recognizable names as bearing semantic meaning – from looking at a name, one is able to identify the specific party being named, Thus we call human-recognizable names as *semantic names*. In this section, we discuss two specific application platforms that make use of semantic names: the Fediverse platforms, and Solid.

These two designs affiliate each user with the server the user is attached to, and embed the server's host name, which are usually a DNS name, into the user's handle (i.e. the user's identifier). This design choice is not a coincidence. We articulate that there are at last two possible reasons for this approach to user naming.

1. The node-centric network model of TCP-IP has made heavy influence on people's view about networking: an interconnection of *nodes*, which can be either user computers and servers and are *identified by IP addresses* for packet delivery. Networking is meant to interconnect the nodes, thus the TCP/IP protocol architecture does not recognize users and applications, other than the nodes they are attached to, and the data that applications exchange through the network are also *invisible* to the network architecture.

2. The client-server asymmetry. All server nodes have DNS names and also security certificates issues to their names. On the other hand, most, if not all, user computers have no DNS names (and hence do not have certificates), resulting in client nodes being unable to securely communicate with other client nodes.

2.3.1 Fediverse

Fediverse is made of a collection of federated social platforms based on the ActivityPub [WT18] protocol, built upon the JSON-LD messaging format [WCL20]. Similar to the email application, ActivityPub assigns each user an inbox, and ActivityPub server instances deliver the publications of a user to his/her following users. Unlike email, the content of publication is structured in a standardized format, which allows applications with different functionalities (such as microblogging, image sharing, video streaming, etc.) communicate and share contents with each other, even when they have different data models[4]. W3C social web working group summarizes the goal of Fediverse platform as "include multiple servers sharing updates within a client-server architecture, and allow decentralized social systems to be built" [Guy17].

A Fediverse user's handle is in the format of `@user-handle@server-instance`, which

[4]Here, data models mean structured documents, exclude media and other binary files. The word "model" has the same connotation with the "modeling" in Object-Oriented Modeling.

affiliates users with its hosting server. The user handle is unique within its server instance, whose name is a DNS name. This formulation of user names makes them globally unique.

Fediverse users' accounts are completely controlled by Fediverse servers. Each user's profile, including the signing key, is managed by the hosting server. In many implementations (e.g. Mastodon and Misskey), the server's key is directly used as the signing key of users. That is, all users data are signed with the same key owned by the server. Therefore, a user's publications can only be authenticated through its attached server. In addition, all user generated data are hosted on the hosting server, and are only accessible via the server's URL. A compromised server can modify its user's posts without the knowledge of, let alone the authorization from, the user, and other users have no systematic methods to detect such modifications. If a user migrates from one server to another one, her previous posts will become inaccessible by her new/current name, unless (i) applications are aware of the user name changes and (ii) the original server keeps serving them under her previous handle. As a result, from users' perspective, although there can be multiple server instances, which are managed by different parties, to choose from, it is costly, if not infeasible, to migrate from one server to another.

Recent observations have shown that Fediverse exhibits an undesirable trend of moving towards server centralization. As shown in [RJC19], the largest Fediverse platform, Mastodon, there are user-driven, infrastructure-driven and content-driven pressures towards centralization. 10% of instances hosted almost half of users, and the removal of top 10 instances can lead to loss of 62.69% of posts. At the time this dissertation is written, the top 10 popular public server instances are hosting 56.93% of total public accounts, by the Mastodon instances website [the24b].

We believe that the most possible reason is the positive feedback loop of performance due to economy of scale. For example,

(A) Intra-server communication offers a better performance than federated inter-sever communication, attracting followers to move to the same server as the blogger they follow

(B) There exists a bit hurdle for a user to migrate to another server, creating an asymmetry between incoming and outgoing users.

(C) Each server must be online all the time, however keeping many servers online is neither desirable (consuming more resources) nor necessary (no computation-intensive or traffic-intensive). Therefore, the more users attach to a server, the less the operating cost per user.

(D) Posts published by remote users will not automatically show up on the timeline of a server, unless some local users follow them.

Here, (A) and (B) come from the federation model, and (C) and (D) are results of affiliating users with servers. To resolve this issue, it is necessary to empower the end users and break the binding between users and server instances (§ 2.6.2).

2.3.2 Solid

Solid is a decentralized data storage system [CBV22]. Solid lets individuals and groups store their data securely in decentralized data stores call *Pods*. Solid gives users more control over their data, by having users to control access to their data: data are restricted to the principals that the user grants access to. Solid decouples the applications and the data storage, and uses standardized data formats and protocols. Therefore, various applications can access shared datasets in an interoperable way.

There is a list of applications[5], but most of them seem experimental and not usable for non-technical users. Here, let me introduce several examples.

One Solid application currently deployed is the "My Citizen Profile" project collaborated with Flemish government [VVC22, Ber20]. In this project, each citizen is given a Solid Pod

[5]https://solidproject.org/apps.

as a digital vault storing personal information, such as diplomas. The users can decide which organization can access which piece of data and when. In this case, data are owned by a single user, accessed in a non-interactive way.

Another social application developped on Solid is Chat [BZ23]. As stated in the technical report, Solid prefers "a chat channel which stored data in dated subfolders much like directly writing a log of an IRC channel". The pod, owned by a single user, is the only authentic source of data living in the pod. This leads to a design that all chat data are stored in a single Pod of a user, no matter who the message sender is. In this sense, SolidChat is not a decentralized application.

There is also efforts combining Fediverse and Solid, such as Plume [Plu24], which is a Fediverse blogging application that supports using Solid as its storage. However, it is not actively maintained now.

At the time of the dissertation writing, Solid is yet to be widely adopted (e.g. million-user level) as public storage platforms. From the three examples above, we can observe that Solid itself only serves as a storage, with no protocol designed for collaboration. The decentralization of Solid also remains at the storage level: users can choose from a large amount of different storage servers (pod servers), but the data in a specific server is still solely controlled by the storage server itself. Solid itself does not solve the problem of communication between data owned by multiple parties. As a result, for a one-to-one application scenario such as information sharing ("My Citizen Profile"), Solid works smoothly, but when collaboration is needed, it will either goes back to the client-server model and treat a Pod as a storage server (Chat) or relies on other protocols to implement collaboration (Plume). This makes it hard to implement collaborative applications over Solid, and impedes the deployment of Solid.

Solid shares a similar issue with Fediverse. Solid uses a similar node-centric design for user and data naming, making data only fetchable from a specific Pod service. Although the access policies are written by the user, it is the Pod service who executes them, and there is no audit in place to detect misbehaviors of Pod servers. Users may self-host their pods or use federated pod providers. The experience of Mastodon shows that in practice, the majority of users will likely concentrate within a few of public pod providers [Gra21].

2.4 Flat and Node-Independent Naming

In this section, we discuss the efforts using flat names, which are node-independent, but carry no semantics. Typically, a flat name of user is the hash of his public key and a flat name of a message (or data object) is the hash of its content. Every message is required to be signed by its producer, and thus the integrity of message is secured intrinsically, *i.e.,* the message fetcher who knows the name can authenticates the message. Since users control the key, unauthorized modification can be detected. Due to lack of semantic relationships in the namespace, these protocols rely heavily on the links among messages, which leads to the preference of structured message formats. Also, most designs directly uses the cryptographic key bits or key hash as user names. Users thereby cannot rotate their keys, which means credential theft and credential loss will lead to permanent loss of accounts.

In some designs, there is a *singleton*, a single data structure or synchronization group that shares all information, existing in the system acting as a rendezvous point.[6] Whether there is a singleton in the system or not results in two design patterns.

2.4.1 Nostr and Scuttlebutt

Both Nostr [fia24] and Secure Scuttlebutt (SSB) [TLM19] use flat names and do not rely on a singleton. These systems follow a Pub/Sub model, *i.e.,* a peer can publish immutable records (called *events* in Nostr and *feeds* in SSB), and other peers can subscribe to the publisher and fetch the records. Records in both Nostr and SSB form an append-only log system and

[6]The word *singleton* is first proposed in [TLM19]. The original author describes it as "centralization aspects requireing consensus" or "single, privileged central authority". Since "authority" is also ambiguous, here we restate it in a different definition.

there is no built-in consensus algorithms. Without a fixed rendezvous point in design, how to communicate async becomes a key problem to solve. Therefore, they assume there exist relays on the network, which are always available and serve the data they received.

In the current version of Nostr, there is no method specifying how a peer selects a relay. From another peer's perspective, the publisher randomly pick relays, and the subscriber needs to connect to at least one relay of those selected by the publisher. Since there is no semantics in the record ID naming, it is hard to implement a scalable forwarding among the relays. As a result, users may tend to use the popular relays. In [WT24], the authors discovered that the top relay hosts 73% of the total posts. However, they also note that most (93%) posts are hosted over multiple relays, and 5% of posts are hosted by 178 relays. The author concludes that Nostr achieves availability with a very high storage overhead (average of 34.6 relays per post). On the contrary of high storage overhead, there lacks incentives for the relays to serve data. Paid relays can charge users for storing data, which is expected to be the norm in future [Jef24], but the user authentication protocol, the proof of storage, and payment methods are still under research. Especially, if users are only identified via semantic-less key hashes, it is very difficult to associate an anonymous user with a specific customer. If a blockchain based payment system is eventually used (like FileCoin), then the Nostr system will depend on a public blockchain to work.

Different from Nostr, SSB has a network layer based on the gossip protocol, which enables P2P communication without a relay. In SSB, clients and relays are equal peers, who can follow each other to form a following graph. When two peers connect, a peer will fetch all records generated by peers who are within three hops in a following graph. When to use a relay, a user and the relay will follow each other. Then, from the user's perspective, the relay becomes a one-hop following. Therefore, the user will fetch all records generated by (i) other peers followed by the same relay and (ii) peers directly followed by (i), *i.e.,* "friends of your neighbors". This creates a storage overhead to all peers including end users, and in the case when P2P connections cannot be used, the incentives of relays remain a problem.

2.4.2 Applications Over Public Blockchains

In public blockchains (or permissionless blockchains), users own the security key and their public keys are directly used as addresses. The developers of public domain typically believe privacy should be achieved by keeping public keys anonymous [Nak08]. With full anonymity, the public blockchain design achieves security (under its own definition) as follows:

1. *Immutable singleton data structure*: all records are stored on a global, immutable data structure, *i.e.,* the chain. A record becomes valid when it is accepted and locked into the data structure.

2. *Consensus via gating function*: consensus on the chain is determined by a competition of resources, which can be computation power (proof-of-work), storage (proof-of-storage), ownership over tokens (proof-of-stake), etc. When conflict happens, the branch with the most computation power wins and becomes accepted as the consensus, while other branches roll back. The gate function also raise the standard for a new peer to join the chain, reducing the chance of scamming and sybil attacks. Blockchains assume no single party owns more than half of the resource and thus controls the whole chain.

3. *Auditability via transparency*: all records are visible to the public. In blockchain, all operations on a blockchain are automatically executed by programs ("smart contracts"), whose code is also a public record. By auditing the chain, adversary programs and participants are naturally cast out by market slection.

However, there are several issues.

Trend of centralized operation. The first one is the trend of centralization. The anonymous gating function makes it difficult to join the chain as a full node, which drives most human users to use an existing node instead of joining the blockchain as a node.[7] Competition of nodes on resources and the high requirement of being a node make the economies of scale work: Currently, in major cryptoconcurrency blockchains, most nodes are running by a small number of exchanges. In this case, a small colluded group has the power to decide which record can get into the chain, and even modify the record history. Even without a monopoly party, the blockchain history is not absolutely immutable. For example, in 2016, Ethereum made a hard fork after an attack, indicating the influence power of the community over the chain.

Account lost and theft. As aforementioned, in a system using flat names, key lost can lead to permanent account lost. In a public blockchain system, the immutability and anonymity make things even worse — almost all security incidents are unrecoverable. Certik [Cer24] reported 751 security incidents leading to \$1,840,879,064 loss in 2023. Among them, there are 47 private key compromises that lead to a loss of \$880,892,924.

Cost of operation. In the scenario of social applications, the "vote on everything" model of blockchains is inefficient and unnecessarily expensive. In Ethereum-like blockchain-based decentralized applications (DApps), the global chain is used as a linear Pub/Sub platform. Since every record must be accepted to the global data structure, connectivity to the global chain is always required and local peer-to-peer communication is impossible. The resource requirement of publishing records to the chain sets a cost for every on-chain operation (called "gas fee"). This puts monetization a high priority of blockchain-based social apps. Due to the high cost, off-chain operations are sometimes unavoidable. For example, large chunks of data are typically stored off-chain, where the blockchain records only store content hashes. For example, Peepeth uses IPFS to store posts. The images of NFTs are also typically stored off-chain. This leads to further risks of availability.

[7]The official Go implementation for Ethereum recommends a qual-core processor, at least 16GB memory, 2TB storage, and 25Mbps Internet speed [The24a].

2.5 Bluesky

Bluesky is another federated social platform. Different from Fediverse's instance-attached naming or Nostr's flat naming, Bluesky uses decentralized identifiers (DIDs) [SLS22] to implement a mixed naming scheme. A Bluesky account is affiliated with a custodian server (like Fediverse), but the account's DID is independent from the server's name. A DID contains a method name field, specifying the method used to resolve this DID. In the current stage, Bluesky only supports DNS and *PLC (placeholder)*, a flat name system designed by the Bluesky team to support key rotation. In this section, we only consider Bluesky accounts using DID-PLC names.

An account is associated with three different identifiers: a handle, a DID, and a DID document with operation histories. The handle is a DNS name, corresponding to a TXT record in the DNS server, whose content is the user's DID. The DID is a flat name, consists of the hash of the initial DID document, and is considered to be a permanant identifier owned by the user. The DID is resolved by a PLC server to the latest DID document. Currently, there is only one central PLC server deployed. A DID document is a JSON document containing the account information, including the current DNS name, DID, custodian data server, and the public key bits. When one resolves one of the identifiers, he obtains all of them and make sure the resolution relations (the hande to the DID, DID to the DID document, the DID document to the handle) are correct. The full operation log of the DID document is stored on the PLC server for audit. Every record in the log contains a pointer to the prior record and is signed by a valid key stored in previous records. The chain traces back to the first record, which gives the initial DID document, whose hash is stored in the DID. This makes a DID self-certifying in auditing.

By using this multi-step name resolution design, Bluesky overcomes some drawbacks in the previous designs. It allows users to switch to a different custodian server easier, since the names are decoupled. It also allows key rotation to avoid the key leak issue existing in the flat name systems. Though every user needs to be affiliated with a custodian server, changing it is easier than Fediverse. Bluesky encourages users to self-host their custodian servers by displacing the services which benefits from scaling, such as data aggregation, discovery and searching, to an overlay called "big-world". However, the design is complicated and difficult to maintain. In the current design, the PLC server becomes a single point of failure, and is likely to have scalability issues in future. The Bluesky team is actively hoping to replace it with or evolve it into something less centralized [PBC24].

2.6 Lessons Learned from Existing Efforts

2.6.1 Naming and Security

Cryptographic keys are essential for security. However, using keys solely does not guarantee a secure system. The following factors are needed to build a secure system:

- A context or a domain, where all entities and relations are defined. This is typically provided by the application instances, like in Fediverse and today's cloud-based social applications. In blockchains, the context is provided by the trading history and the smart contract code running on it.
- Authentication, which identifies the subject of an operation (*i.e.,* the producer of data) in the context. This step depends on the cryptographic key. Given the fact that credential loss and thefts are frequent, keys should not be limited to a specific validity period, enabling rotation of keys. This means that a dynamic binding between the user[8] and the key is necessary.
- Authorization, which verifies the privilege of the subject and decides whether it has the access to a resource, or whether the desired operation is allowed. This step is done

[8]By [Pub94], the user can be an individual or a system process.

after the authentication and depends on the context the entity exists.

- Auditing, which records an immutable copy of history and reveals it when necessary.
- Other validity checking performed by the application, such as syntactic correctness, semantic correctness, and content validity.

Among the works we reviewed above, we figure out that flat naming is flawed in providing a dynamic binding beween names and keys. Majority of systems using flat naming we reviewd above disallows key rotation or recovery. Bluesky's DID-PLC is an exception, but it relies on a centralized resolver server to work. Especially, non-hierarchical names have potential scalability issues on resolution. Therefore, we believe that the use of flat names should be limited to immutable data packets. More specifically, users should not be named by flat names.

2.6.2 Empower End Users With Semantic Names

Affiliating users with server instances comes from the node-centric communication model, which is a cause to centralization. Using flat names harms key rotation, recovery and scalability. Therefore, to empower end users in a decentralized system, we suggest the following features

1. *Semantic namespace.* We suggest a decentalized system use a hierarchical namespace, where the name hierarchy comes from the semantic context, independent from node addresses. The semantic context should be defined by the application scenario, instead of specific instances.
2. *User-controlled accounts.* Users should have control over their accounts, *i.e.,* the user should be able to prove his own identity to another user without necessarily depending on a notary to present. This suggests that cryptographic keys must be used, and the private key must be owned solely by the user himself.

3. *Semantic bootstrapping.* Users should not be anonymous entities, as anonymity breaks semantic relations and increases the difficulty of authroization. Instead, new users need to go through a security bootstrapping process when joining the system. The bootstrapping starts from some existing out-of-band relationship. During bootstrapping, the new user installs necessary security credentials of the system, obtains a name, and propagates its own credentials to the system. When a credential incidents happens, the user can re-bootsrtap into the system to recover.

4. *Data-centric Security.* Data should be secured directly and available in multiple replications. Users should not depend on a secured channel to guarantee the security of data. Data authenticity should also be decoupled from data dissemination channels.

