Repo: Application Agnostic and Oblivious In-Network Data Store

Tianyuan Yu
UCLA
USA
tianyuan@cs.ucla.edu

Jacob Zhi
UCLA
USA
jzhi@g.ucla.edu

Xinyu Ma
UCLA
USA
xinyu.ma@cs.ucla.edu

Yekta Kocaogullar
UCLA
USA
ykocaogullar@g.ucla.edu

Varun Patil
UCLA
USA
varunpatil@cs.ucla.edu

Ryuji Wakikawa
SoftBank
Japan
ryuji.wakikawa@g.softbank.co.jp

Lixia Zhang
UCLA
USA
lixia@cs.ucla.edu

***Abstract*—As a specific application within the broadly-defined realm of *metaverse*, NDN Workspace is a serverless collaborative tool that enables multiple users to jointly edit shared files. Our experience from NDN Workspace trial deployment suggests that such decentralized apps can benefit greatly from persistent data availability. This paper reports the design of Repo, a in-network replicated data store which can automatically stores application contents of authorized users. We illustrate how NDN Workspace's data-centric design, which is built on Named Data Networking (NDN), enables Repo to provide persistent data availability in an application agnostic and oblivious manner.**

I. Metaverse as a Network Problem

Metaverse is envisioned to build a shared, persistent virtual world around users. To create this world, various metaverse apps gather data from multiple sources, such as sensors, cameras, and other computing or storage resources, either nearby or far away. Individual virtual objects may belong to different parties with various control rights, yet they can all be accessed in a timely and secure way obeying user-defined object access rules. NDN Workspace [1] is such an application.

In this paper, we describe the design and implementation of *Repo*, an application agnostic and oblivious in-network data store to provide persistent data availability as needed by NDN Workspace and potentially other metaverse applications. The design of *Repo* draws lessons learned from previous efforts in developing in-network data store [2], [3], and utilizes identified design patterns of NDN-based applications such as NDN Workspace. The data-centric approach in these applications enables Repo to automatically fetch and store application data for authorized users, with little or no changes required for existing applications.

This paper is structured as follows. We describe the design and implementation of Repo in Sections IV and V, and summarize the insights we gained from this effort in Section VI. We further articulate the roles Repo may play in supporting future metaverse applications in Section VII. In particular, we note that Repo differs from today's cloud storage in fundamental ways: Repo is integrated with network data delivery and oblivious to applications. Application entities simply fetch any desired data by data names, without knowing or caring about where the data may come from.

II. Background

Designing a secure in-network data storage requires three basic building blocks: semantic entity and data identifiers for authentication, defined trust relations among all entities to secure storage usage, and new means of data exchange by data names instead of IP addresses.

In this section, we first provide a brief introduction to Named Data Networking (NDN) and explain how NDN provides these basic building blocks, and then use *NDN Workspace* application as an example to illustrate how NDN enables secure data fetching among all authenticated entities.

A. Named Data Networking

While IP views a network as a collection of nodes, each identified by its IP address and with physical connectivity in between, Named Data Networking (NDN) [4] views a network as a collection of entities[1], each uniquely identified by its semantically meanful name, with trust relations in between. To participate in an NDN network, an entity first goes through a bootstrapping process to obtain its trust anchors, semantic identifiers, certificates and security policies [5], [6] [2]. The bootstrapping step enables each entity to publish semantically-named data that is both encrypted[3] and signed by the entity's key, and verify received data according to its security policies. It should be noted that a producer simply makes its published data available locally, ready to be fetched by any interested data consumers. Additionally, a cryptographic key is also a piece of semantically named and secured data; a signed key becomes a certificate.

[1] An entity can be any communication participant that produces and/or consumes data: a device, a running app, a user, etc.

[2] As illustrated in [6], this NDN entity bootstrapping step is remotely analogous to IP bootstrapping when a node joins an IP network.

[3] Encryption in multi-user applications typically uses application-specific keys.

