Distributed Dataset Synchronization in Disruptive Networks

Tianxiang Li
tianxiang@cs.ucla.edu
UCLA

Zhaoning Kong
jonnykong@cs.ucla.edu
UCLA

Spyridon Mastorakis
smastorakis@unomaha.edu
University of Nebraska, Omaha

Lixia Zhang
lixia@cs.ucla.edu
UCLA

***Abstract*—Disruptive network scenarios with ad hoc, intermittent connectivity and mobility create unique challenges to supporting distributed applications. In this paper, we propose Distributed Dataset Synchronization over disruptive Networks (DDSN), a protocol which provides resilient multi-party communication in adverse communication environments. DDSN is designed to work on top of the Named-Data Networking protocol and utilizes semantically named, and secured, packets to achieve distributed dataset synchronization through an asynchronous communication model. A unique design feature of DDSN is letting individual entities exchange their dataset states directly, instead of using some compressed form of the states. We have implemented a DDSN prototype and evaluated its performance through simulation experimentation under various packet loss rates. Our results show that, compared to an epidemic routing based data dissemination solution, DDSN achieves 33-56% lower data retrieval delays and 40-44% lower overheads, with up to 20% packet losses. When compared to the existing NDN dataset synchronization protocols, DDSN can lower the state and data synchronization delays from one-third to two-third, and lower the protocol overhead by up to one-third, with the performance difference becoming more pronounced as network loss rates go up.**

***Index Terms*—Disruptive Networks, Named-Data Networking, Distributed Dataset Synchronization**

I. Introduction

Dataset synchronization is a common need for distributed applications that share information among multiple participants, such as group text messaging or collaborative editing. With today's TCP/IP networking, such distributed applications use centralized cloud servers and achieve dataset synchronization at the application layer. This approach matches well to the traditional client-server application paradigm, but has two major drawbacks. First, it requires the provisioning of a centralized server to enable a distributed application, with consequential requirement on mitigating single point of failures. Second, it depends on network infrastructure support to connect all the involved parties to the centralized server, even when those communicating parties are located next to each other but far away from the centralized servers.

These drawbacks may be minor nuisance in a well provisioned network, however they become roadblocks in supporting distributed applications in adverse environments where the infrastructure support is poor (e.g., in developing countries), or simply non-exists (e.g., during a disaster recovery). In such situations, it is conceivable to achieve dataset synchronization by peer-to-peer communication via local connectivity, however one must address two major challenges first. The first one is what namespace to use to communicate, as IP addresses identify topological locations and become meaningless in infrastructure-free scenarios. The second one is how to ensure communication security: the certificate authority servers may be unavailable, and the TLS or DTLS setup processes may become infeasible in a fast changing, disruptive network.

In this paper, we leverage a newly designed Internet architecture, Named Data Networking (NDN) [1], to propose a data-centric solution for distributed dataset synchronization[1], dubbed *Distributed Dataset Synchronization over disruptive Networks* (DDSN). NDN directly uses application level names to fetch data at network layer, and secures each data packet directly by a cryptographic signature which binds its name and content together. Thus NDN provides answers to the two challenges mentioned above, and DDSN is designed to bridge the gap between an NDN network's datagram delivery of named, secured data packets and the applications' need for dataset synchronization among multiple parties.

The contributions of this work are two-folds. First, we designed a new dataset Synchronization protocol, DDSN, specifically for disruptive network environments. Taking a departure from the previous NDN sync protocol designs, DDSN lets individual entities communicate using dataset state directly instead of its compressed form. Doing so enables synchronization packets to be processed without any assumption on the receiver's state, bringing resiliency to communication when the state at individual nodes diverges under adverse conditions. Second, we implemented a prototype of DDSN and conducted simulation evaluations under various network loss rates. Our results show that, compared to an epidemic routing [2] solution for data dissemination, DDSN achieves 33-56% lower data retrieval delays and 40-44% lower overheads under 0%-20% packet loss rates. When compared to previous NDN sync protocols, DDSN achieves 27-77% lower synchronization delay, 26-76% lower data retrieval delay, and 3-36% lower overhead, with the lower bound representing the difference under zero loss and the upper bound the difference under 20% packet loss rate.

The rest of the paper is organized as follows. Section II provides a brief background on NDN and previous work on distributed dataset synchronization in NDN, and Section III summarized related work. Section IV presents an overview of

[1]In the rest of this paper we use the term "sync" to refer to both distributed dataset synchronization protocol and the overall synchronization process.

the DDSN design, and Section V describes the DDSN building blocks in detail. Section VI reports our evaluation results. Section VII discusses extensions of the DDSN design, and Section VIII concludes the paper and discusses future work.

II. Background

In this section, we first provide a brief overview of the NDN architecture, and then defines the problem of distributed dataset synchronization in the NDN context.