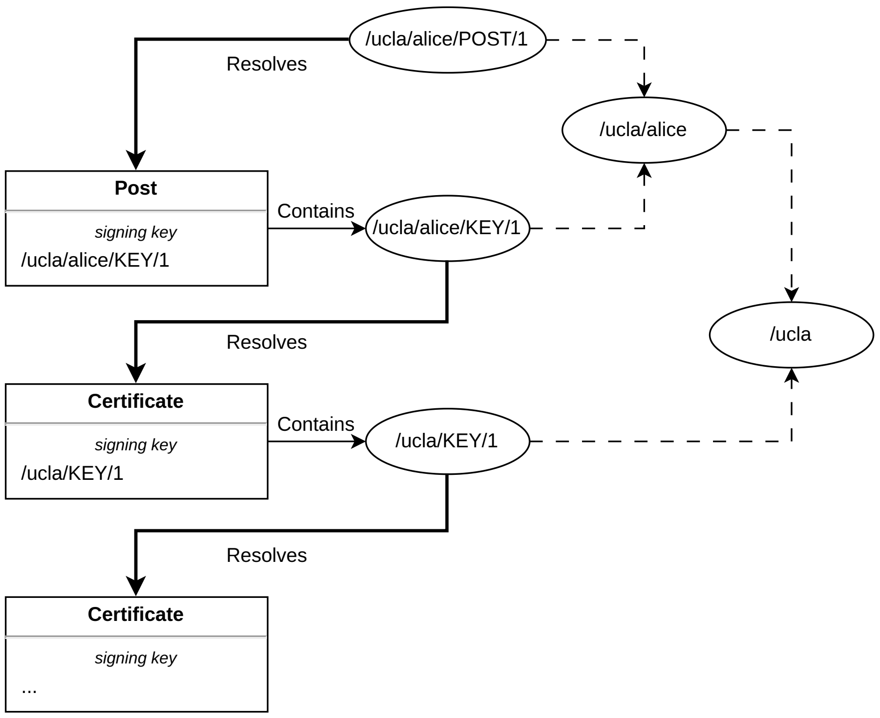


Figure 2.2: Names and security in semantic naming

A hierarchical, semantic namespace allows *naming conventions* to be used to assist data-centric security. Figure 2.2 shows such an example. Suppose Alice (“`/ucla/alice`”) is a

UCLA student, who posts a post named "`/ucla/alice/POST/1`". The post is signed by Alice's key and Alice's key is signed by the UCLA's key. If we have the naming convention that a key of an entity must be named as "`<entity>/KEY/<keyid>`", then the relationship names envolved in the signing chain is correspondingly fixed: "`/ucla/alice/POST/<postid>`" is signed by "`/ucla/alice/KEY/<keyid>`" and "`/ucla/alice/KEY/<keyid>`" is signed by "`/ucla /KEY/<keyid>`". Suppose the UCLA's key "`/ucla/alice/1`" is pre-installed to every device as an anchor of trust. Then, every device can verify the data it obtains using this knowledge. After it obtains the signed post, it extracts the signing key of the post, verifies the key name satisfies the naming convention, obtains the key, and verifies the signature. The same series of checks is applied to the obtained Alice's key. After all checks pass, the device is confident that the post obtained is generated by Alice. Bob from USC cannot forge this post, as his signing keys are not named in the form "`/ucla/alice/KEY/<keyid>`". Therefore, we achieve the same security goal of flat naming, that every device can securely verify everything it fetches, with the prerequisite that the naming convention and the trust anchor are distributed.

CHAPTER 3

NDN Workspace System Design

In the last chapter, we suggested that to empower end users, the application should let users own semantic names, secure data directly, and make data available in multiple replications. In this chapter, we discuss how these principles guided the design of a new decentralized collaborative editor application — NDN Workspace.

3.1 A Brief Introduction to Named-Data Networking

A network's essential function is to move data, which can be done in two ways: moving unnamed data to destination nodes, or moving named data to whoever requesting them. In either way, all communications must be secured.

3.1.1 A Quick Overview

Today's TCP/IP Internet model takes the first approach. Inheriting the notion of naming connection end points by phone numbers from telephone networks, the TCP/IP model views a network as made of interconnected nodes which are identified by IP addresses. Communication security is an afterthought and was patched on top of connections, e.g. running TLS on top of TCP.

Named Data Networking (NDN) [ZAB14], on the other hand, is designed to meet applications' needs. Applications publish and consume named data, therefore NDN names data instead of data containers. Naming data enables *securing data directly*, encrypting and signing data at their production time, to achieve confidentiality and authenticity.

NDN's network model views a network as made of *semantically named* entities with various *trust relations* among each other [YMZ21]. These named entities can be users, devices, services/app instances, or anything that produces and/or consumes named data. NDN uses application data names for network packet delivery, eliminating TCP/IP's need of mapping of application identifiers (e.g. DNS names) to IP addresses.

To communicate, given data cannot walk out on their own, NDN data consumers request data by sending *Interest* packets which carry the names of desired data. In response, the network returns the requested *Data* packets. A Data packet carries the matching semantic name, content and a cryptographic signature by its producer, which consumers use to authenticate the received packets to secure communications.

3.1.2 Interest and Data

NDN directly uses application data name at the network layer for routing and packet forwarding. Similar to URL, these names are semantically meaningful and hierarchically structured. For example, "`/room/temperature/sensor1/20240515`" can mean the temporature data taken by `sensor1` at May 15th, 2024, in the room `room`.

For a specific piece of data, the application which generates it is called *producer*, and the applications which fetch it are called *consumers*. A produces puts data generated into a *Data* packet, carrying its name, content, metainfo (used by the forwarder to handle the data) and a signature. When a consumer needs this piece of data, it sends out a packet called *Interest*, carrying either the full name or a prefix of the name. After the Interest is sent out, the network will eventually returns the desired Data back if it is available. A came prefix is used when a consumer does not know the exact name of the Data packet. In this case, any Data packet whose name contains this prefix can be matched and returned to the consumer.

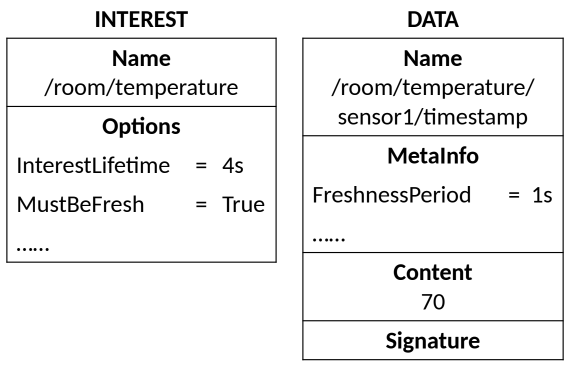


Figure 3.1: Interest and Data Packets

3.1.3 Secure Data-Centric Networking

To produce secured data and to verify received data, each communicating entity must be equipped with cryptographic credentials. When an entity E_{NDN} joins an NDN network, E_{NDN} goes through a bootstrap process, similar to that of an IP node E_{IP} going through when being connected to an IP network. Instead of obtaining IP address, router address, DNS server address etc that allow E_{IP} to send and receive IP packets, E_{NDN} obtains from the bootstrapping step its name, certificate, trust anchor, and security policies, which enable E_{NDN} to send secured data and cryptographically verify all received data.

The security policies capture the trust relations with other entities, and are defined by applications and expressed as a set of schematized trust rules, called *trust schema* [YAC15] and written in domain-specific languages [Nic21a, YMX23a]. They define which Data producer's key, which is identified by its semantic name, is allowed to sign the Data packets of given names, enabling applications to manage the trust policies with various entities. Fig. 3.2 depicts an example Data packet Alice produced under the namespace "`yourworkspaces.app/MeetRoom`". Data consumers define trust policies to only accept "`MeetRoom`" Data signed by the same producer referred in the Data name.

NDN treats *all* cyberspace objects, including app contents of all types (binary, image, video, etc.), cryptographic keys, and security policies, as semantically named and secure

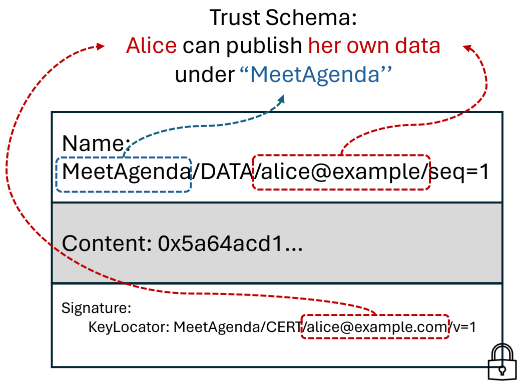

Trust Schema:
Alice can publish her own data
under "MeetAgenda"
Name:
MeetAgenda/DATA/alice@example/seq=1
Content: 0x5a64acd1...
Signature:
KeyLocator: MeetAgenda/CERT/alice@example.com/v=1

Figure 3.2: Alice's data packet

data. Therefore, crypto keys and trust schemas can all be fetched by their names in the same way as any other types of data.

3.1.4 Trust Domain and Trust Schema

As described in § 2.6.1, the cyber space is organized in semantic contexts where security is examined. Especially, human users act under autonomous organizations of various types and sizes, with trust relations of different degrees established between organizations. We consider named entities in a NDN network will reflect the nature of human societies, and thus form similar sturctures. Such an autonomous collection of named entities under the same administrative control is called a *trust domain* [Nic19, Nic21b][1]. Each trust domain has a *trust anchor*, which is a self-signed certificate acting as the root of all trust relations within this domain. The trust anchor's name reflects the name of the trust domain. *Trust domain controllers* are the authorities managing security policies and entities in the trust domain. Each trust domain must have at least a root controller, which is the governing entity of the trust domain who owns the trust anchor. Besides the root controller, there may be other controllers assigned to the trust domain to help bootstrapping new entities and define trust policies. The set of trust policies defined for a set of entities in a trust domain is called

[1]The original paper names it as a *trust zone*

a *trust schema.* Each entity in the trust domain must obtain the trust anchor, an NDN certficate, and the trust schema defined for the entity, before it can start communication. The process of obtaining necessary security information is part of the security bootstrapping process [YMX23b] of a new entity.

With respect to the level of trust described in § 2.6.1, the following tasks need to be done for a message to be trusted.

- *Bootstrapping*: to enroll a new member into an application. Deliver necessary security information to the new member through some out-of-band channel, and announce the new member to others.
- *Authentication*: to verify a piece of data is produced by claimed producer and not manipulated by an impersonator. This includes cryptographically verifying the signature.
- *Authorization*: to ensure a piece of data is produced by an intended producer, and accessed by an intended consumer, as defined in the trust schema. This includes discarding of data whose producers are not allowed to produce, and encrypting data so that only permitted consumers can decrypt.
- *Semantic Validation*: to make sure a record is corrected formed, valid in the system, and able to take effect. This is done by the specific application, and not related to the network layer.

Cryptographically verification belongs to *authentication* and trust schema verification belongs to *authorization.*

3.1.5 Dataset Synchronization

To enable consumers to fetch newly produced data promptly, especially when multiple producers contribute data to the same application, NDN enables consumers to learn the names of the latest data production by its transport service, Sync (short of synchronization), whose job is to synchronize the names of all the shared data among all participants in the same application group [MPW22]. Whenever a producer in a "Sync group" produces a piece of new data, Sync notifies all other participants about the data name; individual entities can then decide whether and when to fetch the desired data. An offline entity can catch up when it reconnects and learns about the names of new data it missed.

State Vector Sync (SVS) [MPS21] is the latest synchronization protocol in NDN. Based on SVS, a Pub/Sub framework named SVS-PS is designed to support general applications' needs on data distribution [PMZ21].

3.1.5.1 Brief Introduction to State Vector Sync and SVS-PS

State Vector Sync (SVS) [MPS21] is a dataset synchronization protocol. It uses sequential Data naming convention. The state of a sync group is described by a *Sync State Vector*, which contains the mapping between participant name and their latest sequence number. More specifically, each sync group is represented with a name prefix "`/<group>`", and suppose there are two participants: Alice with name "`/alice`" who published 5 records, and Bob "`/bob`" in the group who published 3 records. Then, the state vector will be "`[/alice:5,/bob:3]`" encoded in TLV. By specification [MPS21], the latest Data packets are named as "`/alice/<group>/seq=5`" and "`/bob/<group>/seq=3`" by default.[2]

SVS uses notification Interest to propagate the sync state. The notification Interest is sent both periodically and event driven. However, to avoid sync state injection, notification Interests are signed and therefore not able to be aggregated. To avoid too frequent sync Interest expression, an SVS participant enters a surpression state when it receives an outdated Interest from the remote. In the surpression state, it first waits for a random timer, and then sends an Interest with latest state if it has not received an up-to-date remote sync In-

[2]SVS allows the application to choose alternative naming conventions for Data packets.

terest. In the above example, the sync Interest is named "`/<group>/<param-digest>`", where the "`<param-digest>`" is the ParametersDigest component of NDN Interest, *i.e.,* the SHA256 digest of the signature and the Sync state vector.

SVS-PS [PMZ21] is a pub/sub service built upon SVS. In SVS-PS, a sequentially numbered SVS Data packet becomes a wrapper packet carrying several application Data packets or their Names. By doing so, applications can publish arbitrarily named records. If the Data packet is small, the application can also choose to put the whole Data packet instead of the name to save the time of an extra Interest-Data exchange.

3.2 NDN Workspace Design Overview

This section gives an overview of NDN Workspace design, and identify the necessary components to satisfy the design requirements

NDN Workspace is a collaborative application which is built into browsers to ease end user deployment. A group of users who want to jointly develop *a set of documents* form a *workspace instance*, in which they jointly edit shared documents through direct user-to-user data exchanges In this paper, we use "NDN Workspace" to mean the application we developed, and "workspace" for each specific running instance.

As Fig. 3.3 shows, each user obtains a *semantically meaningful* identifier within a workspace instance. Semantically identified users can bring their out-of-band trust relations into the application and endorse each other by issuing certificates to each other. NDN Workspace names data instead of data containers, thus it can utilize any and all available connectivity (*e.g.,* WiFi, Bluetooth, Cellular, etc.) to communicate. Following a data-centric approach, NDN Workspace names data by URI-like identifiers, which are independent from nodes (data containers) or communication channels. Every piece of data is encrypted and signed by its producer's key, which enables communications among users directly. Receivers can validate all incoming data based on application specific security policies defined by the users, so that

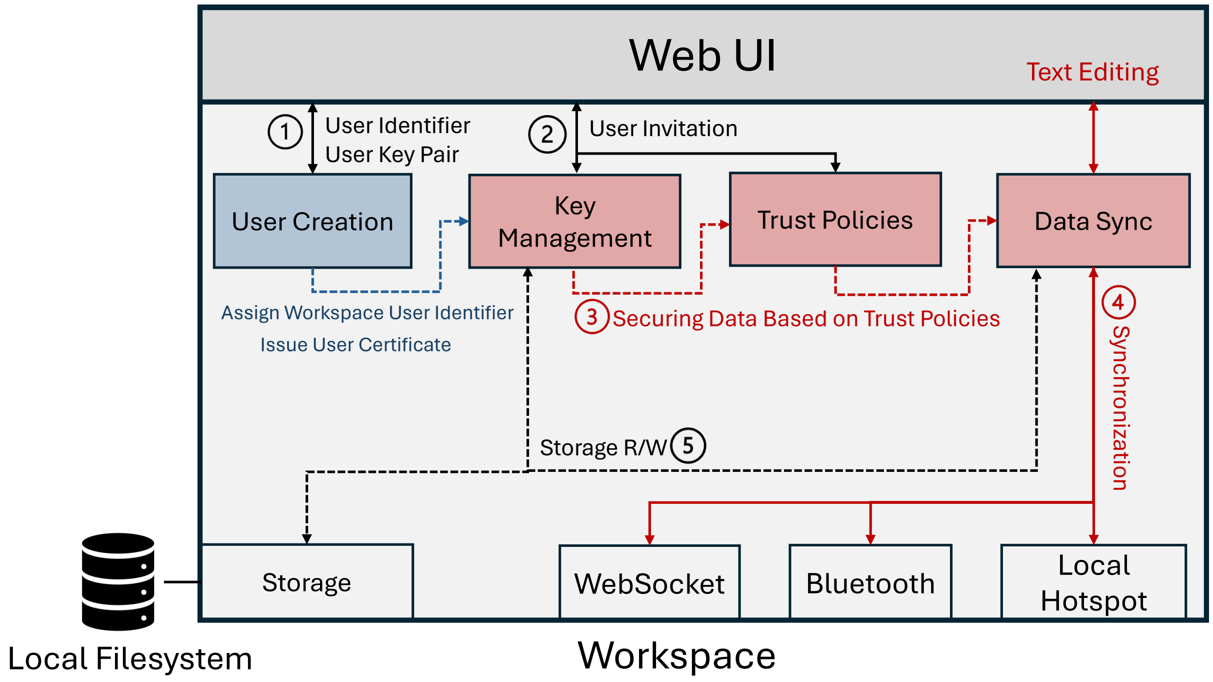


Figure 3.3: The NDN Workspace application architecture

each workspace only consumes data that its users' trust policy allows. Moreover, users can continue to work on shared files even when they are offline, i.e. without Internet connectivity. All the local changes a user U makes to the shared files will be synchronized when U gets connected with other users sharing the same workspace.

3.2.1 Semantic Identifier Is The Key

NDN Workspace secures data directly using user-defined security policies. Thus we need a systematic approach to specify the policies about *who* can produce data under *which* data namespace, and *who* can consume data under a given namespace. This requires assigning structured and semantically meaningful identifiers to users and data, so that security policies can be defined users can define using semantic, hierarchical namespaces. For example, a simple access control policy may state that only the user with identifier "`alice`" can send data under the identifier prefix "`/MeetRoom/alice`", where "`MeetRoom`" is the workspace. [3] Semantically identified and secured data can be exchanged over any physical connectivity, and saved at any place with storage space; consumers can verify all received data independent from data containers or transmission channels.

3.2.2 An Example Workflow

Suppose we have an workspace "`MeetRoom`" with users "`Alice`" and "`Bob`", containing the meeting agenda and notes. When Alice makes a change in a document "`agenda.txt`", the change in the text editor is first captured by the CRDT (conflict-free replicated data type, § 3.4.3) and encoded into delta format. Alice's NDN Workspace instance wraps this delta change into a Data packet, and signs it using Alice's certificate in the "`/MeetRoom`" trust domain. If this is her first change, the packet will be named "`/MeetRoom/alice/UPDATE /seq=1`". Afterwards, NDN Workspace stores generated Data packet into Alice's browser storage. If Alice is online, NDN Workspace will publish packet through the SVS-PS (§ 3.3).

Receives the published Data, Bob's instance validates Data by first verifying the Data signature using Alice's certificate (as indicated in the KeyLocator field), then executing trust policies to check if the signing relation conforms to trust rules. Validated Data is stored into Bob's browser storage. Afterwards, the delta update, the content of the received Data packet, is submitted to the CRDT module and the changes are applied to the local copy of the document "`agenda.txt`". Then, Bob's copy of "`agenda.txt`" becomes identical to Alice's.

[3] Due to the limitation of existing implementation, user names in NDN Workspace is under specific workspace instances, but this does not reflect. We think this limits the control power of user on his own data, and are actively exploring chances to decouple user names from application instances.

3.3 NDN Workspace over SVS-PS

We used SVS-PS for data synchornization of NDN Workspace. We built SyncAgent, a programming shim layer built upon SVS-PS.

3.3.1 SyncAgent: the shim layer over SVS-PS

Though SVS-PS already provides a Pub/Sub API for application usage, it leaves the key fetching mechanism and storage to the application. Therefore, in the development of NDN Workspace, we designed SyncAgent, a shim layer built upon SVS to support the application. SyncAgent is designed to provide the following functionalities

- **Demultiplexing on topics**. Manage multiple Pub/Sub topics in sync groups, and let different program components subscribe to different topics.
- **Delivery guarantee**. Based on the application's requirement, SyncAgent provides at-least-once delivery or latest-only delivery.
- **Persistent storage**. Save the sync state and Data into a persistent local storage, and load them when the program restarts.
- **Data segmentation**. Segment an application message into multiple multiple segments if it is large, and automatically reassemble them on reception.
- **Security verification**. SyncAgent does security verification against the trust schema for all packets before they are delivered to the application.