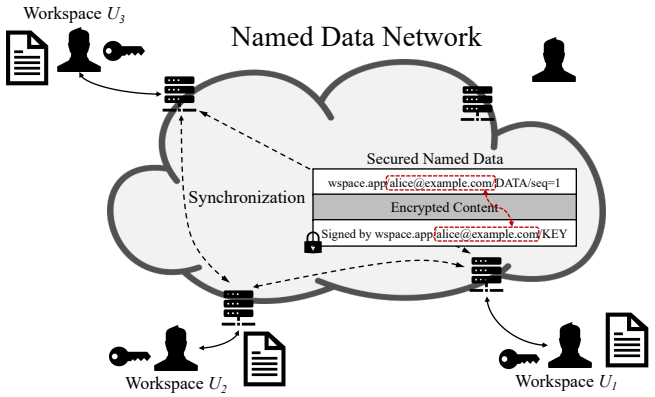

Fig. 1: NDN Workspace users synchronize on document updates in a named data network, exchanging semantically named and secured data.

In an NDN network, entities communicate by letting consumers send interest packets to request desired data by name. These Interest packets are routed towards data producers; any node along the way can reply to an interest with data having a matching name. Since the consumer verifies the authenticity of all received data as we describe below, the returned data does not have to come from its original producer. This allows each NDN node to cache data packets in its local memory, so it can respond to future requests for the same data from its cache. If a data packet is lost on its way to the consumer, the missing data can be retrieved from a router's cache when the consumer retransmits its interest. An NDN network also has built-in multicast data delivery support. We refer interested readers to [4] for more details.

Given certificates are also named data, one can define *schematized* security policies [7] to automate security workflow. Since all data are named and signed by their publishers, upon receiving data, the data consumer automatically fetches the data signer's certificate by its name, verifies the signature, and then checks the signer's certificate name to see whether the signer is authorized to publish the named data. As the example in Figure 1 shows, "`wspace.app/alice@example.com/DATA/seq=1`" identifies the first version of a piece of data produced by Alice in "`wspace.app`". The "`wspace.app`" administrator(s) could define security schema to authorize Alice to produce data under her own prefix in "`wspace.app`".

Given communication in an NDN network is by requesting named data, consumers need a means to learn the names of data as soon as they are published. NDN provides this functionality by its transport protocol, Sync (short for data synchronization), which synchronizes the names of the shared data among all participants in an application group, *e.g.,*"`wspace.app`" [8]. Sync multicasts *Sync Interests* to all participants; each Sync Interest carries a dataset state vector which encodes the latest sequence number of the data produced by each group participant. For example, whenever Alice produces a piece of new data to her app "`wspace.app`", Sync notifies all other participants about Alice's latest data production sequence number (which can be converted to an actual data name); individual entities can then decide whether, or when, they will fetch the new data. An offline user can catch up on the data produced by the rest of the group when it reconnects and learns about the names of new data it missed.

B. NDN Workspace

Current metaverse prototypes rely largely on cloud services, where end users and devices fetch data from cloud servers via secured transport connections, the servers not only can be far away, but also have the full control over all user data. To showcase a new way of networking that can meet metaverse's needs and keep the control of data in users hand, we developed a web-based, serverless, multi-user collaborative application, *NDN Workspace* [1], over the NDN networking model. Users in NDN Workspace are assigned semantic identifiers and establish security relations among each other, enabling them to collaborate *directly* and *securely* over any available physical or virtual connectivity.

Each NDN workspace instance has its own name, whose uniqueness comes from the existing DNS namespace. Under an assigned DNS name "`wspace.app`", a user can be uniquely identified by her email address, and assigned a name which is the concatenation of the app's DNS name and the user's email address, as shown in Figure 1, and is issued a certificate under that name.[4] NDN Workspace semantically names each document or document update that Alice produces as "`wspace.app/alice@example.com/DATA/<seq>`", where "`<seq>`" is the sequence number of Alice's data productions within "`wspace.app`".

Semantically named "`wspace.app`" users can mutually authenticate each other and endorse new users to join, bringing their out-of-band trust relations from human society into a workspace through cross-signed certificates and guided by the application-defined security policies. After going through the NDN bootstrapping process as described in the previous subsection, "`wspace.app`" users are able to securely produce and consume named data among themselves.

As Figure 1 shows, every document change made by a user increments the user's sequence number, and NDN Sync provides notifications to "`wspace.app`" workspace users of all new data being produced. Thus all users in the same workspace can be informed of the latest sequence numbers of all the other users. They can then fetch individually secured data pieces by sending NDN Interest packets to the network, with the data name inferred from the aforementioned naming convention. Receiving secured named data, each user independently executes security policies to validate the data and incorporate the updates to one's local view of the document.