A. Named Data Networking

The basic idea of Named Data Networking (NDN) [1] is to communicate via named packets. In NDN, each piece of data is assigned a semantically meaningful name. This name identifies the data itself and is independent of the data container, its location, or the underlying network connectivity. To communicate, *data consumers* express requests, called *Interests*, which carry the names of desired data generated by *data producers*. Each producer secures all its data at the data production time, by using a cryptographic signature to bind the data name its content.

Each NDN node N utilizes four modules in communication. (i) The Forwarding Information Base (FIB) contains a list of name prefixes together with one or multiple outgoing interfaces for each prefix; (ii) The Pending Interest Table (PIT) keeps track of all Interests that N has forwarded but has not received the corresponding data packets; (iii) The Content Store (CS) opportunistically caches passing-by data packets; and (iv) The forwarding Strategy makes interest forwarding decisions based on multiple inputs including the FIB.

Equipped with the above four modules, each NDN node N forwards Interests based on their names. After N forwards an Interest, it adds an entry to its PIT. Once an Interest reaches a node that has the requested *data packet* P_{data}, P_{data} goes back to the original requester by following the reverse path of the corresponding Interest based on the state kept at each forwarder's PIT. Along the way each forwarder may cache P_{data} in its Content Store, to be used to satisfy future requests for the same data.

B. Distributed Dataset Synchronization in NDN

Sync can be viewed as the transport layer abstraction in a data-centric network architecture. Sync bridges the gap between the functionality required by distributed applications, namely synchronizing the dataset of distributed entities, and the datagram delivery provided by the underlying NDN network. Since each NDN data name uniquely identifies a piece of immutable data, Sync achieves the shared dataset synchronization among distributed parties by synchronizing the data namespace of all the involved parties [3]. For example, in a group messaging application (textchat), each user keeps a local view of the *shared dataset* (i.e. all the messages generated by the other users in the same chat group). We refer to this local view as the user's *shared dataset state*, and all the users in the chat group as members of a *sync group*[2].

[2]We use the terms nodes and members in an interchangeable way.

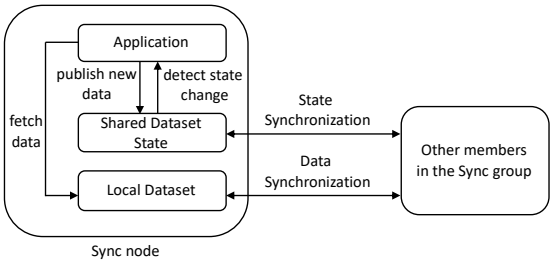


Fig. 1: Main sync components

Each member running the group text messaging application is identified by a name. We call this name "member prefix". Members use a multicast "group prefix" to communicate with all other members in a sync group.

Figure 1 shows the main components of sync. Whenever a piece of new data is published, Sync is responsible for updating the shared dataset state. Upon the reception of newly generated data from other members, Sync updates the dataset state and notifies applications about the new data. The goal of a sync protocol is to maintain a consistent dataset state among members in a sync group (i.e., achieve state synchronization), so that each member has up-to-date knowledge about all the data that has been generated. Then each member fetches the data (i.e., achieve data synchronization) based on application requirements and/or network constraints [4]. In the rest of this subsection, we discuss three fundamental design aspects of a sync protocol.

Dataset state representation: The dataset for each application instance is generally named under the prefix "`/application/instance`", which is abbreviated to "`/[group-prefix]`" in this paper. Within an application instance, one approach to efficiently naming all the data generated by each member is to name each data packet (e.g., a chat message) sequentially, i.e. by appending a sequence number to its producer's name, with the resulting data name "`/[group-prefix]/[member-prefix]/[data-seq-number]`"). Given that sequence numbers monotonically increase, the shared dataset state among members can be represented by a collection of such names, with one name for each member, and the sequence number indicating the latest generated data packet by that member. Sync protocols have adopted different mechanisms to encode the dataset state for transmission over the network. We refer to that as *state encoding*, and we present specific examples in Sections III and V-A.

Dataset state change detection: To detect changes in the shared dataset, members share their locally maintained state information with other members via a sync protocol. When a member produces a piece of new data, it increases its data sequence number by one and informs others. A member detects changes in the dataset namespace by comparing the locally maintained state information with received state information from others. If a change is detected (i.e., someone in the sync group have generated new data), sync notifies the local application.

State and data synchronization: By running a sync protocol, members learn the latest data generated by others in the sync

group (state synchronization). After state synchronization, whether to fetch the newly learned data (data synchronization) is determined by the application. In other words, *the state and data synchronization processes are decoupled*: the sync protocol running at each member learns the latest data generated by other members; the application decides whether, or when, to retrieved the data.