3.3.1.1 Sync deliveries

A sync delivery is a SVS instance with an independent namespace. There are two types of delivery: at-least-once and latest-only. The names come from message queues but may have different meanings. In SyncAgent, deliveries are underlying SVS pipes used by an agent.

At-Least-Once The at-least-once delivery guarantees every message is delivered and acknowledged at least once. The at-least-once delivery uses the return of the application callback as the signal of receiption. To justify the usage, suppose the following use case

1. The delivery receives an update message to a document.
2. The application parses the message and updates the document.
3. The application stores the updated document into the persistent storage.

Now, if the sync delivery stores its state (more specifically, SVS state vector) before (3) finishes, and the application crushes, there will be an inconsistency: when the application restarts, the loaded document does not contain the update, while the sync delivery will treat the message as delivered.

To prevent this from happening, the workflow must be modified to the following

1. The delivery receives an update message to a document.
2. The application parses the message and updates the document.
3. The application stores the updated document into the persistent storage.
4. The application gives the delivery an acknowledge.
5. The delivery stores the sync state into the persistent storage.

In SyncAgent, the application receives messages via async callbacks, and the acknowledge is represented by the resolution of the promise. The application is required to register the callback before SVS sync starts to operate. If there is any problem with the message, including validation failure and promise rejection of the callback, at-least-once Delivery will reset by storing the previous SVS state into the storage and disconnecting.

The at-least-once delivery does not guarantee the order.

Latest-Only The latest-only delivery only fetches and delivers the latest message only with respect to the SVS sequence number of each node. It is designed for real-time status update and close to at most once delivery in message brokers. The latest-only delivery only uses the persistent storage to store the state vector. Fetched data are only stored in a temporary storage.

3.3.1.2 Sync channels

A *channel* is a set of API to the upper layer, representing a specific way to fetch and store data. Current implementation of SyncAgent provides 3 channels: *update* for the update of shared documents, *blob* for immutable data blobs, and *status* for frequent but temporary online status such as awareness (*i.e.,* the cursor position).

update Represents reliable delivery designed for delta updates. The segmentation of updates depend on the size of the content. If the payload size fits in one data packet, SyncAgent will wrap it into a single packet and put it directly into the sync Data packet. If the payload is too large for a single NDN packet, SyncAgent will segment this payload into a segmented object and put the name into the sync Data packet. These two cases are corresponding to the two cases of SVS-PS, embedding Data prefixes and embedding the whole records. Update messages use the at-least-once delivery, and all packets are stored in a local persistent storage. In NDN Workspace, updates to shared documents and folders are published via the update channel.

blob Represents immutable large binary payload uploaded by users. Blob messages are always segmented, and the name is put into an SVS-PS packet. Blob messages also use the at-least-once delivery, and all packets are stored in a local persistent storage. In NDN Workspace, user uploaded non-text files are published via the blob channel.

status Represents frequent but temporary status updates. Status messages are always embedded into SVS packets. Status messages use the latest-only delivery and never stored into the local persistent storage. In NDN Workspace, user cursor status is published via the status channel.

3.4 Improve Data Availability

NDN Workspace works both offline and online. As data semantic and security are decoupled from communication channels, NDN Workspace users can store their data anywhere. This means NDN Workspace allows users to collaborate in real time when they're offline and also to make edits to their copy while offline. These edits are later synchronized when the user becomes online again. This setup leads to two requirements: (1) necessary data must be stored locally, and (2) application must support collaborations through asynchronous communications. To resolve editing conflicts from concurrent updates, NDN Workspace utilizes the Conflict-free Replicated Data Type (CRDT) data structures.

3.4.1 Persistent storage

For the first requirement, NDN Workspace uses browser's origin private file system (OPFS), which is a file-like storage persistent on the disk and it is private to the web application. The security properties for stored data, including signatures, are preserved by storing all data in its original byte form. For example, if Bob receives data produced by Alice, he can forward this change to Jane with the original signature. This means the original signature (by Alice) is still being independently verifiable.

3.4.2 Asynchronous communication using repo

The second requirement, supporting asynchronous communications, is needed as NDN Workspace makes no assumptions about when a user makes edits as well as the time frames when a user is online. In the fully asynchronous communication scenario where Alice makes update as the only "online" user, NDN Workspace ensures data availability for Bob and Charlie using an NDN repo (§ 6.6) inside the network. Repo is an in-network storage service that can join workspace Sync groups, fetch latest Data publication inside the group, thus store all document changes in-network. When the network receives an Interest asking for any Data that Repo already has, Repo directly replies the request with corresponding Data.

Additionally, Repo caches the latest incoming Sync Interests and it is aware of the latest state vectors in each workspace. Whenever Repo receives a Sync Interest with outdated state vectors, Repo will send cached latest Sync Interests to the network, so that remote parties can learn the latest status of the Sync group.

Repo ensures that even if Alice makes changes when neither Bob or Charlie is available, Repo caches Alice's Sync Interest and fetches Alice's Data. Later when Bob and Charlie become available and send Sync Interests to catch up with the latest status in the Sync group, they are informed with the state vector updated by Alice, and can fetch latest Data from Repo.

3.4.3 Conflict Resolution Using CRDT

NDN Workspace utilizes CRDT to implement real-time collaboration and resolve conflicts. Users' updates to shared documents and folders are captured by CRDT, wrapped into NDN Data packets, and then published via the `update` channel of SyncAgent (§ 3.3.1.2).

As shown in Fig. 3.4, we map workspace file structures to CRDT data types. We represent folders as *CRDT Maps*, which maintains a mapping between files and sub-structures. Text files are mapped to *CRDT Texts*, allowing real-time collaboration. Other types of files, which

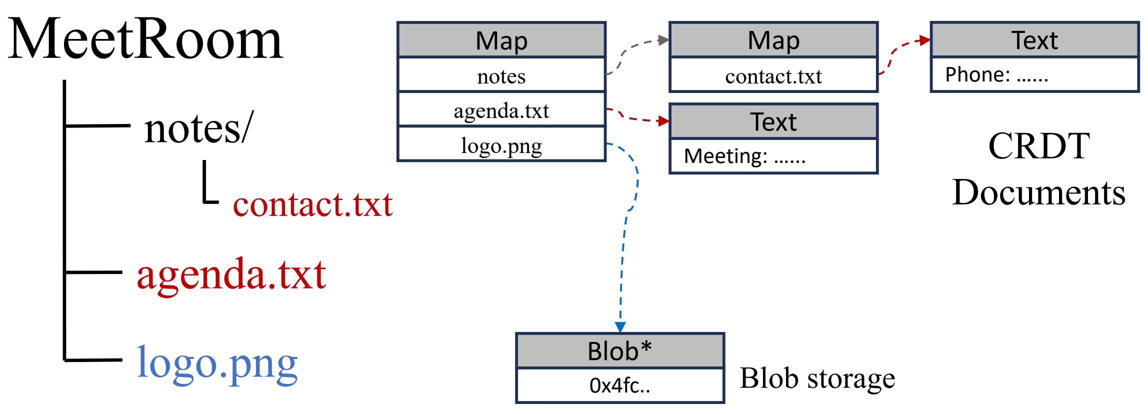


Figure 3.4: NDN Workspace represents file structure as a document, which is a collection of shared data structures.

do not support real-time collaboration, are transformed into immutable versioned, binary blobs, with CRDT only storing their data object names.

3.5 Achieve Security Goals

To embody the goal of user's control over their own data, we can set a concrete target, if users can use local communication channels to collaborate without access to any specific server. Users will be able to securely perform this task if they have the control. This section discusses how NDN Workspace makes this feasible.

3.5.1 Bootstrapping

NDN Workspace is an application designed to attached to a trust zone, and security bootstrapping is a process of the trust zone itself. Therefore, NDN Workspace does not put any special requirement nor design any new protocol for bootstrapping, but utilizes existing NDN bootstrapping designs and tools to perform this task.

As every NDN trust zone, NDN workspace requires the new member to be preshared *the* *trust anchor* before its participation in the workspace. To be recognized by other members, it needs to obtain a certificate signed by the trust anchor during bootstrapping. In the current deployment, NDN Workspace uses NDN Cert [ZAZ17] to manage the certificates, which requires the new participant to prove its identity through some *security challenges*, such as email code verification or Single-Sign On.

After the certifcate is issued, the new partcipant is accepted as a member of the workspace's trust zone, and is able to prove its legitimacy by presenting the certificate to other users.

3.5.2 Local communication

Before the communication starts, we assume all local participants have obtains the trust anchor and its certificate in the workspace via bootstrapping. In an isolated local environment, peers can setup connectivity by utilizing broadcast IP address, or setting up a shared rendezvous point. Since NDN uses node-independent Names, SVS and data fetching are both independent to specific nodes. Therefore, the NDN Workspace instance will not notice the difference in network environments. When a data is received, the consumer checks that the signature is signed by the producer's certificate (which can be fetched from the producer) and the certificate is issued by the trust anchor (which is shared knowledge). And data which passes the verification are secured.

3.6 Summary

Comparing NDN Workspace and cloud-based applications, we can see that with a semantic name space, the functionalities of cloud are provided by generic network components:

(i) Directly name and secure data using named-based security, instead of relying a cloud server for authentication and authorization.

(ii) Use SVS to synchronize user generated data sets, which enables making use of all connectivity channels, instead of relying a cloud-hosted rendezvous point.

(iii) Use NDN Repos, application-agnostic in-network storages, to store data for asynchronous communication, instead of application servers.

CHAPTER 4

Prototype Implementation and Evaluation

This chapter describes the implementation, deployment, and evaluation of NDN Workspace.

We evaluate NDN Workspace from both performance and functionality perspective to show NDN Workspace's effectiveness and uniqueness compared to existing applications.

4.1 Implementation and Deployment

We have implemented the NDN Workspace as a web application using SolidJS as the front-end framework.[1] We utilized Yjs [Jah24] as the CRDT implementation to implement real-time collaboration. We utilize NDN testbed as connectivity for NDN Workspace, and we use PythonRepo deployed on the NDN testbed as the online storage. In the last half a year, our researching team has actively used NDN Workspace as a progress tracker and a note taking tool for meetings.

4.2 Performance Evaluation

To evaluate the scalability and latency of NDN Workspace, we performed experiments to measure the delay for a user to receive a data publication. More specifically, we run multiple simulated users on the test machine, and let each simulated user randomly publish data carrying the timestamp of the publication. When another simulated user receives this

[1] The code is available at `https://github.com/UCLA-IRL/ndn-workspace-solid`

Publication Interval	**Number of Users per Testbed Router**			
	1 user	2 users	3 users	4 users
500 ms	192.55 ms	188.97 ms	190.17 ms	192.62 ms
1000 ms	192.32 ms	189.96 ms	189.66 ms	191.55 ms

Table 4.1: Average Latency in Testbed Experiments

publication, it logs the difference between reception and publication.

We conduct our experiment using an Ubuntu 22.04 machine with an AMD EPYC 7702P (64 core, 1.5 GHz) and 256 GB memory.We set the payload size of the publication to be 100 B. In the real-world scenario, the size of payload is dynamic, varying from several bytes (when a user is typing) to kilobytes (when a user pastes a large text block). However, as stated later, data publication frequency is low enough in shared text document editing, our designated scenario. Therefore, we can assume the payload will not affect performance significantly, as we always have enough bandwidth.

Data publication frequency also varies in different scenarios. In a shared document scenario, the frequency is no more than user typing speed, which is average 140ms per keystroke [DFK18]. There exists implementations[2] that combine multiple keystrokes into one to improve performance. NDN Workspace uses similar techniques to combine keystrokes, so that publication rate varies between per 500 ms and per 1000 ms. For simplicity, we evaluate the scenario where simulated users are connecting to several NDN routers on the testbed with data publication intervals 500 ms and 1000 ms.

In this experiment, all simulated users are in US and but in different geolocations. we pick four NDN Testbed nodes are their closest NDN router, with one in west US, two in central US and one in east US. We conducted four tests running one, two, three and four

[2]`https://github.com/overleaf/overleaf/blob/main/libraries/overleaf-editor-core/lib/ot_client.js`

users connecting to each Testbed router, *i.e.,* at most 16 users in total, which is reasonable user number of a collaborative project. The average is shown in Table. 4.1.

The average RTT of Testbed is 183 ms, which means the average latency is about 5% higher than the RTT. Since the Testbed itself is in an early stage, and also our software is not optimized for performance, we believe NDN Workspace design can be scaled to larger groups of users.

4.3 Functionality Evaluation

We are still in early stage of NDN Workspace trials, and yet to carry out systematic evaluation about NDN Workspace's functionality. As preliminary proof of evidence, in this subsection we first compare the functionalities with two typical text collaboration software in Table 4.2, then describe several anecdotes to show asynchronous collaboration, use of ad hoc connectivity, and the utility of NDN Workspace library in supporting other kinds of decentralized, collaborative applications.

Software	**Offline**	**Local Communication**	**Real-time Collaboration**
Overleaf[3]	✖	✖	✔
Git	✔	✖	✖
NDN Workspace	✔	✔	✔

Table 4.2: Functionality Comparison

Offline support: Though today's Internet coverage is high, there are still some cases when the Internet is not available. For example, when traveling on a plane, Internet access may be expensive. Git and NDN Workspace stores user data locally, so that user can edit shared documents when the Internet is not available or intermittent. Overleaf, on the other hand,

[3]We use Overleaf as a representative of cloud-based collaborative editors.

requires establishing connections to servers.

Local Communication: NDN Workspace does not only offer offline availability, but also allows users to collaborate locally by exploiting local connectivity. Since NDN is a network layer protocol which can run over all types of link layer protocols, NDN Workspace is able to make use of all available connections. For example, when users are on an airplane, they can use Bluetooth, or connect to a WiFi hotspot and use IP multicast. An enterprise can also provide local NDN node as a rendezvous point or a WebRTC server to setup peer-to-peer connections.

Real-time Collaboration: Traditional local software, such as Git, often leads to conflict when multiple users are editing the same file simultaneously. This kind of conflict has to be resolved manually in Git. Overleaf allows automatic conflict resolution using operational transformation (OT), but requires users to connect to the server. NDN Workspace, on the other hand, makes use of CRDT to resolve conflicts, enabling asynchronous real-time collaboration. Two users reach consensus state as long as they receive the same set of delta updates, regardless of the order of reception.

A1: Collaboration from Anywhere. We ran NDN Workspace in several events in the NDN community. We have deployed NDN Workspace instances for NDN Retreat 2023 and NDN Community Meeting 2024 and invited people to jointly edit files and share presentations with other onsite or remote attendees, through the NDN Testbed network. In February 2024, we also setup a NDN Workspace instance in our research group to logs our research progress, issue counter and reports through shared files. People edit this shared files independent from whether one is online or offline, and if offline, the changes get synchronized as soon as one gets connected.

A2: Exchanging Named Data over Any Connectivity. Also in February 2024, two members of the group attended a technical conference. During the long flight back home, the two jointly edited the NDN Workspace design document (hosted in our research group workspace instance) via one member's WiFi hotspot, which only has to Internet access. It shows that by securing data directly, NDN Workspace is able to make use any connectivity that can exchange named data.

A3: Developing Applications with NDN Workspace Libraries. During the development of NDN Workspace, we built up a set of libraries[4] for others to make use of NDN Workspace design and develop their own applications. A developer who is new to NDN Workspace design has implemented a simple decentralized chat app[5] in less than one week by making use of the NDN Workspace library.

[4]https://github.com/UCLA-IRL/ndnts-aux

[5]https://github.com/UCLA-IRL/ndn-workspace-solid/pull/75

CHAPTER 5

Discussion

5.1 Consistency and Availability with CRDT

NDN Workspace utilizes CRDT for automatic conflict resolution and inherits its consistency model (BASE). This section discusses the trade-offs behind the design of CRDT.

5.1.1 Consistency models

The CAP theorem [Bre00] states that strict consistency (C), high availability (A) and network partition tolerance (P) cannot coexist in a system. However, the definitions of these three terms are all in strict forms: strict consistency describes that every read operation must obtain the latest result, and high availability is defined that every read or write operation must succeed. In many cases, these definitions are too strict for the application scenario. For example, DNS does not satisfy the strict consistency nor the availability defined in the theorem, as it is possible for a DNS query to bring back no response or an out-of-dated cache entry. [Bre12] argues that the strict consistency in the CAP theorem is more strict than required by ACID (atomicity, cnsistency, insolation and durability). the consistency model used by relational databases.

Due to the rising requirement on high availability, BASE [Bre12] — basically available, soft state, and eventually consistent — is proposed as a new consistency model for NoSQL databases. In the new model, data are replicated across multiple nodes, so operations are served in best-effort when there is a partition (basically available). Read operations may result in stale data or local changes (soft state). In a partition, each connected component has its own local state, and there is no global consistency. Instead, when the partition ends and updates are propagated globally, the local states are required to merge into a new global consistent state (eventually consistent).

The usage of BASE depends on specific application scenario. Some application integrity rules may be violated when two local states merge into one after a partition. For example, there is no algorithm to merge two git commits with conflict editing while guaranteeing the program code compiles. Therefore, it is necessary to determine on a case-by-case basis based on specific application requirements. More specifically, to what degree the conflicts are automatically resolved, and what level of integrity is guaranteed in a collaborative application.

5.1.2 Collaboration workflows: git and CRDT

[KWH19] describes two workflows of collaboration: real-time collaboration and review-application. In the real-time collaboration, concurrent updates are applied automatically, while it does not guarantee a specific result. For example, if two users write the same section at the same time in a shared document, their changes will both occur in the merged result, but the order of edits are not guaranteed. Also, the same sentence, if written by both users, will ve duplicated. Google Doc and Overleaf belongs to this workflow. In the review-application workflow, conflict changes are first reviewed by a user, who is guided to make a decision. By doing so, human users intervene to ensure that application rules are not violated. For example, when using git, a merge conflict will be brought up to the user and a human resolution is required. GitSync 6.1 uses the review-application workflow.

Targeting high availability in real-time collaboration, Conflict-free Replicated Data Types (CRDT) are designed to automatically resolve conflict resolution created by potential network partitions. CRDTs are not a single data structure, but instead a collection of different data structures supporting different sets of operations. A majority (such as Yjs [NJD16] and Automerge [YZL23]) support shared text string, key-value map, incremental counter, XML rich text, and a composite of them. In CRDT, allowed operations always succeed locally, and the changes will be merged automatically after propagation without a central server or a leader node. CRDT typically use log sourcing to store the changes, *i.e.,* using an append-only log to record all changes, and use a merging algorithm to compute the result.