Assigning users application defined semantic names and data-centric security design enable *direct user-to-user communications*. Since it is the users' browsers that store and manage

[4]A user can be uniquely identified by other existing identifiers, such as a DNS name, and the application assigned user name can be the concatenation of the app's name with any unique ID under that name. Since email addresses are unique, the current use of email address simply saves the app from keeping track its assigned names.

the application's data locally, NDN Workspace naturally supports asynchronous collaboration. Any user can make changes to their local document any time (even when offline); one's secured data will be synchronized once one can meet others in the same application group over any available connectivity (*e.g.,* LAN, Bluetooth, NDN Testbed [9]).

Our experiences from running NDN Workspace identified a critical need in supporting user-to-user communications: persistent data availability in the absence of servers. Because not all users may be online all the time, and in particular there can be no one online when a user gets connected and wants to fetch the latest changes made by others. Therefore, we developed the in-network data store, Repo, to meet this need.

III. Previous In-network Storage

PythonRepo [10] was developed by an earlier effort as in-network storage support to be used by NDN-based applications. As Figure 2 depicts, a network provider P offers in-network storage as part of P's network service and deploys multiple running instances of PythonRepo under its storage service prefix "/N_p/`store`". A customer of network P, C_P, can make use of the provided storage service by issuing data insertion and deletion requests (which are also encoded as named, secured data) to "/N_p/`store`". Leveraging anycast delivery, network-P will forward the request to a storage unit with the lowest routing cost (the cost could account for storage availability as well). Upon receiving a data insertion or deletion request, the storage unit validates the request according to P's security policies, and processes the request (either fetching the data to be inserted and storing it, or deleting the corresponding data from its local storage). The request carries the following parameters from C_P: the name of data to be inserted ("`/alice/photo/v1`"), and the name prefix "`/alice/photo`", which C_P wants PythonRepo to register to network-P.

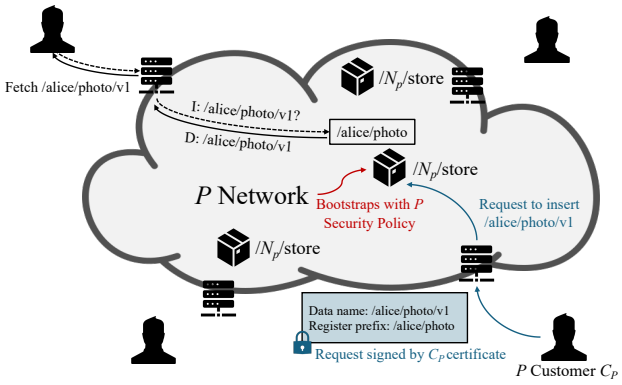

Fig. 2: Network provider P deploys multiple PythonRepo instances in its network, and each has security bootstrapped with P's security policy.

Note that PythonRepo, or NDN networking in general, makes data accessible by data names, e.g. "`/alice/photo`", anyone knowing the name can fetch it. Data-centric encription prevents unauthorized users from seeing the content. One may also note that PythonRepo does not provide automated data replication.

NDN's data-centric security design allows users to secure data directly, which in turn makes PythonRepo independent from specific applications. The data object "`/alice/photo/v1`" is secured by user's encryption and signing. Since users do not expose her security policies to PythonRepo, it has no knowledge about the data's contents. [5]

IV. Developing In-network Storage Support

In this section, we discuss the design goals of Repo and how they are achieved by leveraging NDN's data-centric design.

A. Design Goals

As a web-based application, NDN Workspace runs inside each user's browser, and communicates directly with other users (browsers). In order to enable a user U_1 to fetch and consume data published by another user U_f who has gone offline, the data produced by U_f must be made available online before U_f goes offline. This could be achieved if there are some users online all the time, but one cannot count on it.

To make all produced data available all the time without dependency on dedicated servers, one needs in-network data repository, Repo in short. We identify the following three design requirements.

