PythonRepo: Persistent In-Network Storage for Named Data Networking

Tianyuan Yu*, Zhaoning Kong†, Xinyu Ma*, Lan Wang‡, and Lixia Zhang*
*UCLA
†Purdue University
‡The University of Memphis
Email: *{tianyuan, xinyu.ma, lixia}@cs.ucla.edu, †kong102@purdue.edu, ‡lanwang@memphis.edu,

***Abstract*—Named Data Networking (NDN) takes a data-centric design approach to data delivery, which intrinsically enables asynchronous communication. That is, communicating entities can exchange data effectively even when they are not directly connected or online at the same time, as long as everyone can receive all its requested data. NDN makes data available through persistent in-network data repository, or *repo* for short, which is an integral component in the NDN architecture. In this paper, we first articulate the important role repos play in an NDN network, and then present the design of a simple repo protocol, PythonRepo, which has been used in several NDN applications. We also identify remaining work to be done to make PythonRepofullfil the needs of future NDN applications.**

I. Introduction

Today's Internet applications, by and large, are built on the client-server model over TCP/IP protocol stack. TCP/IP networking provides point-to-point connectivity to support the client-server applications through synchronous communication (i.e., both parties are online at the same time). While clients may come and go at any time, application servers must be online all the time, ready to serve clients whenever needed.

Named Data Networking (NDN) takes a data-centric design approach [1]. Its basic communication primitive is fetching named, secured data packets. This design enables *asynchronous* communication, potentially among multiple parties. These parties may or may not be directly connected with each other (i.e., having working paths between them), or even all online at the same time. They can communicate effectively as long as each can fetch its desired data whenever needed. Given not all data producers may be online all the time, persistent in-network data repositories [2], 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.

Up to now, however, not enough attention has been paid to the repo design and development. Although several repo prototypes have been developed over the years to meet application needs, there is little documentation on their designs, let alone systematic examination of their design choices to gather the lessons learned.

This paper is an effort to help fill that void. We make three contributions. First, we clarify the fundamental differences between NDN repos and today's cloud storage. Second, we describe the design and implementation of PythonRepo, one of the existing NDN repos that provides secure in-network storage to support NDN applications. Third, we identify the remaining work to be done with the current PythonRepo implementation to strengthen its resiliency and availability.

The remainder of this paper is organized as follows. §II provides an NDN overview, and highlights the differences between networked storage systems in NDN and today's cloud storage services. §III discusses the design goals of PythonRepo and how our design achieves the goals. Afterwards, We describe our initial implementation of PythonRepoin §IV, discuss the remaining work to be done for PythonRepo in §V, and conclude the paper in §VI.

II. Background

A. Named Data Networking

Instead of translating application layer names to IP addresses for packet delivery as the Internet works today, NDN directly uses application layer data names in network communication. Data consumers request data by putting the names in NDN *Interest* packets, and in response, the network returns the requested *Data* packets with the matching semantic name and cryptographic signatures, which are then used by consumers to authenticate the received data.

To check the authenticity of received Data packets, NDN lets each application define a set of trust rules, called *trust schema* [3], written in a defined schematic language. Because today's network security solutions are patched on top of TCP/IP's node-centric protocol stack which offers end-to-end reliable data delivery connections, they authenticate application servers b manually configured certificates to secure the connections. Therefore, they do not support elaborated security policies or fine-grained control over data. In contrast, NDN's trust schema enables applications to manage the trust relationships among multiple entities, where each entity can be an application process or any communication participant that produce or consume data. Trust schema defines which cryptographic key, which also has a semantically meaningful name, should sign which specific named Data packets.