5.1.3 Trade-offs between availability and consistency

In conflict resolution, there is a trade-off between automaticity and semantic generality. The ability of automatic resolution requires the algorithm knows exactly the semantic of each operation, rather than general "read" and "write". Since each operation requires a different mechanism to resolve conflicts, the supported operations are limited. That is the reason why each CRDT has a specific set of supported operations, but do not support atomic remote procedure call (RPC) or customized transactions. If the application decomposes the complicated task into a series of supported operations, the automatic conflict resolution may result in semantic errors[1].

Human intervention The development and user experience research of Ink & Switch [KWH19] shows that users have an intuitive sense of human collaboration. They can coordinate in advance when editing a shared document to avoid concurrently editing. When a conflict happens, it is also feasible to leave it for users to fix.

Finer granularity A finer granularity in semantics can also avoid conflicts in semantics. For example, suppose two users concurrently create a section "Section 2" and insert some text into it. If the application is ignorant of the section structure of the document, *i.e.,* viewing the document as a text string and user operations as text insertion, then the result will be two separate "Section 2". However, if the application interprets the users operations

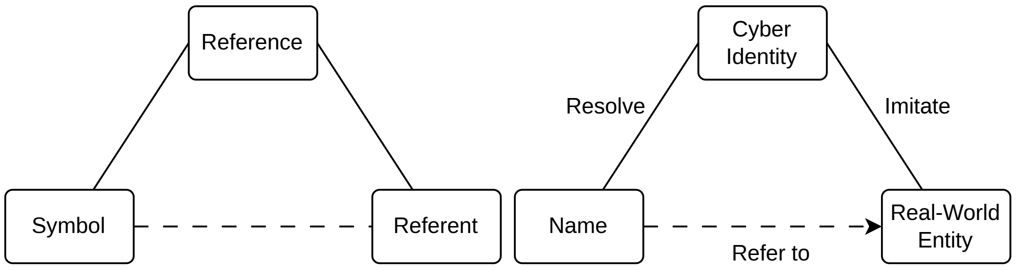
[1]Here, semantic errors refer to the conflict in content that could not happen if all concurrent operations are linearly applied by a single user. These are not programming bugs, but instead considered as features due to the algorithm limitation.

as "setting the title of Section 2" and "inserting text into Section 2", then there will be one merged section.

5.2 Meaning of "Semantics" in the Cyber World

There is a misunderstand that a hierarchical name is a sequence of human-readable strings and thus establishes a binding between the name and the real-world entity [GKR11]. This is not the case in a networking system, because the real-world entity is absent in and thus does not *directly* affect the cyberspace. Instead, a real-world entity needs a virtual representation that acts on its behalf.

To establish a refined model of meaning in networking, let me make an analog to a semiotic model in linguistics: the meaning triangle [OR27][2], which says that every binding consists of three objects: (a) *symbol*, (b) *reference* and (c) *referent* (Figure 5.1a). Here, the *referent* is the real-world entity the symbol refers to ("Things"), and the *reference* means the connotations about it, such as beliefs, values and experiences ("Thoughts"). It is the reference which is directed and organized, recorded and communicated.



(a) The meaning triangle of Ogden (b) The naming triangle in networking

Figure 5.1: The triangle of semantics

In the cyberspace, we draw a similar diagram consisting of (a) *name*, (b) *cyber entity* and (c) *real-world entity* (Figure 5.1b). The *name* is the identifier in the cyberspace. The

[2]Though typically called *the* meaning triangle, it has multiple versions in different literatures.

real-world entity is the real-world principal owning the name. The *cyber entity* is the digital simulacrum or representation of the entity, which essentially is the ensemble of social relations of the real-world entity cast into the cyberspace. The relation between name and the real-world entity is imputed, as it is impossible to directly bind a virtual thing with a physical essence. However, the other two relations are direct and solid: the name resolves to the simulacrum, and the simulacrum imitates the essence of the real-world entity. For example, an account in a social platform owns a username, and posted an article (Figure 5.2). Then, the account is the entity of the human-being behind it, and the post on the platform is the representation of the thoughts in the article. The fact that the account is the author of the post is stored in the platform's database. Therefore, we say the account represents the human being, because its relationship with the post reproduces the real-world relationship of the human-being and the article. The username can be resolved the username to the account by the social platform, but cannot be directly resolved to the human-being behind this account, since biological organisms are not digitalizable. Therefore, in the eyes of digital systems, there is no need for the username to be parsed to the human-being, *i.e.,* no need to be human-readable. Instead, the binding between the username and the account is beneficial: suppose we exploit a naming convention that all post names must contain the username as a prefix. The relationship of posts and the account is then expressed in the naming convention, and thus those database entries storing the authorship become unnecessary. If everything were perfectly expressed using such naming conventions, we would observe that the essence of the account become eroded by the username, as the username itself had already absorbed the social relationships and obtained the ability to reproduce the human-being. And that demonstrates the goal of using semantic names, *i.e.,* to reproduce the social relationships and absorb the connotations carried by cyber entities.

In summary, a more accurate statement of semantic names should be as follows: a semantic name is an identifier designed to carry part of the semantic relations of the corresponding cyber entity, expressing the context using hierarchical naming conventions.

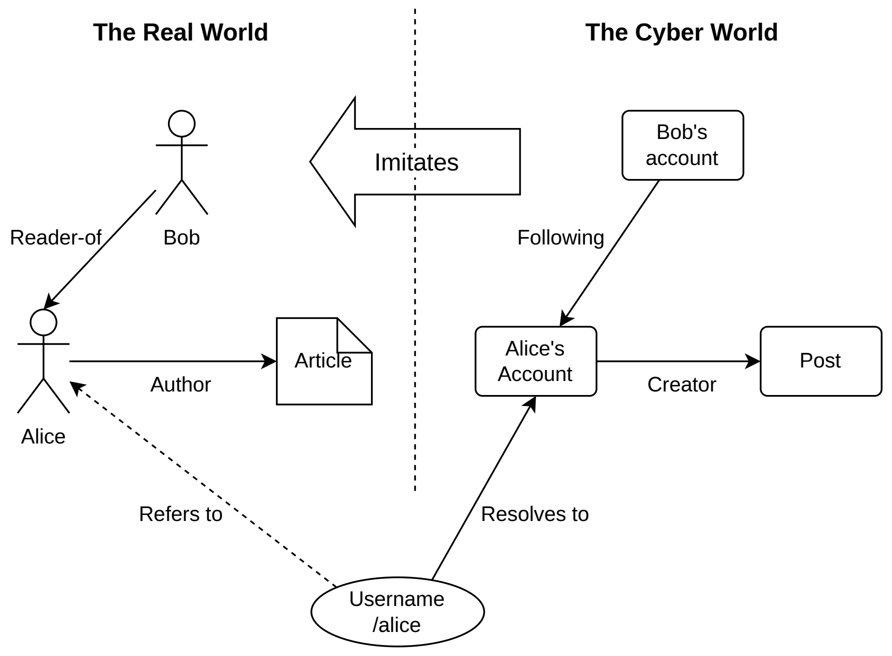


Figure 5.2: Semantics: the cyber world is a symbolic simulation of the real world.

5.3 Unsolved Issues in NDN Workspace

When we implement NDN Workspace, we notice that there are some issues occuring in corner cases that we were ignorant of. We are actively investigating these issues and have some drafts of solutions.

5.3.1 Scalability Concern

The design of NDN Workspace is designed for a small group of users on a shared document. This is due to the naturally relaxed real-time requirement of the collaboration editing scenario: though a document may be shared with a large group, it is unlikely that hundreds of people are editing it at the same time. As aforementioned in 5.1.3, human users tend to avoid simultaneous editing of the same text block. Therefore, the workload we used to evaluate NDN Workspace (§4.2) is low-intensity compared with general stress tests for distributed databases.

Regardless of the application scenario, we consider the following factors limiting the scalability of NDN Workspace.

- The size of SVS states grows linearly with the number of users. In a scenario with a large amount (such as hundreds) of users, the size of SVS Sync Interest will be significant, and the network traffic will be heavy since this SVS state is frequently exchanged. We are finding algorithms to reduce the SVS state size, such as compressing the state vector, exchanging active parts instead of the whole state, and grouping participants.
- The size of the CRDT document grows linearly with the number of updates, which may hit the storage limit if used in constrained devices.

5.3.2 Multiple Instances

In SVS-PS, one sequence number of a member must uniquely identify a publication. Therefore, the sequence number must be saved to the persistent storage before the program shuts down and loaded when the program starts. The program needs to be carefully implemented so that a sequence number is never reused. Also, if a user has multiple devices or runs multiple instances, then each instance must join the sync group as a separate member, using separate sequence numbers. Currently, for implementation simplicity, we require the user to bootstrap into the workspace for each instance and become individual members. An ideal solution is to separate the user's identity in security and the sync group members used in the workspace, and support multiple devices.

5.3.3 Snapshots

NDN Workspace assumes every member has a full history of changes, which leads to an overhead when the history is long. Also, when a member is offline for a long time and becomes online again, it takes a long time for the member to catch up, since each publication is an individual packet. To solve these issues, NDN Workspace needs to be able to create snapshots, the checkpoint for the history at a certain state. We recognize that there are two types of snapshots: a *local bundle*, which is published by a member on its own decision, recording the content of the workspace up to a specified sync state into one packet; a *global snapshot*, which is created on a global consensus, telling the sync state before when the history can be forgotten.

A local bundle is easy to design since its creation does not require coordination. When a member comes back online, after it learns the latest sync state, it can try to fetch a local bundle created by any other online peer. Since we assume that all members who have write permission to the workspace are equally trustworthy, the creator of the bundle does not matter. Since a bundle contains the sync state when it is created, the new online member can always fetch the SVS-PS publications after that sync state, and the result will be the same no matter what sync state the bundle is created upon. Therefore, each member can have its own schedule for the creation of bundles. If we want to avoid duplication, we can also use some random selection method, such as doing a round table. Local bundles are optional, so it is forgivable for a member to not create the bundle on the scheduled time.

Currently, we are still investigating the need and the design space of a global snapshot.

5.3.4 Failures

SVS delegates failure handling to the application due to the lack of domain knowledge. However, even in the specific application scenario, the diversity in error causes error handling to be a complicated process. In today's client-server application, since the security is attached to the connection, every failure is attributed to connectivity. Therefore, the only thing that the client does is to shut down the connection to the server and tell the user whenever a failure happens. This prevents the user from using the application until it reconnects.

The same process does not work in NDN Workspace, since the security is decoupled from the connection. For example, suppose an adversary forges a record and injects it into the workspace by sync vector pollution.[3] In a typical client-server application, this attack is not noticed by any member after the fake record gets into the server's storage. However, in NDN Workspace, members will notice the attack via a signature validation failure, but this problem does not automatically get fixed by reconnecting to the NDN network. If the application tries to cut down the connection after failure, the user will not be able to work online until the issue is fixed. To achieve high availability, we want to avoid this kind of hard failure.

In development of NDN Workspace, we recognize at least the following events may cause a noticeable failure to happen.

- A sequence number is skipped, *i.e.,* no Data packet produced.
- A Data packet is produced but not able to be fetched. Waiting for longer and reconnecting may solve this issue.
- The producer is permanently lost. The inactive member needs to be removed from the sync group.
- Sync state vector is injected.
- A data record is forged so the signature verification fails.

We leave the more detailed investigation and possible solutions to future work.

5.3.5 Name Coupling

As shown in $ 3.2.1, current user names are under the name of application instances. However, to enable users' complete control over user-generated data, we believe users should

[3]This attack is not trivial since the SVS notification Interests are signed.

have independent identities. We are exploring the direction of practical inter-domain trust relations [YMX22].

5.4 Application Driven Networking Research

NDN Workspace is the first application for daily use deployed on the NDN testbed, which plays as a driven application of NDN research. In the deployment of NDN Workspace, we also made progress in testing and improving existing efforts.

Integrates existing components Previous to NDN Workspace, the NDN Repo (§6.6) only supported on-demand data insertion. We when deploy NDN Workspace for asynchronous communication, we realized that there is a need for the Repo to join a Sync group and overhear the real-time publications. We updated the protocol of NDN Repo to implement this feature. In the bootstrapping of NDN Workspace, we realized it would be easy for users to reuse existing identities in other applications, which drives the integration of OpenID Connect into NDN Cert.

Generate real-world traffic flow As the first real-world NDN application deployed on the NDN testbed, it generates traffic flow in daily basis, which are recorded as traces for future analysis [TPS23].

Debugs NDN software and testbed In the development of NDN Workspace, we figured out a design issue related to ASF forwarding strategy, fixed several implementation bugs in the NDNts library, and improved NDN testbed by removing unreachable nodes and containerized essential software.

CHAPTER 6

Other Contributions to NDN Ecosystem

This chapter introduces my contributions to the general NDN ecosystem, including the implementation of libraries, forwarders, and tools. These contributions have helped in easing the burden of usage and application development over NDN.

6.1 DLedger and GitSync

DLedger is a distributed logging system over NDN [ZVM19]. It is lightweight storage system that supports immutable append-only logs. GitSync is a distributed git platform build on top of distributed ledger [MZ21].

DLedger is made up with two parts:

- a directed acyclic graph (DAG) as data structure to organize log records;
- a synchronization protocol to inform peers of the latest update.

6.1.1 Overview

GitSync runs as a daemon on every peer, which acts as a local git server. A local user can push to and pull from this daemon just like connecting to a server. We write a `git-remote-helper` to let git support NDN protocol.

GitSync is built on top of DLedger. It overlaps git object storage and DLedger's DAG data structure(§ 6.1.2): all git objects are synchronized as DLedger records and stored in bare git repositories; git branch references are used as DLedger's tailing records. For example, when a user pushes a commit to the `master` branch of `foo` repository, GitSync will

- store the commit and all objects referred to by this commit into a bare `foo` repository (§ 6.1.2);
- set the `master` branch reference to be this commit (§ 6.1.5);
- publish DLedger records `foo` repository (§ 6.1.3).

To reduce the size of single synchronization, GitSync uses a different ledger for each repository.

Inspired by Gerrit, GitSync leverages git repositories and branches for user management, project management and code review. GitSync uses a git repository `All-Users` to store the information for all users, user groups and certificates. For each repository, GitSync uses a metadata branch for project management, storing users' privileges associated with this repository. For code review, GitSync stores each proposed code change into a different branch. Review comments are also stored in this branch.

6.1.2 The DAG Data Structure

Each record of distributed ledger carries pointers to related records besides its payload. Relationship between records is defined by the application. In GitSync, we use the relation between git objects.

- A commit object points to its parent commits and the root directory of this commit. A commit may have multiple parents, but only one root directory.
- A directory object (called *tree*) points to files and subdirectories.

These records points to each other and form a directed acyclic graph (DAG). For example, in Figure 6.1, the head of the `master` branch is set to be the commit "`98ca9`". The commit

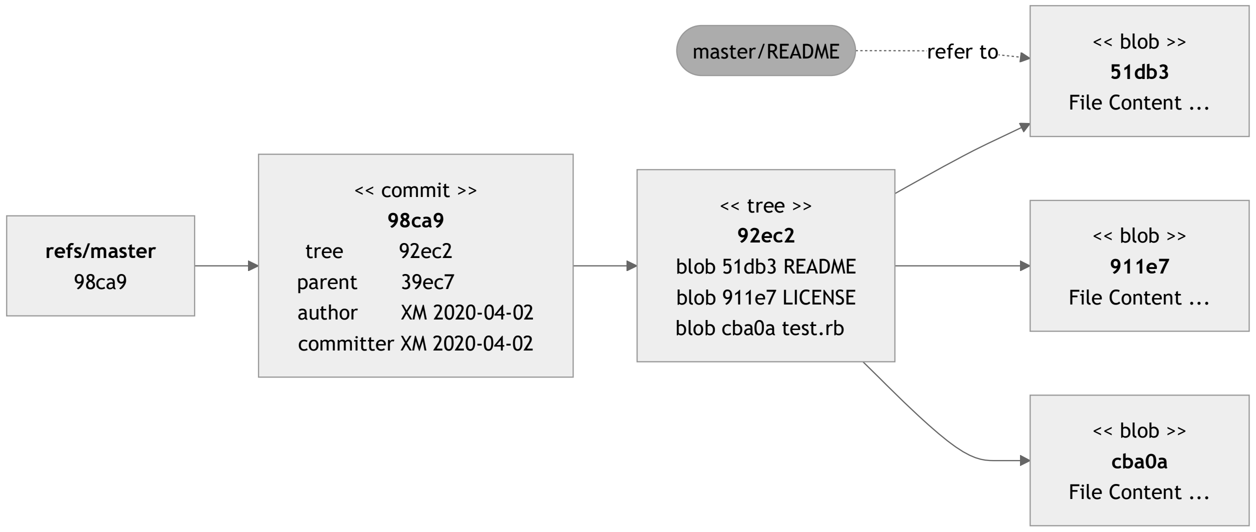


Figure 6.1: Data structure of git objects

refers to its parent commit "`39ec7`", and the `tree` record "`92ec2`" representing the root directory of this commit. In the root directory, there are three files: `README` represented by record "`51db3`", `LICENSE` represented by record "`911e7`", and `test.rb` represented by record "`cba0a`".

Records in DLedger and GitSync are immutable, as they are named in hashes, but the pointers of branches are mutable. These pointers and the reference relations among records form a mapping between the semantic file paths and the contents. Using the example of Figure 6.1, when the application see the file path "`master/README`", it uses the pointer of `master` branch to find the commit record "`98ca9`", and follows the reference chain to reach the record "`51db3`" containing the file content.

6.1.3 Synchronization Protocol

Similar to git, each peer is supposed to host all records, so that a record can be fetched from any peer as long as its name is known.

To disseminate records in DLedger, a peer needs to synchronized the namespace of records, *i.e.,* the names of all records. After a peer has the latest namespace, it will be able to fetch missing records. Since all records form a DAG, it suffices to learn the *tailing records*, the records without being pointed to by any other record, since all other records are referred to by them. In GitSync, peers synchronize on latest commit of every branch as tailing records. When a peer discovers a new commit, it recursively fetches the missing records and crawls along the references, until all records are stored locally.

Peers synchronize by sending a notification Interest containing names of all tailing records. More specifically, the peer puts a digest of all tailing record names into the `ApplicationParameters` field of the Interest and sends it out. When another peer receives this Interest, it

- compares the local list of tailing records with the digest in the Interest packet;
- if the local state is out of date, this peer starts recursive fetching process;
- if the received state is out of date, this peer sends another synchronization Interest to inform the sender;
- unless the two list are the same, this peer sends the a synchronization Interest with latest tailing record digest to faces other than the incoming face, to propagate the latest state to other peers.

A peer sends synchronization Interests on three situations: (i) periodically, (ii) after it receives an outdated Interest, and (iii) after one or more new records are created. The periodic synchronization ensures all peers to have the same knowledge of ledger; the event driven synchronization acts as a notification of newly generated records. The synchronization interval is decided by the specific application, which is 30 seconds in GitSync by default.