III. Related Work

A number of sync protocols, running over NDN, have been developed over the last few years [3], [5], [6], [7], [8], [9], [10]. Two of them have been widely used: ChronoSync [6] and PSync [7]. Despite their differences, a common design assumption is the operation in an infrastructure-based environment, where stable connectivity is the norm and nodes are synchronized most of the time.

ChronoSync adopts a cryptographic digest data structure for the dataset state encoding. However, this representation is not semantically meaningful, thus it cannot be used to interpret the difference in the dataset state. As nodes form disconnected network clusters, they accumulate different state locally over time. This leads the state information of members to severely diverge; each member may encounter different members over time, thus be aware of different data produced by different members. To recover from such cases of *state divergence*, ChronoSync needs to fall back to a recovery process in order to resolve differences in dataset state. This process requires at least three, in practice several, rounds of message exchanges.

iSync [5] and Psync uses an invertible Bloom Filter (IBF) [11] to encode dataset state, relying on the IBF subtraction operation to infer the state difference. However, the IBF subtraction allows state comparison between only two members at a time. This becomes inefficient as the number of members with diverged state in the network increases. IBF by design also has certain limitations on the amount of information it can decode under a single operation due to the probability of false positive errors [11]. As a result, in disruptive environments with severe state divergence, it may take several rounds of exchanges to decode state differences among encountered members.

VectorSync [8] introduced the idea of using a version vector [12] for state encoding. Instead of using digest based state encoding, VectorSync enumerates the most recent data sequence numbers of all group members. To allow each member correctly interpreting the vector (which member owns which sequence number), VectorSync adopted a leader-based mechanism to maintain a consistent view of the current sync group members (membership list). DSSN [10] was designed to synchronize dataset of sensor groups that are wirelessly connected. Due to energy conservation, not all sensors are online all the time, making membership agreement infeasible. Thus DSSN extended the VectorSync design by explicitly listing the member name prefix in the version vector. However, DSSN is designed for stationary sensor groups within a single-hop communication range, and does not work in ad hoc mobile scenarios with intermittent multi-hop connectivity.

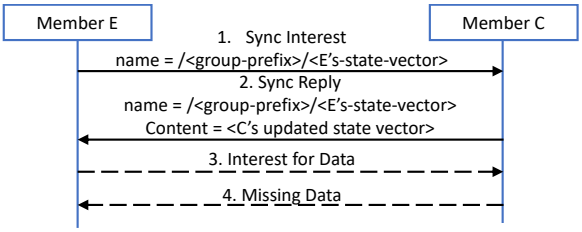


Fig. 2: High-level example of the DDSN operation

DDSN inherited a few basic ideas from the previous work, including: (i) naming data sequentially [6], (ii) using a vector based approach for state encoding [8], [10], and (iii) using periodic transmissions for state change detection [6], [7]. However, different from all the previous NDN sync protocols, DDSN is specifically designed to operate under ad hoc mobile environments with intermittent connectivity and potentially severe packet losses.

IV. Design Overview

The goal of DDSN is to meet the needs of distributed applications for data sharing in disruptive network scenarios, such as disaster recovery (Section IV-A). The fundamental design challenges that DDSN aims to address include:

- How to efficiently recover from state divergence, a situation that is the norm in intermittently connected networks.
- How to asynchronously exchange state information under intermittent ad hoc connectivity in a timely manner, while being resilient to lossy network channels.
- How to prioritize transmissions of up-to-date state information and minimize redundant or outdated state information transmissions under limited network capacity and short-lived connectivity.

To tackle the first challenge, DDSN uses a vector based structure for state encoding (Section V-A). We refer to this structure as *state vector*, which enables lightweight state divergence recovery between multiple members.

To address the second challenge, DDSN utilizes two different messages, a sync Interest and a sync Reply (messages 1 and 2 in Figure 2), to synchronize the dataset state among members. By attaching a state vector to the name of sync Interests, receivers can directly interpret the state information within a single round of message exchanges (Section V-B). This minimizes the transmission overhead and DDSN resilient to different degrees of network losses. Once members synchronize their state, they may send Interests to retrieve missing data from each other based on application needs (messages 3 and 4 in Figure 2).

To tackle the third challenge, DDSN's semantically meaningful naming abstractions can be interpreted by any receiving member. Members overhearing exchanged Sync Interests and Sync Replies can interpret the state information directly, and utilize this information to prioritize their transmissions. As a result, members with up-to-date state information receive priority, while members with outdated state information suppress their otherwise redundant transmissions (Section V-B).

This ensures that up-to-date state information propagates in the network without any delay.

DDSN also takes into account the case of node isolation. DDSN switches between *regular* and *inactive* modes based on whether a member overhears transmissions from others. In inactive mode, members suppress transmissions and switch to a probing phase for neighbor discovery (Section V-C).