In order to perform the above functions, each NDN entity must go through a bootstrapping process [4][5][6] first. We consider that all entities under the control of the same administrator constitute a *trust domain* [7], and each entity obtains the

979-8-3503-7099-7/24/$31.00 ©2024 IEEE

following parameters from the bootstrapping process: (i) the trust domain's self-signed certificate as its *trust anchor*, (ii) the trust schema, and (iii) its own identity certificate. Note that an individual user, say Alice, can make a trust domain for her self, D_{Alice}, e.g. having Alice's phone holds a self-signed certificate as her trust anchor. If Alice possesses additional devices, e.g. a laptop in addition to the phone, and each device may run some apps, then D_{Alice} will contains multiple entities. Also note that each app is an NDN entity as it can produce and/or consume data, therefore it must go through a bootstrapping process as well before it can actively participate in an NDN system.

B. Networked Storage

NDN repos are application processes themselves running on the nodes with storage resources to provide persistent storage for other applications. Repos accept data insertions requests, fetch the named data objects from requesters and make data available. Repos are transparent to data consumers, which simply fetch desired data by names, without needing to know where the data come from.

Various questions have been raised regarding the differences between NDN repos and other types of in-network storage. First, various storage systems are deployed in the cloud and at edges in today's TCP/IP Internet. We point out that today's cloud storage services are built on top of TCP/IP's node-centric protocol stack. Given a TCP/IP network delivers data to IP addresses, application developers must handle the task of figuring out where to fetch a requested dataset. Content Distribution Network (CDN) services offer location-transparent service to end users by building application layer overlays, and they only serve a relatively small number of paid content providers.

In contrast, NDN integrates networking and storage, and enables *all* consumer applications to request data by name, without having to identify specific data containers or locations. An Interest packet can find and retrieve the requested data from the nearest location, be it from router cache, repo storage, or data producer.

One basic reason that NDN can fetch desired data from anywhere is its design of *securing data directly*. Data owners make their data authenticable by cryptographically signing them, and make the data confidential by encrypting them. This design puts (i) data access control in the hands of data owners, independently from data containers; and (ii) data authenticity validation in the hands of data consumers, independently from communication channels. Because security is attached to data, data replication is also made easy. In contrast, the security of cloud storage relies on TLS connections between user nodes and cloud servers, and the security of data is bundled with servers. This makes data replication complex to handle, as one must ensure trust on all replicas.

Second, within the NDN context, the content store at each NDN router is already a form of in-network storage. Although both router content store and repos can store NDN Data packets, a content store caches passing-by Data packets *opportunistically*, Data packets can be evicted due to resource constraints, and thus does not ensure data availability. In contrast, repos are *managed* in-network storage system, which ensure Data packets availability until data are evicted upon request by their applications. To provide resilient data availability in face of failures, repos should also replicate all stored data in multiple servers.

III. Design of PythonRepo

In this section, we first define the basic operations of PythonRepo, then describe our design assumptions and goals. Afterwards, we give an overview of PythonRepo workflow, followed by the PythonRepo operations details.

PythonRepo Operations: PythonRepo runs as an application process on nodes with storage resources. It interacts with *users*, a generic term we use to refer to NDN entities that utilize repos by inserting or deleting data objects.

An observation we make from existing NDN applications, such as those described in [8][9][10][11], produce application data objects of various sizes, each object may be segmented to multiple Data packets. We refer application data object as Application Data Unit (ADU) [12]. PythonRepo uses ADU as the basic data unit in its operations.

Design Assumptions: We assume that both Users and PythonRepo go through the NDN bootstrapping process before they start operations. Therefore, they possess necessary security parameters to secure as well as validate the data exchange between each other. Consumers express Interests to fetch desired data from the network. They validate received data following the security policies defined by their applications, independent from where the data is retrieved.

Design Goals: PythonRepohas the following two design goals:

- **User Authenticity and Authorization:** PythonRepo should accept ADU insertion and deletion requests from authenticated and authorized Users only.
- **ADU Availability:** after a User successfully inserts an ADU, PythonRepo should keep this ADU available persistently.