6.1.4 Scalability: efficient data dissemination over NDN

Notification Interests do not suffer from scalability issues. The size of notification Interest only depends on the size of the tailing record set, which does not increase linearly with the number of peers. Also, if there are too many tailing records at the same time, a peer can create a record with an empty content but referring to all existing tailing records, and this new record becomes the only tailing record to be synchronized. If the traffic overhead is too heavy, a peer can also choose to surpress too frequent notification Interests. Records can be segmented into multiple Data packets, and the consumer reassembles them after all segments are fetched.

DLedger relies on NDN Interest multicast on notification Interests to achieve efficient data dissemination. With intrinsic support of multicast, peer discovery is naturally delegated to the network layer, which is not maintained by DLedger itself. Implementing peer discovery at the network layer also improves performance, because the network layer is aware of the physical or topological distance between two peers, so neighbour peers in DLedger are close to each other in latency. Upper layer solutions, such as Kademlia [MM02], often leads to a virtual network where two neighbour peers are physically far from each other.

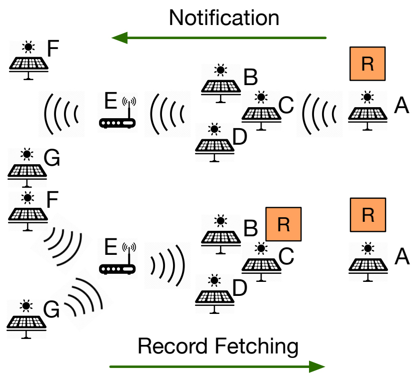


Figure 6.2: Notification and Record Fetching

NDN's Interest aggregation and in-network cache also help in reducing the traffic. Figure 6.2 shows such an example. After the producer A multicasts the synchronization Interest to its first-hop neighbors B, C, and D, they will relay the multicast to the further peer E. After one peer multicasts this Interest, duplicated Interests will be suppressed. When F and G receives this synchronization Interest, they will send Interest packets to fetch the new record. Their Interests will be aggregated at E. If C have already fetched the data, it will serve the record from local cache. [ZVM19]

6.1.5 Conflict resolution

DLedger is an append-only system. As a result, a peer is able to append records without communicating with any other peer. The synchronization process will propagate new records and eventually every peer will receive it. Therefore, DLedger is tolerant to network partitioning. Due to CAP theorem [Bre00], DLedger itself cannot offer strong consistency on mutable objects. However, we can still achieve consistency in GitSync via finer granularity. We categorize data into four cases. From the perspective of mutability, data can be categorized into append-only data and mutable data. From the perspective of ownership, data can be categorized into data modifiable by single user, and data modifiable by multiple users. For example, a developer may first write code under his own branch and then merge it into `master`. Then, the branch owned by him is modifiable by a single user, and the `master` branch is shared by multiple users. Data modifiable by a single user are consistent by nature, since a user's operations are ordered by the user. We may assume the same user will not perform conflict operations at the same time, using the same identity, but at different locations. If this rare case happens, we may pop an error to the user and let the user handle this manually, just like what git does when a merge conflict is detected.

1. *Append-only data modifiable by single user.* They are consistent by nature. Nothing special needs to be done.

2. *Mutable data modifiable by single user.* Since DLedger only supports append-only data, event-sourcing can be used. Event-sourcing means to keep the history of all operations done to an object, so the history is append-only and the object can be restored by history logs. A branch assigned to a single developer belongs to this case. Given that only one user is allowed to generate this history, there is generally no conflict.

3. *Append-only data modifiable by multiple users.* Multiple users may append data at the

same time. However, the result is a collection of all data, which is irrelevant to the order of appending. Therefore, we can simply merge all data together. Code review comments belong to this case.

4. *Mutable data modifiable by multiple users.* Event sourcing is used to transform mutable data into append-only histories, but the result of this history is ordering sensitive. Thus, conflicts are unavoidable and the system cannot automatically resolve conflicts. Shared code branches such as `master` branch belong to this case.

In software developing scenario, code conflicts caused by developers must be eventually resolved by developers. The system can only give limited support on this. Therefore, GitSync only tries to resolve conflicts as much as possible, and falls back to popping up an error to the user if it is not resolvable. Fortunately, code review can reduce the chance of getting a conflict. Also, most non-code data in GitSync belong to 2 or 3, which is resolvable.

6.1.5.1 Code branch

A code branch is not append-only.

- Developers may push conflict commits.
- Developers may revert a submitted commit.

For the second case, DLedger does not allow withdrawing any existing records. Therefore, GitSync maintains a history branch for each code branch, which records all changes to the head of the code branch. Then, reverting a branch becomes pushing a "revert" commit to the history branch, which can be stored in DLedger.

The first case must be handled manually. If the code branch is maintained by a single developer — for example, a personal branch — this case will be rare. This is because generally, a developer will not submit conflict changes at different locations at the same time. If the code branch is shared by multiple developers, code review process can reduce the chance of conflict commits.

6.1.5.2 Code review

Code review can be considered as a consesus algorithm executed by human beings, which will solve most potential conflicts on `master` branch. In GitSync, code review is designed as follows:

1. A developer propose a change by creating a change branch, referring to another code commit which contains the actual code changes.

2. Reviewers can write comments into this branch. Each comment is immutable after creation. If multiple reviewers submit multiple comments at the same time and lead to branching, GitSync will automatically create a merging branch that include all comments.

3. Reviewers can vote for the code change. A reviewer is allowed to change his vote by voting again, only the latest vote will count.

4. After the code change gets enough positive votes, it can be merged into the target branch. A user can create a commit to the target branch's history branch, which refers to all positive votes and suggest this change. This step may result in a conflict in the history branch, but if every change must go over code review before merging, there is small chance to have a conflict.

5. Other peers see the commit in the history branch, verify the votes, and set the target branch to be the version including the proposed change.

For step 2 and 3, even though multiple comments or votes can be automatically merged, if multiple peers are creating the merged commit at the same time, their merged commits may lead to new conflicts. Therefore, automatically merged branch should be identical and independent on by who and when it is created. This is achieved by removing author signature and timestamp in the merged branch. Other peers can check the validity of merged branch by executing the same merging algorithm. Thus, a valid signature is not necessary. This also makes non-reviewers, who do not have write privileges to this branch, be able to perform automatic merging.

6.1.6 Security

In git, every object is referred to by its hash name. Therefore, as long as the heads of branches — references to latest commits — are secured, data integrity is achieved. Git does not specify how to secure these branch heads. In GitSync, every file in a non-code branch is signed by the producer. Since a code branch's head must agree with its history branch, where the latter branch is signed, all data in GitSync are secured by the trust system.

6.1.6.1 Access control

GitSync uses `All-Users` to contain all information about users including their certificates. This repository is also synchronized by DLedger. Therefore, every peer has all public keys of other peers, so peers can authenticate each other.

In GitSync, each repository has a meta branch containing project configurations of this repository. Access control rules are also defined in this repository. Each rule allows or denies a user or user group to perform a specific operation, such as proposing changes, adding review comments, pushing commits, etc. Most rules are defined at a granularity of branch.

For authorization, there is no authority server that handles it in GitSync. Every peer checks authorizes commits based on its local knowledge. Therefore, there is no single point of trust in the system. When a peer receives a new commit, it does the following for every file:

1. Verify the signature.

2. Look up the permissions stored in the repository, and check whether the signer is allowed to creating or modifying this file.

3. If any of the checks fails, reject the commit.

6.1.6.2 Bootstrapping

To join a GitSync system, a peer needs to be able to authenticate and be authenticated by other peers. To authenticate other peers, it needs to know the trust anchor of the GitSync system. Since every certificate is directly or indirectly signed by the trust anchor, the new peer is able to start synchronization process immediately after it has the trust anchor. Eventually, it will have all certificates and be able to verify every other peer. To get authenticated by other peers, it needs to have a certificate recognized by other peers. This certificate can be issued by any administrator peer who has the privilege to write `All-Users` repository. After the administrator peer gets the public key of the new peer, it will sign a certificate for this key and add it into `All-Users` repository. Then, this certificate will be eventually disseminated to all peers.

Therefore, the security bootstrapping of GitSync works as follows:

1. A new peer communicates with an existing administrator peer and prove its identity by some out-of-band method. Existing tools such as NDNCERT [ZAZ17] can be used here.

2. The new peer sends its public key to the administrator peer, and receives the trust anchor.

3. The administrator peer issues a certificate, creates a user for the new peer and adds them into the `All-Users` repository.

4. The new peer starts synchronization.

6.1.6.3 Certificate Revocation

Since DLedger does not support deleting records, to revoke a certificate, a peer needs to add a revocation record for the certificate to indicate it is invalid. In some case, this revocation record may be received before some old records signed by the certificate are received, which may lead to inconsistency. To prevent this, a revocation record needs to contain all tailing records at the time the certificate is revoked. All records signed by the revoked key are invalid only if they are created after the tailing records in the revocation record.

6.1.7 Summary

DLedger provides distributed append-only logging. GitSync is built on top of DLedger due to the following reasons.

- With the support of NDN on multicast and peer discovery, DLedger solves the peer-to-peer connectivity issue and achieves efficient data dissemination.
- DLedger adopts DAG as its data structure which is aligned with git's object model. By reducing sync state to the tailing record set, it decouples the state size and the number of peers, naturally solving the scalability issue.
- Peer management is reduced to data management of a special repository. No extra work is needed.

However, after our prototype implementation, we found several issues in the design.

Not integrated with storages. DLedger does not have generic in-network storage in the design and it does not support partial synchronization. It requires every peer has a full history of records, and depend heavily on the peer's local storage. In an asynchronous scenario, DLedger requires there are always some peers online, and their uptime overlaps with each other. These peers must be enrolled in the application's trust domain and cannot be generic storages. The easiest way to achieve this is to set up a server running DLedger as a peer, but if we do so, the server peer becomes similar to a central server in the sense that data exchange is controlled by the application deployment.

Drawback with flat names. GitSync uses flat naming for immutable records, which inherits the drawbacks of flat naming. Lack of semantics in naming makes it impossible to infer missing data names, and therefore crawling along the graph becomes a necessity. When the reference relations form a very long chain, it will take a long time for a peer to sync up. Crawling cannot make full use of the bandwidth when records are small in size, unless someone else is online and able to serve a snapshot.

Security coupled with transport. The security of GitSync is tightly coupled with data fetching. This coupling makes implementation hard. A record's validity can only be verified after all records prior to it is fetched. During the fetching process, data packets fetched but not verified yet must be cached in the local storage. This means that each peer has to implement two storages: one has stack-like data structure for the fetched but not verified records, and the other for the verifed data records. This could also lead to potential vulnerbilities. For example, the depth of the record reference chain is unknown and thus can lead to runtime stack overflow. Also, a malicious record can trigger multiple other peers to send Interests before the record is detected in security verification.

Conflict resolution needs human attention. GitSync cannot automatically resolve conflicts in branch heads, and therefore only has limited support in real-time collaboration.

6.1.8 Comparison between SVS-PS and DLedger

A comparison of SVS-PS and DLedger is listed in 6.1.

	DLedger	SVS-PS
Publication Record Name	Flat name	Application name
Data Packet Name	Flat name	Sequence number
Reference	Refer to historical records	Wrap the newly published records
Immutability	Immutable records only	Not the goal
Data Persistence	Internal	Application specified
Sync State	Tailing record set	State vector
Sync State Size	Depend on record generation	Equal to # participants
Notification	Notification Interest and Data	Notification Interest only
Signed Notification	Not signed	Signed
Rate control	Interest aggregation	Supression Timer
Data fetching	Recursive fetch the missing records	Defined by application

Table 6.1: Comparison between DLedger and SVS-PS

Data records : DLedger records are required to be immutable, and named with flat names. There are built-in references among records, making the records a DAG, which is used to reduce the state size and crawl historical records. Though it is possible to publish a full Name[1] as a record in DLedger, this will put the actual Data packet outside of the interlocked DLedger storage, which is against the design goal. SVS uses sequence number for Data packets. SVS-PS has no restriction on the record names, allowing application namespace to be used. SVS and SVS-PS are not supposed to be used as a storage system. Specific applications must handle Data persistence themselves.

Sync state and data fetching : DLedger is designed for a permissioned ledger scenario, *i.e.,* hundreds to thousands of peers owned by several organizations that reciprocally recognize each other in a collaboration. In DLedger, the sync state is the names of tailing records, which are independent of the peers. Therefore, peers are open to join and leave without special handling in DLedger (except for security configuration required by the application). A benefit of this is that each participated organization can operate individually after mutual trust relationship is established. Other peers do not need to be aware of the participant list. The size of the DLedger state, the set of tailing records, depends only on the parallel production rate of records, but not on the number of participants. Therefore, the scalability is unlikely to be restricted by the number of peers, but instead the length of history is likely to be a limitation. Since each peer is required to fetch all records, and the way to fetch records is recursively crawling the chain, it may take very long for a newly joined peer to catch up with the latest status.

SVS is designed for a small sync group (less than 100 participants) within a single trust domain. The design to scale to a larger group is under development. In SVS, every state vector contains all participants, and every participant is required to obtain the latest state

[1]The full name of every Data packet includes a logical final implicit digest component, which secures the integrity of the content.

vector. Therefore, every peer has to obtain the latest participant list. With the increase of the number of participants, the state vector also increases in size. Currently, there lacks a way to remove a participant from the state vector, even the participant itself is removed from the group. Therefore, the group member list should remain relatively stable to avoid potential scalability problems. However, in SVS-PS the record names are decoupled from the sync group, so it is possible to reconstruct a new sync group and include historical record names when the scalability becomes a problem. With the help of sequential naming, an SVS peer can fetch multiple missing data packets in parallel, which accelerates the new participant to bootstrap. In NDN Workspace, we find out that parallel fetching can significantly reduce the fetching time when an offline peer befomes online again. In SVS-PS, only application record names are involved in the sync, and the fetching mechanism is defined by the application. If the application needs, segmentation can be supported.

Security : DLedger relies on the reference relationship for security. When a record is referred to by another record, its content hash is also locked in the new record. Therefore, an adversary who wants to forge a record has to forge all records directly and indirectly referring to it. In the original paper [ZVM19], a reference is called an "endorsement". The more peers endorses an record, the harder it is to forge this record. After an incident occurs, the DAG is submitted to a trustworthy third party for audit, and the number of endorsement will be considered in the audit process. For SVS and SVS-PS, since records interlock and auditability are not the goal, the integrity of a record is soly secured by the signature of its producer.

However, DLedger's method has a drawback when we talk about network layer security. Due to the usage of flat names, NDN name based trust schema cannot be used in DLedger. From the network's perspective, all peers of DLedger have to be equal in status, and DLedger itself allows any participated peer to publish any record. In SVS, a packet name contains the participant's name as a prefix; in SVS-PS, a record name can be any name in the application's namespace. Therefore, both SVS and SVS-PS can make use of NDN trust schema.

6.2 Light VerSec: Language for Trust Schema

This section introduces my work on Light VerSec. The major part of this section is adapted from [YMX23a]. I have designed the Light VerSec language based on VerSec [Nic19], and implemented the library in Python.

6.2.1 Background and Motivation

Trust schemas (§ 3.1.4) play a key role in NDN security. As entities enroll into the trust domain and the roles of entities within the trust domain change, trust schemas also need to evolve over time. To make sure all entities in the trust domain receives and executes the same trust schema, a systematic way to express trust schema is needed. One important port of trust schemas is the rules on the relations between data and producer names, *i.e.,* "who can produce what". Here, we assume that all the information needed for decision making must be encoded into data names and/or key name, as NDN encourages expressing the relationship between *semantic names*. This restriction makes it easy to design languages to systematically express trust schema rules, and libraries to automate the execution of them.

6.2.2 Design and Implementation

We developed LightVerSec, a modified VerSec [Nic21b], to be used as the trust schema language. In the following, we introduce LightVerSec from three perspectives: LightVerSec syntax, modifications from VerSec, and name schema tree.

LightVerSec Syntax: LightVerSec employs pattern matching on NDN names to validate packets. To validate a packet, a LightVerSec implementation first searches for a rule in the defined trust schema whose name pattern matches the packet name, and then checks whether the KeyLocator complies with the pattern specified by the rule.

In LightVerSec, a *name pattern* consists of a sequence of name components and pattern variables, separated by slashes. Name components are enclosed in quotes, while patterns are represented as C-style identifiers. For example:

```
"TechDaily"/"admin"/adminID/"KEY"/_
```

is a name pattern consisting of three name components and two pattern variables (`adminID` and `_`). A name pattern can only match a name when they have the same length and name components given in the name pattern equals to the components in the name at the corresponding positions. Then, the pattern variables are assigned with corresponding components in the given name. Variables starting with an underscore are considered as temporary variables and not assigned. Name patterns can be named with an identifier starting with a hash. When this identifier occurs in another name pattern, it is replaced by its definition.

A rule is defined in the form of

```
#PacketNamePattern <= #KeyLocatorNamePattern
```

Which means any packet matching "#PacketNamePattern" should have a key locator matching "#KeyLocatorNamePattern", and the pattern variables with same names assigned during the matching must agree. Extra restrictions can be expressed in the form of `{var: func()}`, where "`var`" is a pattern variable and "**func**" is a user defined function passed to the LightVerSec implementation at run time.

Listing 6.2 gives an example of TechDaily's trust schema. In this trust schema, six name patterns and three rules are defined; the three rules are the administrator rule `#admin<=#root`, the author rule `#author<=#admin`, and the article rule `#article<=#author`. In the `#article<=#author` rule, the "authorID" in the article's name is required to be the same as is in the name of the author's key. A user-defined function named `isVersion()` is used to restrict the last name

```
TAG_IDENT = CNAME;
RULE_IDENT = "#", CNAME;
FN_IDENT = "$", CNAME;
CNAME = ? C/C++ identifiers ?;
STR = ? C/C++ quoted string ?;

name = ["/"], component, {"/", component};
component = STR
          | TAG_IDENT
          | RULE_IDENT;

definition = RULE_IDENT, ":", def_expr;
def_expr = name, ["&", comp_constraints], ["<=", sign_constraints];
sign_constraints = RULE_IDENT, {"|", RULE_IDENT};
comp_constraints = cons_set, {"|", cons_set};
cons_set = "{", cons_term, {",", cons_term}, "}";
cons_term = TAG_IDENT, ":", cons_expr;
cons_expr = cons_option, {"|", cons_option};
cons_option = STR
            | TAG_IDENT
            | FN_IDENT, "(", fn_args, ")";
fn_args = (STR | TAG_IDENT), {",", (STR | TAG_IDENT)};

file_input = {definition};
```

Listing 6.1: LightVerSec Grammar

```
#KEY: "KEY"/_/_/_version & {_version: isVersion()}
#site: "TechDaily"
#root: #site/#KEY
#admin: #site/"admin"/adminID/#KEY <= #root
#author: #site/"author"/authorID/#KEY <= #admin
#article: #site/"article"/topic/articleID/authorID/_version & {_version: isVersion()} <= #
    author
```

Listing 6.2: The policy of the TechDaily website

component of every packet, which simply checks if the component is a valid version number. The formal Backus–Naur Form (BNF) notation of LightVerSec is shown in Listing 6.1.