- **Repo should be agnostic to applications.** Repo should provide storage for applications in general, with neither application-specific knowledge nor individual application's security credential that today's CDN services need.
- **Applications should be oblivious to Repo.** NDN applications exchange named, secured data without the awareness of where the data come from, a network model that must be preserved by the introduction of Repo.
- **Repo should be resilient to failures.** Repo should be able to provide resilient data availability that can sustain individual storage unit failures, so that applications can count their data availability as long as the network as a whole does not fail.

B. Repo Design Overview

As described in Section III, PythonRepo provides in-network, application-agnostic storage. However, applications need to send insertion requests to PythonRepo explicitly with *specific data names*. There is also a remaining question of how to *automatically* replicate stored data for resiliency against a single point of failure. The design of Repo extends the functionality of PythonRepo by automating both data insertion and data replication.

[5]The above simplified description of injecting user data prefixes into network routing may raise a scalability concern. NDN addresses this scalability challenge through the use of a *Forwarding Hint* which acts as a locator, pointing to the place where the named data can be found. Please see [11] for details, and [12] for an example usage of Forwarding Hints by another NDN-based application, Hydra.

Oblivious Sync Participation: As described in Section II, NDN applications utilizes NDN Sync to exchange the latest data production information among end-users. Therefore, Repo can learn the latest synchronization states of a workspace instance by joining the application's Syn group G to learn about all data productions. Whenever Repo learns there is a new publication in group G, it expresses Interest to fetch and store the latest published data D. Whenever an NDN Workspace user wants to fetch D, Repo can directly reply with D from its storage.

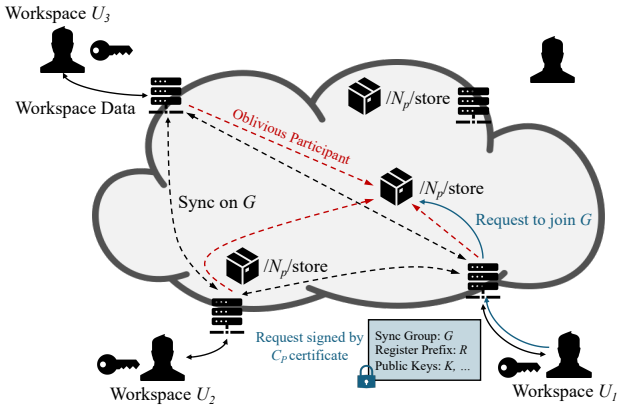

Fig. 3: In network P, three Workspaces are running NDN Sync over group G. Repo is an oblivious participant in G, as requested by U_1.

In this process, Repo is a passive participant in a Syng group – it neither generates Sync Interest nor produces new data packets, making the application unaware of the existence of Repo.

Requesting Repo Service: Any customer of a network P, referred as C_p, can make use of Repo provided by network P by sending a Repo service request to ask Repo to (i) join a synchronization group G; (ii) fetch all Data published in G; and (iii) serve them under a data name prefix R. The request contains the name of G, the data name prefix R, and all necessary certificates that can be used to verify application Data and Sync Interests (to be discussed shortly). Repo validates each received service request by the user's certificate issue by C_p, and starts joining synchronization group G. Requesting Repo to leave G follows a similar procedure.

Automated Replication: Repo units in network P runs a synchronization group on all the validated requests that have been received. After receiving a validated join or leave request for G, each unit executes the request independently, resulting in all the Repo instances storing all published application data automatically. One can also design an algorithm to make Repo automatically select n out of the total number of Repo units for replicating data of each application.

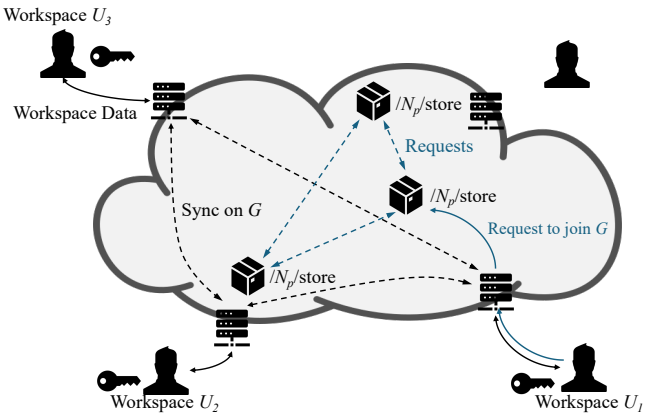

Fig. 4: Repo instances synchronize on received requests and process them independently.