A. PythonRepo Overview

PythonRepo takes ADU insertion and deletion requests from the application and perform corresponding tasks. Since the request must carry the necessary ADU information that PythonRepo needs to know, it should be a piece of semantically named and secured data that PythonRepo fetches from the application. Therefore, it is the user application that initiates the ADU 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 ADU from the network with the information provided within an insertion request, or delete the ADU from its local storage for a deletion request. After sending an ADU Insertion

Request to PythonRepo, the User can optionally check whether the ADU is ready for the Consumer to retrieve.

When PythonRepo is ready to serve the ADU, Consumers can fetch individual segments of the ADU as fetching normal Data packets from the network.

B. PythonRepo in Operation

In the rest of this paper, we use an example to demonstrate PythonRepo's protocol design. Assuming the building manager Alice "`/edu/ucla/alice`" monitors offices with smart sensors. The monitor system produces sensor data every hour, and pushes sensor data to a PythonRepo named "`/repo`", to be fetched by a remote data analytics application.

Inserting ADU into PythonRepo: In order to insert data to PythonRepo, Alice first prepares an ADU Insertion Request that informs "`/repo`" *what are the ADU names and where to fetch ADUs*. The naming convention of the request is "`/<user-prefix>/<repo-prefix>/<operation>/msg/<nonce>`". 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 "`<nonce>`" is a 32-bit randomly generated number uniquely identifying the request. As shown in Figure 1, Alice puts two *Insertion Request Parameters* blocks into an ADU Insertion Request. Each parameter block includes the request description for an ADU[1]. The first block specifies the ADU prefix of "`/eng6/office365/humid/8am`". The block has segment number range of "0-3", indicating that this ADU has four segments in total and the segment number starts from zero; The forwarding hint "`/edu/ucla/alice`" instructs PythonRepo to fetch this ADU from Alice; The prefix registration field instructs PythonRepo to register the name prefix "`/eng6/office365/humid`" in order to serve this ADU to consumers. Finally, Alice signs the request with the private key "`/edu/ucla/alice/KEY`".

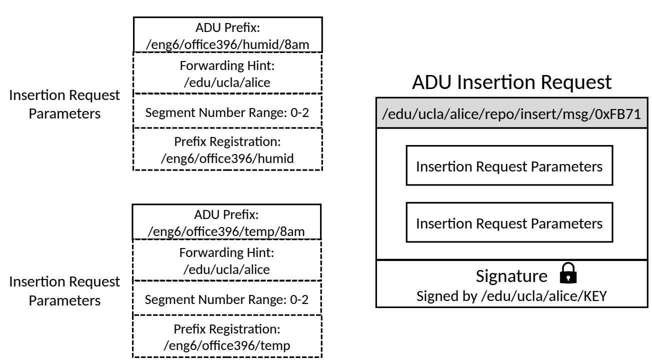

Figure 1. ADU Insertion Request and Insertion Request Parameters

After preparing the request, Alice initiates the ADU 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 request nonce "`0xFB71`".

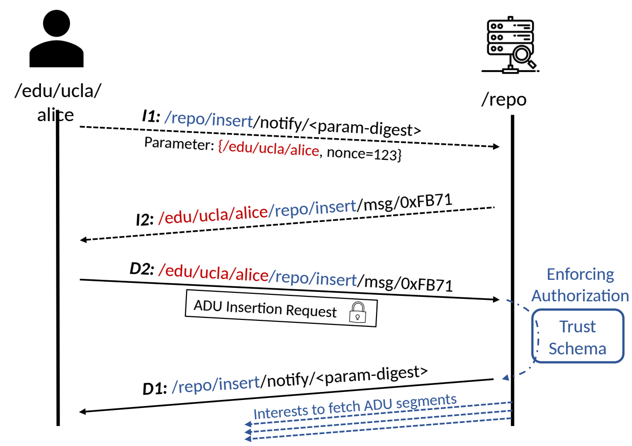

Figure 2. Alice sends Object Insertion Commands to PythonRepo

On receiving $I1$, PythonRepo learns Alice's prefix and the nonce "`0xFB71`" that uniquely identifies her request, expresses Interest $I2$ to fetch Alice's ADU Insertion Request $D2$, and validates the request's authenticity and legitimacy using its trust schema. For example, if the trust schema allows keys under the prefix "`/edu/ucla/<user>`" to be the legitimate signers for Data under the prefix "`/edu/ucla/<user>/repo/insert`", then Alice is authorized by the trust schema, thereby a legitimate User to insert ADUs 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.