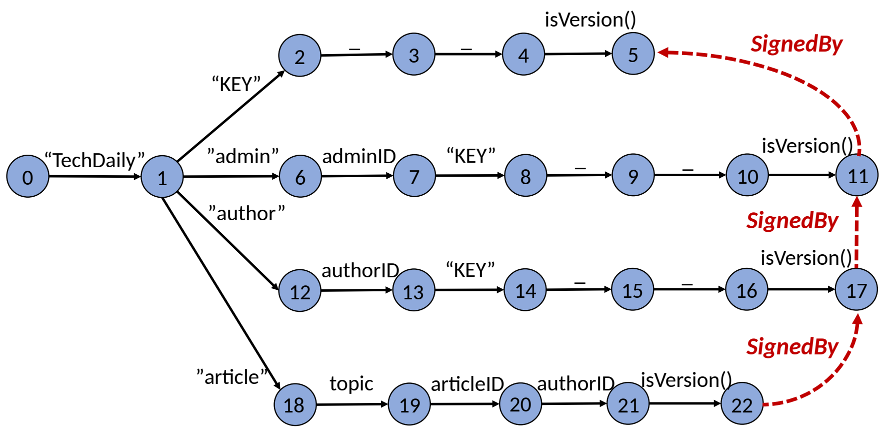


Figure 6.3: An example name schema tree compiled from the trust schema text

Modifications from VerSec: LightVerSec differs from the original VerSec in the following two aspects:

1. *Decoupling from Pub/Sub.* VerSec has specific syntax and semantics defined for Pub/-Sub, such as publication naming rules [Nic22]. Though the core VerSec language does not require Pub/Sub implementation, it is hard to implement a VerSec compiler without Pub/Sub features. To target more general NDN applications and simplify compiler implementation, LightVerSec removes such Pub/Sub specific support. Internal functions in VerSec are also replaced by more flexible user-defined functions in LightVerSec, which are

written by the application developers and passed to the implementation at runtime.

2. *Focusing on Authenticating Received Packets.* When compiling a trust schema, the VerSec compiler can optionally embed internal information for building publications, such as signers, while LightVerSec does not optimize for Data producers, simplifying compiler implementation.

These modifications do not restrict the usage of the language. That is, all applications supported by VerSec are still supported by LightVerSec. The differences only affect the way to write the schema and use corresponding libraries.

Name Schema Tree: To optimize the execution of the trust schema, LightVerSec compiles the trust schema text into a trie-like data structure called a *name schema tree*. In this data structure, each edge represents a name component or pattern variable, and each leaf node corresponds to a name pattern of a packet. This name schema tree efficiently represents the structure of the trust schema.

When a trust schema is compiled into a name schema tree, rules are attached to the leaf nodes. These rules are represented by pointers that point to the KeyLocator's name patterns. This allows for quick and accurate matching of packet names against the defined trust schema. For example, Figure 6.3 illustrates the compiled name schema tree for TechDaily's trust schema. It demonstrates how the trust schema is transformed into a hierarchical structure for efficient validation and verification of packet names.

6.3 Name Tree Schema: Programming Framework for NDN Application

Continuing the work on Light VerSec, I proposed Name Tree Schema (NTSchema), an application framework that organizes program functionalities by the application namespace.

6.3.1 Background and Motivation

NDN application development capabilities can be viewed as a spectrum (Figure 6.4) along the dimension of the role NDN libraries play in the development. As the features move to the right of the spectrum:

- The developer needs to do less programming work.
- The libraries play more important roles in the development.
- The developer's input (such as source code) becomes more declarative. That is, the developer describes more on what the application does, and less on how to achieve it.

There are many different ways to support application development, and there are many use cases. This spectrum shows the range of capabilities of NTSchema application framework, and a roadmap of future works to support application developers. Namespace is the most important concept for NDN application. A namespace used by an application can be viewed as a tree. Since objects have static structures and dynamic states, the namespace tree also has two forms — the name schema tree and runtime name tree. NTSchema exploits the schema tree to provide a uniformed programming framework for NDN applications.

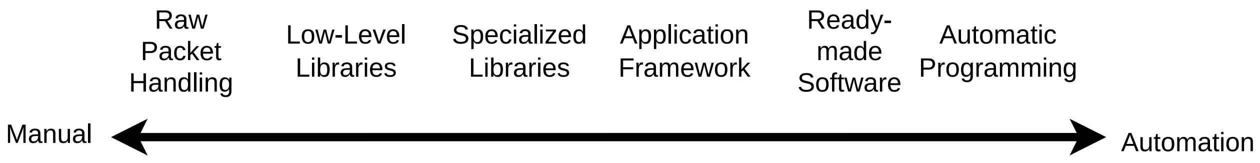


Figure 6.4: Application Development Spectrum

6.3.2 Design and Implementation

As shown in 6.2, named data objects of an application can be viewed as a tree of schema, which can be used to express trust relations. In NTSchema, we further use the schema tree to organize the application programming itself, *i.e.,* write the code in a tree. More specifically, NTSchema provides the follows

- **Nodes:** typed components as tree nodes and subtrees, which can be nested and reused as programming modules handling data object generation and retrieval. For example, segmented object node handling data segmentation and reassembling can be reused to build a versioned segmented object component, like in Real-Time Data Retrieval (RDR) [MGA18].

- **Policies:** customized annotations attached to nodes, called *policies*, which are used to expressing attributes of data objects. For example, trust relations (like the edges in the tree of Light VerSec in Figure 6.3), persistent local storages, prefix registration, etc..

- **The processing pipeline:** A standard pipeline processing low-level Interest and Data exchanges, including data object production and consumption, insertion into and look up from storage, signing and verification, etc..

6.3.2.1 Example: RDR

In this section, we use the example of RDR to demonstrate how NDN programs are developed using NTSchema. The RDR (Real-time Data Retrieval) protocol [MGA18] is a protocol designed for discovering the latest version number of segmented object, and fetching the object. The prefix of the segmented object is provided as input, assuming it is "`/<filename>`". There are two kinds of data packets in RDR:

- **Segmented Data packet**, named as "`/<filename>/v=<version>/seg=<segment>`". This packet represents the segment numbered "`<segment>`" of the version "`<version>`". The version number is typically a timestamp.

- **Metadata packet**, named as "`/<filename>/32=metadata`". This packet represents a metadata packet, containing the name of the latest version of this object, used in version discovery.

To fetch an RDR object, the consumer needs to first fetch the metadata packet by sending an Interest carrying the prefix "`/<filename>/32=metadata`". Then, the consumer will learn the latest version number carried by the metadata packet, and fetch the corresponding segments and performs security verification. To produce a new version of an RDR object, the producer needs to generate a version number for it. Then, the producer segments the object's content into segmented data packets, signing it with the key, and publishes a new metadata packet carrying the name prefix of the new version.

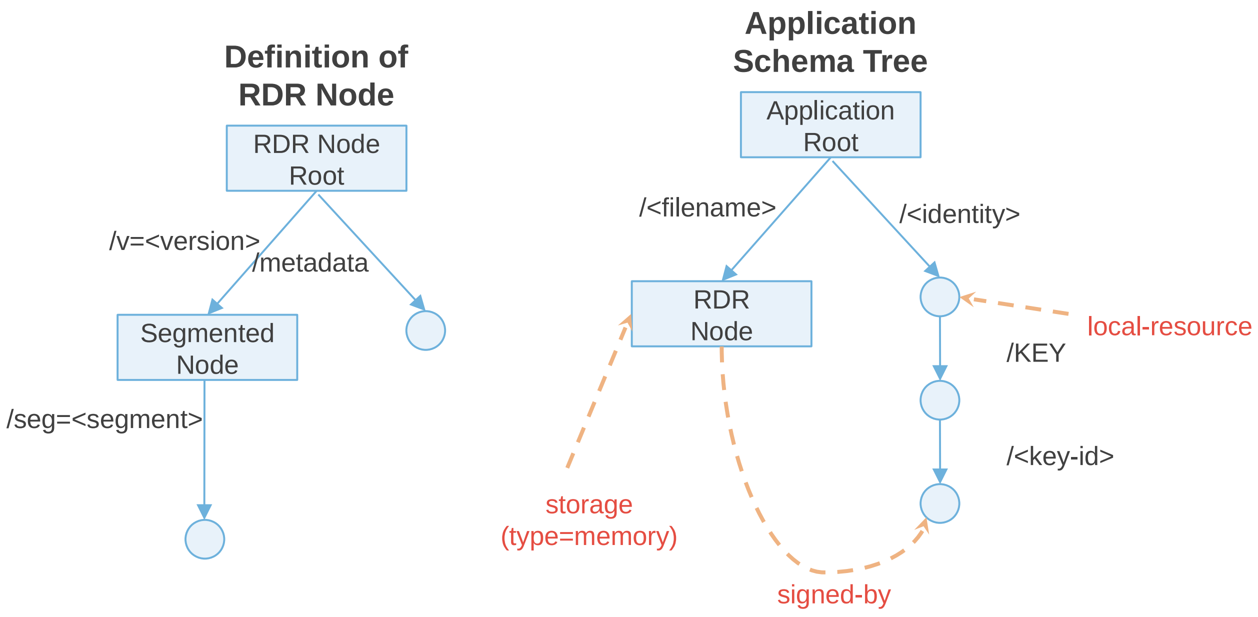


Figure 6.5: RDR example using NTSchema

Figure 6.5 shows the NTSchema tree of the RDR application. In the figure, we assume the user's public key is of form "`/<identity>/KEY/<key-id>`", which is provided to all peers when the program starts. There are two typed nodes (except the trivial ones) and three policies involved. The two non-trivial typed nodes are *SegmentedNode* and *RDRNode*:

- **SegmentedNode:** this node handles data segmentation and reassembly. It also enables a pipeline to fetch segmented data packets. This node can be used in any protocol with segmented data objects, not limited to RDR.
- **RDRNode:** this node handles version number generation and discovery. It is is a subtree which has two children. One is the SegmentedNode at path "`/v=<version>`", and the other is a leaf node at path "`/32=metadata`" representing the metadata packet. When a new version of RDR object is produced, it generates the version number and calls the leaf node at the "`32=metadata`" to generate the metadata packet. When the consumer requests to fetch the latest RDR object, it first uses the metadata leaf node to fetch the metadata packet to discover the version, and then calls the SegmentedNode child with corresponding name to fetch the object.

The three policies are *signed-by*, *storage*, and *local-resource*:

- **signed-by:** this policy specifies which naming pattern is allowed to sign the data packets under this prefix. It provides the same functionality as VerSec, but more powerful, since it cnsists of programming code, which allows Turing-complete parsing and computation on names.
- **storage:** this policy specifies the local storage to store data packets. In the figure, it specifies an in-memory temporary storage.
- **local-resource:** this policy specifies the data objects under this subtree must be fetched from the local storage only, without expressing any Interest.

6.3.2.2 Implementation

NTSchema is implemented in go-ndn written in Go, and currently being implemented in NDNts, an NDN client library written in TypeScript [Jun24].

6.4 NDN Forwarders

This section introduces my work on design and implementation of two NDN forwarders. The first one is NDN-Lite, a home IoT NDN forwarder. The work is submitted as part of [ZYM22]. I have solely designed and implemented the forwarder part of the publication. The second one is YaNFD, a multi-thread NDN forwarder written in Go. The major part of the corresponding text is adapted from [NMZ21]. I have actively joined the design process and am the current maintainer of this project.

6.4.1 Background and Motivation

For many years, the primary forwarder used for both experimentation and development in NDN has been the NDN Forwarding Daemon (NFD) [ASZ21]. NFD was among the first forwarders designed specifically for NDN and its development produced many innovative solutions to address the problems introduced by NDN's stateful forwarding plane. However, NFD is limited in some scenarios by its complex single-threaded forwarding implementation and heavyweight storage and compilation requirements. Here, we developed two forwarders: NDN-Lite for limited-performance IoT boards, and YaNFD to for end-hosts with multi-threaded forwarding.

6.4.2 NDN Lite: NDN Forwarder for Home IoT

NDN-Lite project is a lightweight NDN library and forwarder targeting resource-constrained devices.

Our goals of NDN-Lite include

- **Usability**. NDN-Lite is designed as an IoT package providing high-level application supports which benefit from NDN, such as bootstrapping, service discovery, access control, schematized trust, etc.

- **Lightweight**. NDN-Lite should be lightweight in the terms of memory, performance and overall code size to minimize the requirements on hardware resources. Non-critical function can be omitted if it does not hurt the usability.
- **Generality**. NDN-Lite is designed for a broad range of low-power IoT boards. Due to the diversity in hardware features, operating systems and developing tools, we need to put extra efforts into cross-platform compatibility.

NDN-Lite consists of 3 layers which are platform adaptation, forwarder and application support. This section focuses on the forwarder part.

6.4.2.1 Design and Implementation

NDN-Lite has different goals from NFD, which lead to different design trade-offs. NFD focuses on modularity, functionality and robustness, while NDN-Lite puts high priorities on compatibility, performance and rapid iteration. NFD's algorithm needs to be scalable, but NDN-Lite generally works on small datasets, where the constant factor affects the performance more than the complexity. Table 6.2 gives a summary of trade-offs we have made.

Similar to NFD, NDN-Lite also uses a Trie-based data structure called *NameTree*[2] for PIT and FIB, but the implementation is different. NFD uses a hybrid data structure of NameTree and hash table. NFD's tables are supposed to be large, so the time complexity is expected be constant on average. Hence, a CityHash based hash table is used when searching the NameTree. Given that most of the search operations are done by the hash table, the hash table is the main data structure who shapes the performance. The maximum sizes of tables are unknown, so in order to keep the constant complexity, the hash table is self-adjustable, i.e. the number of buckets dynamically changes. The same strategy does not work in NDN-Lite due to the limitation on dynamic memory allocation. However, the time

[2] *NameTree* is a Trie which uses NDN Name as keys to store packets, supporting insertion, prefix-matching and exact-matching.

	NFD	NDN-Lite
Language	C++	C
Memory Management	Shared Pointer	Static
Third-Party Library	Boost, OpenSSL, Sqlite	None
Collaborative Workflow	Gerrit, Redmine, Mail-list	GitHub, Module Maintainer
Face Class	LinkService and Transport	Single Class
Application	Multiple Application, UnixSocket Face	Single Application, Forwarder API
Data Structure	Self-Adjusting Hashtable	Fixed-Size Table
Dead Nonce List	Yes	No
Decoding API for Forwarder and App	Same	Different

Table 6.2: Design trade-offs of NDN Lite forwarder

complexity is not equal to performance here due to the small sizes of PIT and FIB. Thus, the NDN-Lite exploits left-child right-sibling representation, where children of a node sorted in lexical order, which enables direct search on NameTree. This is not a scalable solution due to its linear complexity, but its constant factor is small, which provides efficiency on constrained IoT devices. The deletion of a NameTree node is delayed after the space runs out in order to prevent frequent insertion and deletion when the table is sparse.

NDN-Lite has a similar pipeline to NFD for Interest and Data packets, though there are some differences due to different functionalities to support. In NDN-Lite, only the key forwarding functionalities are supported for efficiency. The Dead Nonce List, Content Store, and forwarding hint are not supported to save resource.

6.4.2.2 Evaluation

This section evaluates NDN-Lite and NFD by a localhost scenario in terms of performance, memory usage and code footprint.

We use a file transport scenario to compare the performance of forwarders. For the NFD side, there needs to be a producer program and a consumer program. The producer reads a specific file, fragments it into segments, encodes into Data packets and responds with corresponding ones when Interests come; the consumer encodes Interests, sends them to the forwarder and receives Data segments back. But for the NDN-Lite side, generally the forwarder is bundled with an application. Given that the producer has a passive role and less burden to CPU, we decide to put the forwarder into the producer side. The test environment is described in Table 6.3, and the results are shown in Table 6.4 and Table 6.5.

As shown in Table 6.4, the time NFD used is nearly 18 times that of NDN-Lite. Due to the different application designs, each packet needs to go through the Unix socket twice in NFD but only once in NDN-Lite, which means that NDN-Lite forwarder is about 9 times as fast as NFD.

	Mac OS	Raspberry Pi
Model	MacBook Pro (Retina, Late 2013)	Raspberry Pi 3B
Processor	2.3GHz Intel Core i7	1.2GHz Broadcom BCM2837
Memory	16GB	1GB
Operating System	MacOS 10.14.4	Raspbian 9
Word Size	64 bit	32 bit
Test File Size	25 MB	
Segment Size	1024 B	
Num of Packets	25600	
Name Prefix	/example/testApp/randomData	
NDN-Lite Table Capabilities	NameTree Size = 64 PIT Size = 32	

Table 6.3: Test Environment

	NFD	NDN-Lite
Time on Mac	2.40s	0.11s
Time on Raspberry Pi	11.33s	0.66s
Socket	9.8%	64.5%
NameTree	30.2%	21.4%
Parsing	28.7%	3.0%
Others	31.3%	11.1%

Table 6.4: Performance

		NFD	CXX Producer	NDN-Lite
Mac	Max	269 MB	337 MB	34 MB
	Min	13.6 MB	8.4 MB	0.8 MB
Raspberry Pi	Max	161 MB	269.0 MB	33.8 MB
	Min	28.8 MB	7.8 MB	1.1 MB

Table 6.5: Memory Usage

Table 6.4 also shows the proportion each activity takes, where Socket means time used by Unix socket, NameTree measures looking up FIB or PIT entries, Parsing includes encoding / decoding packets and constructing value objects, and Others include all other activities such as scheduled events, deletion of Interests and idle time. In NDN-Lite most time is used by Socket and NameTree as expected. However, in NFD, Socket and NameTree take less than half of the total time, especially Socket, which means NFD is not an IO bound task.

In Table 6.5, we record the maximum and minimum values of resident memory. The minimum values of producers are measured after initialization but before loading the file. That of NFD is measured before the first run of producer, so the CS is empty. There is a memory waste in the implementation of encoding in ndn-cxx [MAZ22], which results in a overhead which is 8 times as large as NDN-Lite.

6.4.3 YaNFD: Multithreaded NDN Forwarding Daemon

YaNFD is a lightweight and multithreaded NDN forwarder targeting the end hosts, written in Go.