As a result, Repo is resilient to single failures. Synchronizing Repo requests, rather than application data simplifies the design of Repo's resiliency by letting individual Repo units execute the request and fetch data, and data fetching also benefit from NDN's built-in multicast data delivery.

Routing and Data Security: Repo verifies the name ownership for data prefix registration, and only registers data prefixes with the routing system when customers of network P can prove the ownership of the name prefix. Therefore, all data prefix registrations Repo makes come from prefixes' original owners.

Repo operates according to the security policies installed by the network operators. Whenever receiving G Sync Interest and application Data published in it, Repo always match the signing key with authenticated keys obtained from the service request and verify the signature. Repo performs data authenticity verification when fetching published data to prevent storage pollution. The default authentication policy is to compare Data name with the signing key name to check whether the signer is the data prefix owner. Additionally, Repo can also take C_p defined specific application policies carried in the Repo service request to further refine the Data authentication verification. Note that, although Repo may execute application-specific policies, the policy execution is application-agnostic, since Repo only performs *schematized* naming-key pattern matching [13], [14] which is generic to all applications.

Supporting Asynchronous Applications: In addition to store all application produced data, to help newly arrived users quickly learn about the latest data production, Repo also keeps the latest Sync Interest it has received for each application group, which represents the latest data production state of an application. When Repo receives Sync Interests carrying outdated state vectors, which are likely produced by users who are just back online, it re-sends the latest Sync Interest to inform users of the latest data production that they may have missed when offline. Such Sync Interest retransmissions do not lead to replay attack, as receiving duplicate Sync Interest has no effect on user applications.

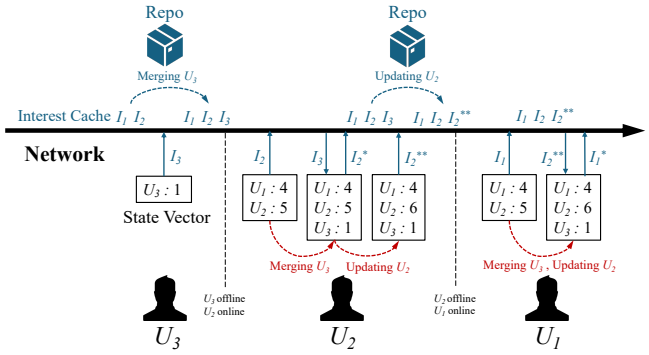

Fig. 5: An example where three Workspace users fully asynchronously collaborate by Repo updating Interest cache and replaying them upon updated Sync Interests.

The example in Figure 5 illustrates how Repo enables NDN Workspace to be fully asynchronous. Assuming there are three users collaboratively editing a document, and U_3 is a new user in G who is not aware of the other two, who are already synchronized with each other.

U_3 first connects to the network, makes a change to the document, and then disconnects. Its Workspace instance multicasts a Sync Interest I_3 to the network, indicating the existence of U_3 with sequence number 1. Repo adds I_3 into its cache since this Interest packet updates G's state, and expresses an Interest to fetch U_3's data "`/G/`U_3`/DATA/seq=1`". Later, U_2 comes online, it also multicasts its last Sync Interest I_2 to synchronize with the network, and receives I_3 obliviously replayed by Repo. Since this Interest carries a new entry of state vectors, U_2 merges the state vectors and sends a new Sync Interest I_2^* to the network. Afterwards, it fetches "`/G/`U_3`/DATA/seq=1`" as indicated by the latest vector, makes its change to the document, and sends I_2^{**} that contains updated state vectors to the network. Finally, when U_1 goes online, it will discover I_2^{**} with the latest state vectors, and fetches "`/G/`U_3`/DATA/seq=1`" and "`/G/`U_2`/DATA/seq=6`".

In this process, each user makes document changes independently and asynchronously. As there were no other instances or third-party servers users could synchronize with when publishing, each user instead obliviously synchronizes with Repo. When the application is connected to network P, NDN Sync states always inform everyone of the latest publications, as memorized by the network.