Checking ADU Availability: 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 ADUs have become available), or if the request has failed due to ADU fetching failure[2], unauthorized requests, or full storage.

To this end, PythonRepo allows the Users to check ADU availability using commands under the prefix "`/<repo-prefix>/check/<adu-prefix>`", where the suffix "`<adu-prefix>`" is the ADU prefix Alice wants to check.

[1]Deletion Request Parameters follow a similar structure, but without the forwarding hint [13] and prefix registration fields.

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

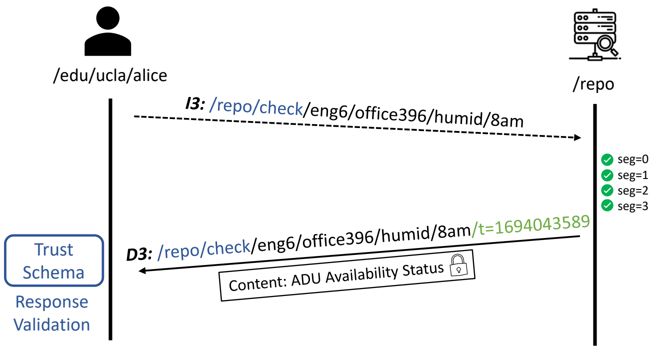

Figure 3. Checking ADU Availability Status

In the example as shown in Figure 3, Alice checks the ADU availability "`/eng6/office396/humid/8am`" by expressing an Interest $I3$ following the aforementioned naming convention. As PythonRepo receives $I3$, it checks its local database whether all segments of the ADU are available, and returns the signed ADU availability 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.

Fetching Data from PythonRepo Consumers send Interests to fetch individual ADU segments from PythonRepo in the same way as fetching Data packets from the original data producers. Specifically, Consumers express Interests that are attached a forwarding hint containing the PythonRepo prefix "`/repo`" to the Interests, so that the network is able to steer the Interests with the ADU Interest prefix to the repo.

Note that attaching forwarding hints to Interests is not encapsulation. Encapsulated packets do not benefit from NDN's in-network caching. However, even with forwarding hints, the original Interest name is still visible to the forwarders. Hence, when the Data packet matching the Interest name follows the reverse path back to the Consumers, it can be cached by intermediate routers and match other Interests with the same name.

This approach requires the original ADU producer (not necessarily the User who inserts the ADU to PythonRepo) to inform the Consumers of the PythonRepo prefix out-of-band, so that the Consumers can attach forwarding hints to Interests. We argue that applications are responsible for conveying the forwarding hints for Consumers.

IV. Implementation

PythonRepo is implemented in Python [14]. Since 2020, PythonRepo has been used in multiple projects, including smart home data storage [15], power plant sensor data management, and mobile health applications [8]. The NDN Testbed [16] also has globally deployed PythonRepo instances on each site, serving as in-network storage for NDN applications. The design and implementation of PythonRepo also inspired an ongoing project Hydra [9], which aims at providing federated storage for genetic researchers.

V. Discussion

Storage Management: PythonRepo has simple storage functions: insert and delete ADUs. Our past experiences indicate that the current two functions serve existing NDN applications adequately. As NDN applications get developed further, PythonRepo can benefit from having storage management functions, such as analyzing storage capacity, and monitoring new ADU insertions under specific prefixes.

Distributed PythonRepo: A frequently asked question is whether PythonRepo is designed as a single instance, and therefore susceptible to single point of failure. Although in this paper we introduced PythonRepo from the single instance's perspective, PythonRepo design is extensible to the distributed multi-instance case. Benefiting from NDN's built-in anycast, deploying a distributed PythonRepo is as easy as starting multiple PythonRepo instances that advertise the same name prefix to the routing system and running a synchronization protocol [17][18] among themselves. Each User's insertion requests will be routed to the closest instance, and then the ADU will be disseminated to all other instances in the synchronization group.