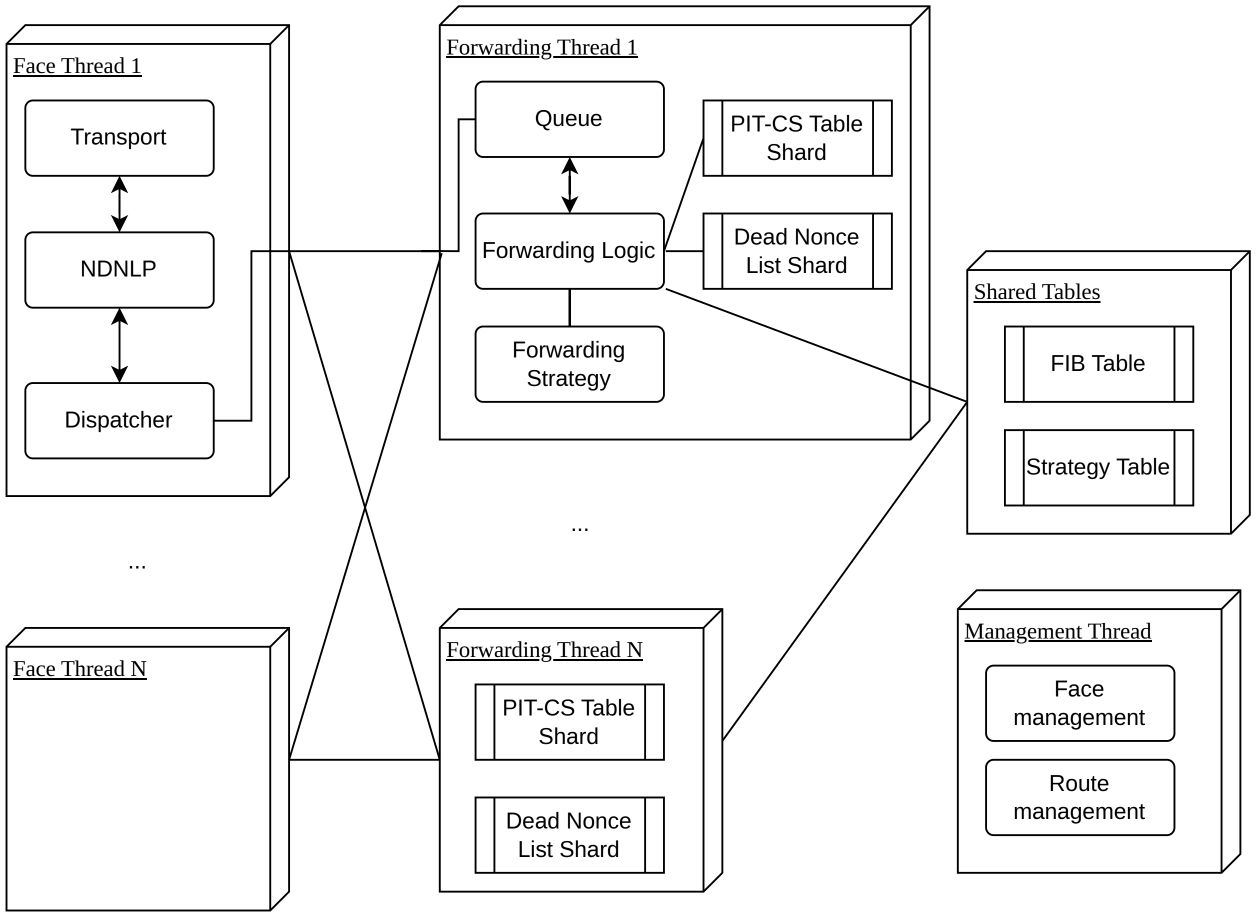


Figure 6.6: Design overview of YaNFD

6.4.3.1 Design and Implementation

The forwarding pipelines of YaNFD approximately follow the designs of NFD's forwarding pipelines, as described in detail in NFD's "Developer's Guide" [ASZ21]. However, we modified these pipelines to accommodate the use of several thread-local, as opposed to global, data structures. Additionally, we have modified the pipelines to pass packets between the different components of the forwarding pipeline using queues (utilizing Go's "channels" feature), as opposed to direct function calls. We also use channels to handle timeouts and similar events, as opposed to callbacks in NFD.

A high-level overview of YaNFD's design is presented in Figure 6.6. Our design features a user-configurable number of forwarding threads (by default, eight), two threads for each face: one for receiving (input) and one for sending (output), and a separate management thread. We inherited the link service/transport modular split from NFD (as shown in the figure), as this design allows each type of transport (e.g., UDP, Unix stream, or Ethernet) to be treated identically by the forwarder. In this design, the link service provides common features as part of the link protocol – currently, we implement NDNLPv2, which provides optional fragmentation/reassembly, hop-by-hop reliability (not yet implemented at the time of writing), congestion marking, and other features . Meanwhile, the transport is responsible for interface-specific operations, such as reading to and writing from a socket or Ethernet packet capture handle.

The link service passes received packets on to the appropriate forwarding thread by hashing complete names for Interests and through PIT tokens for Data packets [3]. PIT tokens generated by YaNFD contain both the thread ID and the identifier of the particular PIT entry the Data packet satisfies. Packets are queued to be processed in the order they are received by the appropriate forwarding thread, which will then perform its forwarding computations, including passing Interests to the appropriate forwarding strategy before pushing the packet on to the appropriate outgoing face thread queue(s), which will then process and send the packets using their link service and transport-specific mechanisms.

6.4.3.2 Evaluation

During the development of YaNFD, we evaluated it against the NDN testbed to ensure that it is both able to perform basic NDN forwarding operations and correctly interoperate with existing NDN forwarders – in particular, NFD, which is the only forwarder currently running on the testbed backbone. We found that we were able to successfully send NDN pings (from

[3]For Data packets received from local producers, it is not feasible to expect a PIT token to be attached to returning Data packets, since this is not part of the standard NDN interface with applications. Therefore, in this case, we simply dispatch to multiple forwarding threads based upon the hash of all prefixes of the Data name, dropping the packet if there are no matching PIT entries.

ndn-tools) to various nodes on the testbed, some operating over multiple hops from where we connected to the testbed. As part of this, we pinged multiple testbed nodes throughout the testbed (connecting through a North American node), including testbed nodes in Asia, Europe, and South America.

Although the above evaluation confirms that YaNFD is, at a high level, compatible with existing NDN forwarders, it does not evaluate YaNFD's correctness is regards to more detailed specifics of the NDN protocol specification. These include the various optional fields present in Interest packets, such as MustBeFresh and CanBePrefix, as well as longest-prefix matching for names. To evaluate the correctness of our implementation with more complex aspects of the NDN protocol spec, we utilized the ndnchunks consumer-producer application pair from ndn-tools.

We conducted two experiments to transfer a 100 MB file between two directly-connected nodes virtualized using Ubuntu 20.04 running on VirtualBox. In each experiment, one node ran NFD and the other ran YaNFD. The producer and consumer applications were swapped between the YaNFD and NFD nodes in each experiment in order to provide coverage of both roles. In both, experiments ndnchunks was able to successfully transfer the file, which indicates that our forwarder is able to correctly interoperate with existing NDN networks running NFD, allowing it to be deployed in combination with NFD.

We have listed the fields present in the Interest and Data packets during these transfers (excluding the "Name" and "Content" fields) in Tables 6.6 and 6.7, respectively.

CanBePrefix	Yes (first Interest), No (further Interests)
MustBeFresh	Yes (first Interest), No (further Interests)
Nonce	Random 4 byte value
InterestLifetime	N/A (implicitly 4000 ms [Namb])

Table 6.6: Values of Interest packet fields during correctness transfers.

ContentType	N/A (implicitly “BLOB” [Namb])
FreshnessPeriod	10 ms (first Data), 10000 ms (further)
FinalBlockId	N/A (first Data), Final segment ID (further)
SignatureType	DigestSha256 [Namb]
SignatureValue	32 byte SHA-256 hash of Content

Table 6.7: Values of Data packet fields during correctness transfers.

We also performed evaluation on performance at the time we developed YaNFD. We conducted file transfer tests using ndnchunks for three NDN forwarders for edge environments: YaNFD, NFD [ASZ21], and MW-NFD [BLS20]. We evaluated using three consumers requesting a different content from one of three producers on the same host, and we computed the throughput for different file sizes: 100 MB, 1 GB, and 5 GB. We averaged these values across the three consumers and then averaged this value over all trials for the same forwarder and the same file size. The results of our evaluations at the time of development are shown in Figure 6.7.

6.5 NDN Forwarder Manager

This section introduces my work on NDN Forwarder Manager, a UI tool to manage NDN forwarder based on the NFD management protocol, improving the usability of NDN forwarders. The major part of this section is adapted from [MNZ21]. I am the main designer and maintainer of this work.

6.5.1 Background and Motivation

However, up to this point, configuring NDN for use on end hosts has been difficult. In contrast to today’s user-friendly graphical tools for configuring traditional networking equip-

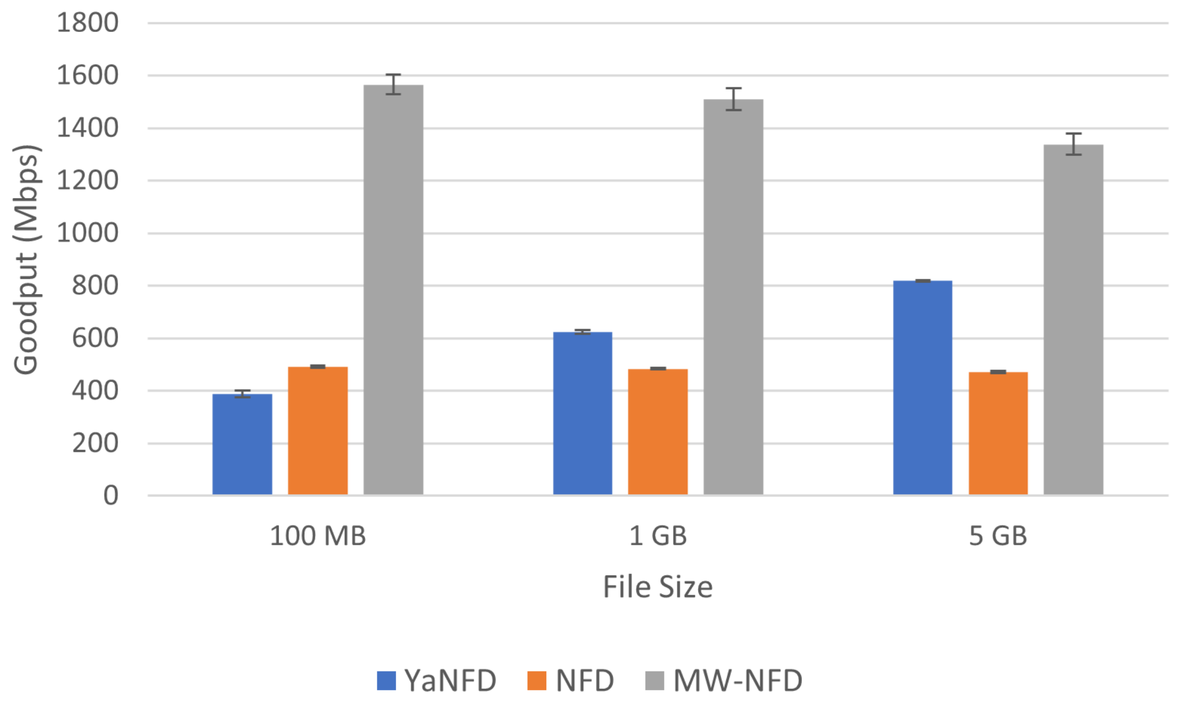


Figure 6.7: Application-layer throughput of file transfers comparing YaNFD, NFD and MW-NFD

ment, such as wireless routers, one needs familiarity with command line tools to configure an NDN node. For example, at the present time, NFD is generally configured using a combination of a configuration file and command line tools. Upon startup, NFD automatically loads a configuration file, which provides "startup configuration". Meanwhile, the remaining portion of the forwarder configuration (the "runtime configuration") is provided through the `nfdc` command line utility. Additionally, the `ndnsec` command line utility is used to manage the NDN security configuration of the local host. An earlier effort was made to create a graphical management interface for NFD [Nama]; however, the result was platform-specific and did not keep up with the pace of the NDN platform's development. However we do note the existence of a standardized management protocol [Namc], which allows NFD, and any other forwarder supporting this protocol, to be configured over native NDN protocols.

Therefore, we developed the NDN Forwarder Manager (NDN-FM), which offers a graphical, browser-based management interface for NDN forwarders. Utilizing a browser-based interface provides significant benefits over a standalone program, including simplified deployment and cross-platform support. NDN-FM allows end users to perform a number of operations critical to configuring and managing a local instance of a forwarder, including: monitoring the status of the forwarder; creating, updating, and deleting faces and routes; managing certificates; and running basic debugging tools.

6.5.2 Design and Implementation

6.5.2.1 Design Goals

The central goal of NDN-FM is to provide a usable, general, and flexible management solution for NDN forwarders. This is because, while at the moment NDN deployments are limited to research and development environments, they will eventually roll out into end-user environments. End users will likely need to configure and manage their local forwarders to at least some extent, as is commonly necessary for wireless routers in home and small business environments. NDN-FM is intended to be used directly by end users with varying levels of technical knowledge, so it must satisfy the three principles stated above. More specifically, it satisfies each principle like so:

- *Usability.* NDN-FM integrates frequently used forwarder management functions into a browser-based GUI. This relieves users of the burden of having to remember the `nfdc` commands needed to perform various NFD forwarder management operations. Instead, the user interface is organized into reasonable categories that allow users to quickly find the operation they wish to perform. Moreover, NDN-FM is easy to install and run, being based upon standard, cross-platform Python libraries.

- *Generality.* NDN-FM supports all operating systems that can run NFD and Python 3. Since it controls NFD via the OS-independent NFD Management protocol [Namc], no operating system-specific differences are presented to the user.

- *Flexibility.* The NFD Management protocol provides a standardized mechanism through which a wide variety of management operations can be performed. Therefore, NDN-FM implements a number of management operations of varying levels of complexity and frequency of use. This includes common operations like route and face configuration, as well as more advanced operations like security management.

In future widely-deployed NDN networks, forwarders on each device will serve as a gateway for application connectivity to the wider network and perform a very similar role to that of today's home wireless routers. This not only matches the level of configuration required in today's IP networks, but goes beyond it, as packet forwarders must be configured on every end devices in addition to on intermediate network hardware. Therefore, similar tools to those available for wireless routers today must be developed for NDN forwarders that satisfy the above-listed requirements.

6.5.2.2 System Overview

NDN-FM provides a browser-based graphical user interface to interact with users and to process user queries and commands. The user's browser communicates with the NDN-FM backend using HTTP. Meanwhile, the backend communicates with the forwarder (e.g., NFD) using the NDN-based NFD Management protocol [Namc] to query the status of the forwarder and perform various management operations. This design is shown in Figure 6.8.

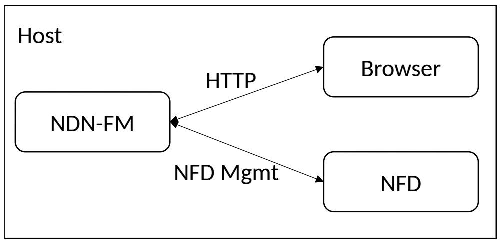


Figure 6.8: NDN Forwarder Manager Design Overview

NDN-FM provides an icon on the system tray, which is used as a shortcut to the user interface and to allow the user to quit the application. With NDN-FM, users can view, create, and delete faces and routes, as well as set strategies for given name prefixes. Additionally, users can modify the local security keychain, including adding and deleting identities, keys, and certificates. Moreover, users can view the status of the forwarder (including many counters), perform basic reachability tests, and even automatically connect to the nearest NDN Testbed node.

6.5.2.3 Implementation

NDN-FM is implemented in Python 3, using `python-ndn` [MKN] and `aiohttp` to realize the NDN-based management and user-facing HTTP interfaces, respectively. NDN-FM consists of two modules: a backend module and a system tray module. The backend module processes user input and obtains status information from the forwarder. Meanwhile, the system tray module provides an icon in the system tray.

The backend module consists of two components, an `aiohttp` server and a `python-ndn` application. The aiohttp server responds to HTTP requests and translates between HTTP's request-response paradigm and NDN's Interest-Data paradigm. The `python-ndn` component expresses these Interests and waits for corresponding Data. It also subscribes to face status change notifications from the NDN Management Protocol [Namc] and records such events.

6.6 PythonRepo

This section introduces my work on PythonRepo, an NDN data storage written in Python. The major part of this section is adapted from [YKM24]. I am one of the main designer and the current maintainer of this work.

6.6.1 Background and Motivation

In NDN applications fetches named, secured data packets that can be served from anywhere. This design enables *asynchronous* communication, potentially among multiple parties. Given not all data producers may be online all the time, persistent in-network data repositories [Zha19], or *repo*s for short, are designed to meet the goal of making all data available all the time, similar to servers in a TCP/IP network being online all the time. However, different from traditional cloud servers, an NDN repo is a generic infrastructure independent of specific applications.

The main features and design goals of NDN repos are

- **General-purpose:** a repo is a general-purpose networking service, not designed for specific application and not operated by application teams. A repo does not run application logic. It is also not necessary for a repo to execute the application's trust policies.
- **Application agnostic in security:** a repo is not trusted in the application's trust

domains. It has its own trust domain with users who authenticate to the repo to publish data into it, and the repo's trust domain is not related to the application's trust domain. A repo is not allowed to produce any data for the application.

- **Resilient to failures:** when application users cannot be online simultaneously, repos become always online rendezvous points that make data available, *i.e.,* repos substitute the role of application servers in today's Internet. Therefore, repo should be designed as a distributed system to enhance data availability.

- **Application oblivious:** the application and users do not depend on the existence or knowledge of repos to work. They are supposed to fetch desired data as usual, not sensing the existence of repos.

6.6.2 Design Overview

PythonRepo runs as an application process on nodes with storage resources. In NDN application data objects are of various sizes, with each object may be segmented to multiple Data packets. Therefore, PythonRepo uses segmented objects as the basic data unit in its operations.

PythonRepo takes segmented object insertion and deletion requests from the application and perform corresponding tasks. The user application initiates such insertion or deletion process by notifying PythonRepo there is a new request to be processed. Upon receiving the request, PythonRepo checks whether the request is produced by an authorized User through validating the request with the bootstrapped trust schema. If the request is signed by an authorized User, PythonRepo proceeds to fetch the segmented object from the network with the information provided within an insertion request, or delete the segmented object from its local storage for a deletion request. After sending an segmented object Insertion Request to PythonRepo, the User can optionally check whether the segmented object is ready for the Consumer to retrieve. When PythonRepo is ready to serve the segmented object, Consumers can fetch individual segments of the segmented object as fetching normal Data packets from the network.

6.6.3 Protocol Demonstration

In the rest of this paper, we use an example to demonstrate PythonRepo's protocol design. Assuming user Alice "`/ntt/alice`" pushes some sensor data to a PythonRepo named "`/ntt/repo`", which is deployed by her ISP NTT.