V. Implementation and Trial Integration

The initial implementation of Repo is in Python and utilizes the `python-ndn` library [15] for NDN communication and utilities. To achieve data persistence, Repo is reliant on a local or external data store, such as MongoDB, LevelDB, or SQLite. In addition to application data, the data store also keeps persistent state information for any Sync group the local browser has joined.

A. Trial Integration with NDN Testbed

Users running NDN Workspace can be NDN-connected over the NDN Testbed, which goes across four continents to interconnect all NDN-speaking nodes. We have deployed Repo integration on the NDN Testbed.

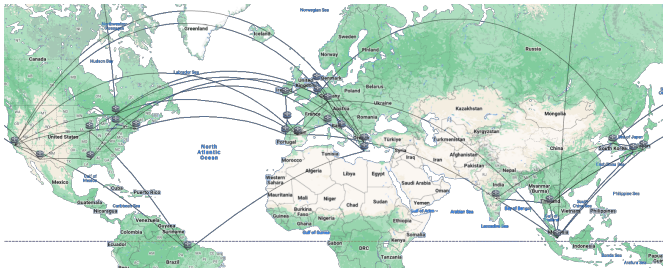

Fig. 6: The topology map of global NDN Testbed. Repo is deployed on every Testbed router.

Our research group has been using NDN Workspace to track progress and report issues anyone encountered. Every user can edit a shared Progress-Tracking file, F_{track}, at any time, independent from whether one is online or offline. Since people in general have intermittent (NDN)-connectivity with each other, before the Repo integration was deployed, it often happened that a few users made changes to the shared F_{track} and then went offline together before the changes were fetched by others. Thus when later user U_B comes online, they could not get those change right away. The application will eventually work out when any of the users comes online later, but the delay in updating F_{track} can be long.

The Repo solved this problem. Repo fetches all the latest changes to F_{track} as soon as they are produced. In the above scenario, when B comes online, its sends Sync Interest I_B to the group; upon receiving I_B which indicating missing data, Repo follows the standard SVS operation by resending the latest Sync Interest, which informs B of its missing data. As results, Repo enables NDN Workspace to support collaboration in a fully asynchronous manner.

VI. Discussion

In today's practice, application developers typically set up storage servers and explicitly synchronize end-users with these servers. Storage replication is also handled explicitly.

A *seemingly* better approach is to develop applications that run natively in a cloud service, which provides both data synchronization and data storage. This approach frees applications from worrying about data storage separately, with the undesirable consequence of yielding all the data control and exposing application semantics and security policies to the cloud. Rapid technology advances have brought us inexpensive, high-volume data storage, yet users today do not seem to have an easy way to take great advantage of it to remove reliance on the cloud for data availability. We believe the root cause of this dilemma is lack of networking and storage integration: network only delivers packets to destination by addresses, and storage is handled at application layer.

This paper points to a new direction in data storage solution development: developing apps based on a data-centric networking model, where network and storage are unified

in performing the function of supplying requested data. As technology advances drop storage cost, this model allows us to take the most advantage of an opportunity to leverage such in-network storage. Particularly, NDN Repo enables asynchronous applications by providing an in-network storage service that is completely transparent to app development. Security is ensured through an application-agnostic model that does not rely on knowledge of application keys or security policy. We observe that deploying Repo to support the NDN Workspace app allows for complete continuity and collaboration between users regardless of overlap in who is online.

Both CDNs and Repo are transparent to the end-users. However, CDNs are impossible to be application-agnostic, since today's data security is realized through securing the TLS channel. As the actual communication endpoint that faces end-users, CDNs must possess keys that can represent themselves as if the original applications owners. This limitation inevitably leads to key sharing between application owners and CDN providers. Repo design follows the opposite approach by storing data that are secured directly by applications, and executes security policies for its own sake of preventing data poisoning but meanwhile staying agnostic to applications.

VII. Looking into the Future

Distributed applications, including metaverse applications, desire persistent data availability to support multiparty collaboration and asynchronous communications. Today, such applications rely on cloud storage for always-on data availability with its associated costs: reliance on infrastructure connectivity, increased communication delay, and, more importantly, the loss of control over application data.