VI. Summary and Future Work

In-network storage is an important component in an NDN network to support peer-to-peer applications. In this paper, we first clarified the differences between NDN in-network storage and today's storage system, and then introduced the PythonRepo design which provides secure in-network ADU storage for NDN by semantically securing each request. We also explained how PythonRepo can be easily extended to a distributed design. In the future, we plan to add storage management function to PythonRepo, enable PythonRepo to join application synchronization groups to automatically replicate ADUs in a distributed manner, and explore the idea of distributed PythonRepo, especially on the deletion request handling across the system.

Acknowledgment

This work was supported in part by US National Science Foundation under awards 2019085 and 2126148.

References

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," pp. 66–73, 2014.

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

[3] Y. Yu, A. Afanasyev, D. Clark, K. Claffy, V. Jacobson, and L. Zhang, "Schematizing trust in named data networking," in *proceedings of the 2nd ACM Conference on Information-Centric Networking*, 2015, pp. 177–186.

[4] T. Yu, P. Moll, Z. Zhang, A. Afanasyev, and L. Zhang, "Enabling plug-n-play in named data networking," in *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, pp. 562–569.

[5] T. Yu, X. Ma, H. Xie, X. Jia, and L. Zhang, "On the security bootstrapping in named data networking," *arXiv preprint arXiv:2308.06490*, 2023.

[6] T. Yu, X. Ma, H. Xie, D. Kutscher, and L. Zhang, "Cornerstone: Automating remote ndn entity bootstrapping," in *The 18th Asian Internet Engineering Conference*, 2023.

[7] K. Nichols, "Trust schemas and icn: key to secure home iot," in *Proceedings of the 8th ACM Conference on Information-Centric Networking*, 2021, pp. 95–106.
[8] S. Dulal, N. Ali, A. R. Thieme, T. Yu, S. Liu, S. Regmi, L. Zhang, and L. Wang, "Building a secure mhealth data sharing infrastructure over ndn," in *Proceedings of the 9th ACM Conference on Information-Centric Networking*, 2022, pp. 114–124.
[9] J. Presley, X. Wang, T. Brandel, X. Ai, P. Podder, T. Yu, V. Patil, L. Zhang, A. Afanasyev, F. A. Feltus *et al.*, "Hydra–a federated data repository over ndn," *arXiv preprint arXiv:2211.00919*, 2022.
[10] J. Thompson, P. Gusev, and J. Burke, "Ndn-cnl: A hierarchical namespace api for named data networking," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, 2019, pp. 30–36.
[11] C. Ghasemi, H. Yousefi, and B. Zhang, "Internet-scale video streaming over ndn," *IEEE Network*, vol. 35, no. 5, pp. 174–180, 2021.
[12] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4, pp. 200–208, 1990.
[13] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "Nfd developer's guide," *Dept. Comput. Sci., Univ. California, Los Angeles, Los Angeles, CA, USA, Tech. Rep. NDN-0021*, vol. 29, p. 31, 2014.
[14] N. Team, https://github.com/UCLA-IRL/ndn-python-repo, 2023, accessed: 2023-5-27.
[15] Z. Zhang, T. Yu, X. Ma, Y. Guan, P. Moll, and L. Zhang, "Sovereign: Self-contained smart home with data-centric network and security," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 13 808–13 822, 2022.
[16] The NDN Team, "Ndn testbed," Online at https://named-data.net/ndn-testbed/, 2022.
[17] T. Li, Z. Kong, S. Mastorakis, and L. Zhang, "Distributed dataset synchronization in disruptive networks," in *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2019, pp. 428–437.
[18] P. Moll, V. Patil, L. Zhang, and D. Pesavento, "Resilient brokerless publish-subscribe over ndn," in *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 2021, pp. 438–444.