6.6.3.1 Inserting Segmented Objects

In order to insert data to PythonRepo, Alice first prepares an Insertion Request that informs "`/ntt/repo`" *what are the segmented object names.* The naming convention of the request is "`/<user-prefix>/<repo-prefix>/<operation>/<param-digest>`". The prefix "`<user-prefix>`" and "`<repo-prefix>`" are the User prefix and PythonRepo prefix, respectively, and "`<operation>`" represents the operation name which is "`insert`" for insertion. The name component "`<param-digest>`" is a SHA-256 hash of the request parameters. As shown in Figure 6.9, an insertion request includes the name or name prefix of data objects, and the starting and ending segment numbers if applicable.[4] Finally, Alice signs the request with the private key "`/ntt/alice/KEY`".

After preparing the request, Alice initiates the object insertion process by first expressing a notification Interest $I1$ to PythonRepo's insertion prefix "`/repo/insert`" with the application parameters carrying Alice's prefix "`/edu/ucla/alice`" and a request nonce "`123`" (`0x7B`), which is a random number to avoid confliction and identify this request.

On receiving $I1$, PythonRepo learns Alice's prefix and the nonce "`123`" that uniquely identifies her request, expresses Interest $I2$ to fetch Alice's ADU Insertion Request $D2$, and

[4]The protocol described here is a simplified version to explain how the design works. The actual implementation allows the user to specify multiple objects in one request. Please refer to the PythonRepo documentation for details.

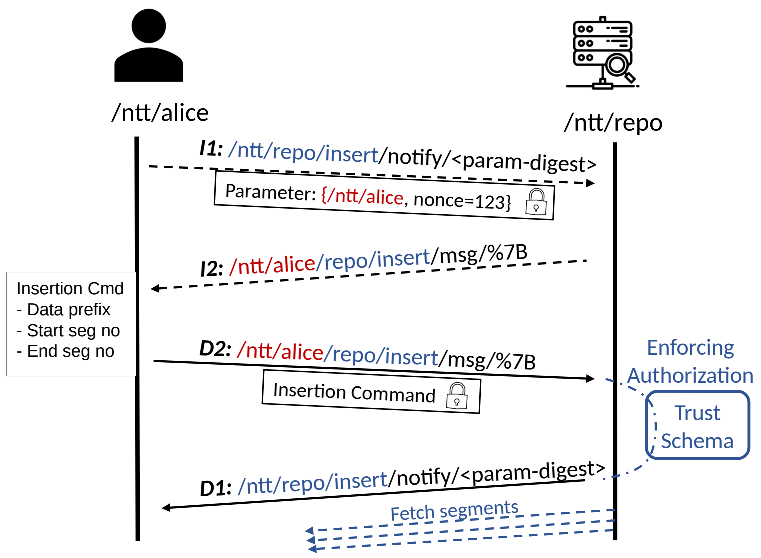

/ntt/alice
/ntt/repo
I1: /ntt/repo/insert/notify/<param-digest>
Parameter: {/ntt/alice, nonce=123}
I2: /ntt/alice/repo/insert/msg/%7B
Insertion Cmd
- Data prefix
- Start seg no
- End seg no
D2: /ntt/alice/repo/insert/msg/%7B
Insertion Command
Enforcing Authorization
Trust Schema
D1: /ntt/repo/insert/notify/<param-digest>
Fetch segments

Figure 6.9: Alice sends an insertion request to PythonRepo

validates the request's authenticity and legitimacy using its trust schema. For example, if the trust schema allows keys under the prefix "`/ntt/<user>`" to be the legitimate signers for Data under the prefix "`/ntt/<user>/repo/insert`", then Alice is authorized by the trust schema, thereby a legitimate User to insert objects in PythonRepo. If the request validation succeeds, PythonRepo replies to $I1$ with $D1$ with empty content, and begins fetching the ADU that Alice has requested to insert.

6.6.3.2 Checking Request Results

Since PythonRepo processes requests asynchronously, it needs to provide a mechanism for its Users to check whether an insertion request has succeeded (i.e., all inserted objects have become available), or if the request has failed due to ADU fetching failure[5], unauthorized requests, or full storage.

To this end, PythonRepo allows the Users to check the status of requests using com-

[5]PythonRepo will perform basic retransmissions up to a certain number of times to overcome packet losses.

mands under the prefix "`/<repo-prefix>/<operation> check`", where the command name "`<operation>`" is the same as the request, *i.e.*,"`insert`" for insertion command requests.

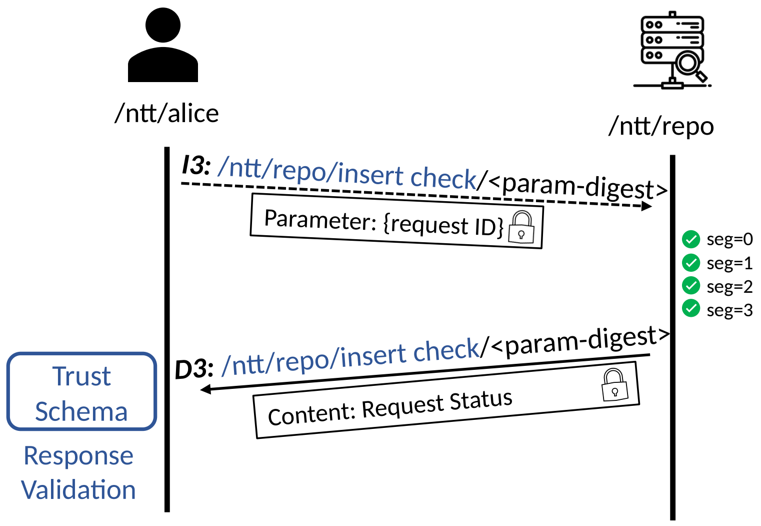


Figure 6.10: Checking insertion request Status

In the example as shown in Figure 6.10, Alice checks the insertion request's execution sattus by expressing an Interest $I3$ following the aforementioned naming convention. As PythonRepo receives $I3$, it checks its local database whether all segments specified by the request are inserted, and returns the signed execution status code in $D3$. Upon receiving $D3$, Alice validates it using its trust schema to ensure the data is indeed produced by PythonRepo. Therefore, if Alice's insertion request triggers errors, she is able to learn the reason from the status code.

6.6.4 Joining SVS Sync

Besides passively listening to the insertion commands, the repo can also join the Sync group and listen to Sync Interests. That is, the repo caches the latest Sync Interests of a sync group representing the latest state of the Sync group, which makes it available to any user who comes online. Whenever Repo receives Sync Interests carrying outdated state vectors, it replays cached Sync Interests to notify the network about the group's latest state.

The Repo relies on replaying state vectors from its cached set to communicate a contemporaneous state because the repo itself is agnostic to application security and not allowed to produce a new Sync Interest. It determines the highest sequence number from each member, and sends a set of Sync Interests from its cache whose union includes the highest sequence number for each member. When the repo becomes aware of new data through a more recent vector, it sends a data interest packet to participants and stores the data in order to respond to future interests.

In the worst case, the set of latest Sync Interests can have as many Interests as the number of members in the group. However, the size of this set is practically supposed to be small. This is because the repo is supposed to be online all the time, and a member in the sync group is supposed to publish a sinle latest Sync vector. Therefore, wheneven a member becomes online, it is supposed to reduce the set size to 1.

6.6.5 Deployment

PythonRepo is implemented in Python. Since 2020, PythonRepo has been used in multiple projects, including smart home data storage [ZYM22], power plant sensor data management, and the NDN Workspace. The NDN Testbed also has globally deployed PythonRepo instances on each site, serving as in-network storage for NDN applications.

CHAPTER 7

Conclusion and Future Work

In this dissertation, I compared the efforts towards decentralized collaborative applications. I defined decentralization as the ability of users to control the data generated by themselves. By identifying the three functionalities of the cloud: rendezvous point, security verification, and serving the authentic copy of data. I clarified the two tasks of decentralization: naming (including the verification of named entities), and data dissemination. I analyzed the relationship between application namespaces and security. Based on the lessons learned, I suggested a decentralized system use semantic namespaces, user-controlled accounts, and data-centric security. Guided by these principles, I designed NDN Workspace, the first practical decentralized collaborative editing application over NDN. I deployed this application to NDN testbed and showed that NDN Workspace satisfies the requirements of functionalities: providing offline availability, collaborating via any communication channel (local or global), and editing both real-time and asynchronously.

NDN Workspace utilizes CRDT and inherits the eventual consistency model (BASE) CRDT provides. I discussed the limitations of its automatic conflict resolution and showed remediation methods in the collaborative editing scenario. NDN Workspace also has its own limitations, including scalability concerns, missing snapshots of the current state, immature failure recovery, and application-coupled namespace. Ongoing research is being performed on these issues.

Applications are important driving power to an ecosystem. NDN Workspace is the first real-world application that is deployed and used daily, which gives us a chance to review all building blocks of NDN. We have figured out several issues and bugs in existing libraries, and developed a deeper understanding of our designs.

In the future, we plan to continue the development of NDN Workspace in two directions: as a collaborative user tool, and as a platform to explore the design space of NDN security and synchronization.

REFERENCES

[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. `http://named-data.net/techreports.html`.

[Ber20] Berners-Lee, Tim. "The Flanders Government and Solid: "An important milestone in Flemish history".", 2020. `https://www.inrupt.com/blog/flanders-solid`.

[BLS20] Sung Hyuk Byun, Jongseok Lee, Dong Myung Sul, and Namseok Ko. "Multi-Worker NFD: An NFD-Compatible High-Speed NDN Forwarder." In *Proceedings of the 7th ACM Conference on Information-Centric Networking*, ICN '20, pp. 166–168, New York, NY, USA, 2020. Association for Computing Machinery. `https://doi.org/10.1145/3405656.3420233`.

[Bre00] Eric Brewer. "Towards robust distributed systems." In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, p. 7, 01 2000.

[Bre12] Eric Brewer. "CAP twelve years later: How the "rules" have changed." *Computer*, **45**(2):23–29, 2012.

[BRS24] Annabelle Backman, Justin Richer, and Manu Sporny. "HTTP Message Signatures." RFC 9421, February 2024. `https://www.rfc-editor.org/info/rfc9421`.

[BZ23] Tim Berners-Lee and Hadrian Zbarcea. "Solid Chat: Version 1.0.0, Editor's Draft, 2023-08-21." W3C Recommendation, August 2023. `https://solid.github.io/chat/`.

[CBV22] Sarven Capadisli, Tim Berners-Lee, Ruben Verborgh, and Kjetil Kjernsmo. "Solid Protocol: Version 0.10.0." W3C Recommendation, December 2022. `https://solidproject.org/TR/2022/protocol-20221231`.

[Cer24] Certik. "Hack3d: The Web3 Security Report 2023.", 2024. `https://www.certik.com/resources/blog/7BokMhPUgffqEvyvXgHNaq-hack3d-the-web3-security-report-2023`.

[DFK18] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. "Observations on typing from 136 million keystrokes." In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pp. 1–12, 2018.

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

[GKR11] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. "Naming in content-oriented architectures." In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking*, ICN '11, pp. 1–6, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2018584.2018586.

[Gra21] Jay Graber. "Ecosystem Review." Technical report, Bluesky, 2021. Archived at https://perma.cc/RJ2Y-H6YT. https://gitlab.com/bluesky-community1/decentralized-ecosystem.

[Guy17] Amy Guy. "Social Web Protocols." W3C Working Group Note, December 2017. https://www.w3.org/TR/social-web-protocols/.

[Jah24] Kevin Jahns. "Yjs: Shared data types for building collaborative software.", 2024. https://yjs.dev/.

[JBS15] Michael B. Jones, John Bradley, and Nat Sakimura. "JSON Web Signature (JWS)." RFC 7515, May 2015. https://www.rfc-editor.org/info/rfc7515.

[Jef24] JeffG. "What are Nostr Relays?", 2024. https://nostr.how/en/relays.

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

[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.

[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*, pp. 91–102, 2022.

[MGA18] Spyridon Mastorakis, Peter Gusev, Alexander Afanasyev, and Lixia Zhang. "Real-time data retrieval in named data networking." In *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*, pp. 61–66. IEEE, 2018.

[MKN] Xinyu Ma, Zhaoning Kong, and Eric Newberry. “python-ndn.” `https://github.com/zjkmxy/python-ndn`.

[MM02] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.” In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pp. 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[MNZ21] Xinyu Ma, Eric Newberry, and Lixia Zhang. “NDN Forwarder Manager: Improving the Usability of NDN Forwarders.” *Named Data Networking, Tech. Rep. NDN-0070*, 2021.

[MPS21] Philipp Moll, Varun Patil, Nishant Sabharwal, and Lixia Zhang. “A Brief Introduction to State Vector Sync.” *Named Data Networking, Tech. Rep. NDN-0070*, 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`.

[MZ21] Xinyu Ma and Lixia Zhang. “GitSync: Distributed version control system on NDN.” In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, pp. 121–123, 2021.

[Nak08] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System.”, 2008.

[Nama] Named Data Networking. “NDN Control Center.” `https://github.com/named-data/NDN-Control-Center`.

[Namb] Named Data Networking. “NDN Packet Specification.” `https://named-data.net/doc/NDN-packet-spec/current/`.

[Namc] Named Data Networking. “NFD Management Protocol.” `https://redmine.named-data.net/projects/nfd/wiki/Management`.

[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*, pp. 112–122, 2019.

[Nic21a] 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`.

[Nic21b] Kathleen Nichols. "Trust schemas and ICN: key to secure home IoT." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, pp. 95–106, 2021.

[Nic22] Kathleen Nichols. "The VerSec Trust Schema Compiler.", 2022. `https://github.com/pollere/DCT/blob/main/tools/compiler/doc/language.pdf`.

[NJD16] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Near real-time peer-to-peer shared editing on extensible data types." In *Proceedings of the 2016 ACM International Conference on Supporting Group Work*, pp. 39–49, 2016.

[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*, pp. 30–41, 2021.

[OR27] Charles Kay Ogden and Ivor Armstrong Richards. *The Meaning of Meaning: A Study of the Influence of Language upon Thought and of the Science of Symbolism.* Harcourt, Brace, 1927.

[PBC24] Bluesky PBC. "DID PLC Method (did:plc).", 2024. `https://github.com/did-method-plc/did-method-plc`.

[Plu24] Plume contributors. "Plume: A federated blogging application.", 2024. `https://joinplu.me/`.

[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`.

[Pub94] FIPS Pub. "Security requirements for cryptographic modules." *FIPS PUB*, **140**:140–2, 1994.

[RJC19] Aravindh Raman, Sagar Joglekar, Emiliano De Cristofaro, Nishanth Sastry, and Gareth Tyson. "Challenges in the decentralised web: The mastodon case." In *Proceedings of the internet measurement conference*, pp. 217–229, 2019.

[SLS22] Manu Sporny, Dave Longley, Markus Sabadello, Drummond Reed, Orie Steele, and Christopher Allen. "Decentralized Identifiers (DIDs) v1.0." W3C Recommendation, July 2022. `https://www.w3.org/TR/did-core/`.

[The24a] The go-ethereum Authors. "Hardware requirements.", 2024. `https://geth.ethereum.org/docs/getting-started/hardware-requirements`.

[the24b] thekinrar. "Mastodon instances.", 2024. `https://instances.social/`.

[TLM19] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. "Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications." In *Proceedings of the 6th ACM conference on information-centric networking*, pp. 1–11, 2019.

[TPS23] Sankalpa Timilsina, Davide Pesavento, Junxiao Shi, Susmit Shannigrahi, and Lotfi Benmohamed. "Capture and Analysis of Traffic Traces on a Wide-Area NDN Testbed." In *Proceedings of the 10th ACM Conference on Information-Centric Networking*, ACM ICN '23, p. 101–108, New York, NY, USA, 2023. Association for Computing Machinery. `https://doi.org/10.1145/3623565.3623707`.

[VVC22] Melanie Verstraete, Sofie Verbrugge, and Didier Colle. "Solid: Enabler of decentralized, digital platforms ecosystems." *digital platforms ecosystems*, 2022.

[WCL20] Christopher Webber, Pierre-Antoine Champin, and Dave Longley. "JSON-LD 1.1: A JSON-based Serialization for Linked Data." W3C Recommendation, July 2020. `https://www.w3.org/TR/2020/REC-json-ld11-20200716/`.

[WT18] Christopher Webber and Jessica Tallon. "ActivityPub." W3C Recommendation, January 2018. `https://www.w3.org/TR/2018/REC-activitypub-20180123/`.

[WT24] Yiluo Wei and Gareth Tyson. "Exploring the Nostr Ecosystem: A Study of Decentralization and Resilience." *arXiv preprint arXiv:2402.05709*, 2024.

[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`.

[YKM24] Tianyuan Yu, Zhaoning Kong, Xinyu Ma, Lan Wang, and Lixia Zhang. "Python-Repo: Persistent In-Network Storage for Named Data Networking." In *Proceedings of 2024 International Conference on Computing, Networking and Communications (ICNC): Next Generation Networks and Internet Applications*, pp. 927–931, 2024.

[YMX22] Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Yekta Kocaoğullar, and Lixia Zhang. "Intertrust: establishing inter-zone trust relationships." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pp. 180–182, 2022.

[YMX23a] 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*, pp. 46–54, 2023.

[YMX23b] Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Dirk Kutscher, and Lixia Zhang. "Cornerstone: Automating Remote NDN Entity Bootstrapping." In *Proceedings of the 18th Asian Internet Engineering Conference*, pp. 62–68, 2023.

[YMZ21] Tianyuan Yu, Philipp Moll, Zhiyi Zhang, Alexander Afanasyev, and Lixia Zhang. "Enabling Plug-n-Play in Named Data Networking." In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, pp. 562–569, 2021.

[YZL23] Peng Yin, Shiqi Zhao, Haowen Lai, Ruohai Ge, Ji Zhang, Howie Choset, and Sebastian Scherer. "Automerge: A framework for map assembling and smoothing in city-scale environments." *IEEE Transactions on Robotics*, 2023.

[ZAB14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. "Named data networking." *SIGCOMM Comput. Commun. Rev.*, **44**(3):66–73, jul 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, 09 2017.

[Zha19] Lixia Zhang. "The role of data repositories in named data networking." In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–5. IEEE, 2019.

[ZVM19] Zhiyi Zhang, Vishrant Vasavada, Xinyu Ma, and Lixia Zhang. "Dledger: An iot-friendly private distributed ledger system based on dag." *arXiv preprint arXiv:1902.09031*, 2019.

[ZYM22] Zhiyi Zhang, Tianyuan Yu, Xinyu Ma, Yu Guan, Philipp Moll, and Lixia Zhang. "Sovereign: Self-contained smart home with data-centric network and security." *IEEE Internet of Things Journal*, **9**(15):13808–13822, 2022.