Our design of Repo makes a unique contribution to providing app-agnostic and oblivious in-network data store, which provides always-on data availability and can be seamlessly integrated with network connectivity services. The designs of NDN Workspace and Repo share a fundamental goal, which is to put the power of data control back in users' hands. This goal is achieved by leveraging the results from decade-long NDN research efforts, which developed foundational support for user-named and secured data.

We note that Repo is not meant as a substitute for cloud storage services. We believe that cloud storage service will continue to play an important role in supporting future applications, for example, to be used for high-volume data sharing and/or long term data backup, and for use cases that are yet to come. However, we expect that future data-centric applications could make a fundamental difference between today's cloud storage service and its future usage: the cloud will store user-named and secured data, forcing a change of hands in data control from today's cloud operators to future users.

Integrating storage with network connectivity offers new opportunities for metaverse apps which are typically multiparty in nature and require timely and secure data exchanges. We also envision network-storage integration to open new opportunities for network service providers, which can now provide joint data delivery (networking) and data storage services, moving away from the existing "dumbpipe" service model. User-controlled metaverse apps do not imply that users necessarily provide all needed resources; rather, they could benefit greatly from network-provided resources, as shown by the example of Repo.

References

[1] T. Yu, X. Ma, V. Patil, Y. Kocaogullar, and L. Zhang, "Exploring the Design of Collaborative Applications via the Lens of NDN Workspace," in *Proceedings of the 2nd Annual IEEE International Conference on Metaverse Computing, Networking, and Applications*, 2024.

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

[3] N. Team, https://github.com/UCLA-IRL/ndn-python-repo, 2024, accessed: 2024-2-3.

[4] A. Afanasyev, J. Burke, T. Refaei, L. Wang, B. Zhang, and L. Zhang, "A brief introduction to named data networking," in *IEEE Military Communications Conference (MILCOM)*. IEEE, 2018.

[5] T. Yu, P. Moll, Z. Zhang, A. Afanasyev, and L. Zhang, "Enabling plug-n-play in named data networking," in *2021 IEEE Military Communications Conference (MILCOM)*, 2021. [Online]. Available: https://doi.org/10.1109/MILCOM52596.2021.9653033

[6] T. Yu, X. Ma, H. Xie, D. Kutscher, and L. Zhang, "Cornerstone: Automating remote ndn entity bootstrapping," in *Proceedings of the 18th Asian Internet Engineering Conference*, 2023. [Online]. Available: https://doi.org/10.1145/3630590.3630598

[7] 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. [Online]. Available: https://doi.org/10.1145/2810156.2810170

[8] P. Moll, V. Patil, L. Wang, and L. Zhang, "Sok: The evolution of distributed dataset synchronization solutions in ndn," in *Proceedings of the 9th ACM Conference on Information-Centric Networking*, 2022. [Online]. Available: https://doi.org/10.1145/3517212.3558092

[9] The NDN Team, "NDN Testbed," Online at https://named-data.net/ndn-testbed/, 2024.

[10] T. Yu, Z. Kong, X. Ma, L. Wang, and L. Zhang, "Pythonrepo: Persistent in-network storage for named data networking."

[11] A. Afanasyev, C. Yi, L. Wang, B. Zhang, and L. Zhang, "SNAMP: Secure namespace mapping to scale NDN forwarding," in *Proc. of IEEE Global Internet Symposium*, 2015.

[12] J. Presley, X. Wang, X. Ai, T. Yu, T. Brandel, P. Podder, V. Patil, A. Afanasyev, F. Feltus, L. Zhang *et al.*, "Hydra: A scalable decentralized p2p storage federation for large scientific datasets," in *2024 International Conference on Computing, Networking and Communications (ICNC)*, 2024, pp. 1–7.

[13] K. Nichols, "Trust schemas and icn: key to secure home iot," in *Proceedings of the 8th ACM Conference on Information-Centric Networking*, 2021. [Online]. Available: https://doi.org/10.1145/3460417.3482972

[14] T. Yu, X. Ma, H. Xie, Y. Kocaoğullar, and L. Zhang, "A new api in support of ndn trust schema," in *Proceedings of the 10th ACM Conference on Information-Centric Networking*, 2023, pp. 46–54.

[15] X. Ma, Z. Kong, and E. Newberry, "python-ndn." [Online]. Available: https://github.com/zjkmxy/python-ndn