At the same time, DDSN members perform hop-by-hop propagation of sync Interests over multiple wireless hops, and probabilistic forwarding of Interests for missing data (Section V-D). This enables members to synchronize their state with other members multiple hops away from each other, as well as retrieve data available more than one hop away.

To support partial data sync, we extend DDSN's baseline state vector design to contain application-defined preference information. This allows members to build local knowledge about the data that their neighbors are interested in. As a result, members prioritize the retrieval of data that themselves and most of their neighbors are missing (Section V-E).

Throughout the sync process, DDSN leverages the built-in NDN security mechanisms [13] to secure data directly at the network layer. All the data packets carry the cryptographic signature of the data producer, and each data can be authenticated regardless of where it is stored. Members also authenticate the sync Interests containing the state vectors of others based on pre-established common trust anchors [13]. (Section V-F).

A. Example Scenario

We use an example scenario throughout this paper to help illustrate our design. In disaster recovery scenarios, such as earthquakes, the communication infrastructure may be severely damaged. As a result, communication needs to be performed via local network connectivity. In the aftermath of an earthquake (Figure 3), first responders (e.g., nodes A, B, C, D and E) move around on a field to perform rescue operations. We assume that the first responders run an information sharing application and an instance of the DDSN protocol on their devices, and they are members of the same sync group. In this way, they can collect/generate information about survivors or the status of the disaster, which they would like to synchronize and share with all the other responders in the group.

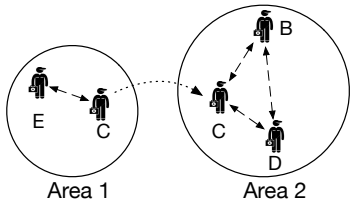


Fig. 3: An example of a disaster recovery scenario

For example, let us assume that E has information about a survivor that was found in area 1. As E moves around, it encounters C and exchanges this new survivor information with C. When C moves into area 2, it further disseminates this information to B and D. In this way, new information is shared asynchronously among members of the rescue team under intermittent connectivity.

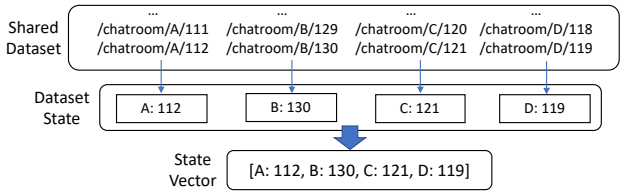


Fig. 4: State vector example

V. Design Building Blocks

A. Dataset State Representation

As mentioned in Section II-B, a sync protocol synchronizes the namespace of a shared dataset. State divergence among members raises challenges when it comes to interpreting the differences of a large state mismatch. This is common in networks with intermittent connectivity, since members may not be connected to each other when they generate new data. To address this challenge, DDSN adopts a vector based state encoding, called *state vector* [10]. Each member maintains locally a state vector that represents the latest data generated by every other member in the sync group that it is aware of. The vector contains a set of key-value pairs, the ith pair $\{(p_i : seq_i)\}$, indicates that the latest data packet produced by the member with prefix p_i has a sequence number of seq_i. A state vector example for a "`/chatroom`" sync group is shown in Figure 4, where the sequence number of the latest data generated by members A, B, C, and D is 112, 130, 121, and 119 respectively.

When a member receives a state vector from another member, it first compares the received vector to its local vector by computing the difference between the sequence numbers under the same member prefixes. For example, in Figure 3 (after C moves to area 2), let us assume that C's state vector is $[B : 1, C : 2, D : 2, E : 3]$ and that C receives D's state vector, which is $[B : 2, D : 4, E : 1]$. By comparing its local vector to D's vector, C identifies that B has generated a new data packet with sequence number 2 and D has generated two new packets with sequence numbers 3 and 4 respectively. A member merges the received vector with its local vector by choosing the pairwise maximum between the sequence numbers under the same member prefix in the vectors. It also adds new member prefixes found in the received vector to its local vector. In the previous example, assuming that D also receives C's vector, D's merged vector will be $[B : 2, C : 2, D : 4, E : 3]$. A state vector is raw state information expressed in a condensed way, which is meaningful on its own. To this end, we do not make any assumptions on the receiver state or the network topology, allowing members to recover from any degree of state divergence.

B. State Synchronization

The dataset state synchronization process consists of the exchange of a sync Interest and a sync Reply. Figure 2 shows an example of a basic sync Interest-Reply exchange between members E and C of Figure 3 (while C is still in area 1). A sync Interest name contains a *group prefix*, which allows

members in a sync group to multicast [3] their sync Interest(s) to others in the same group. The second part of the sync Interest name is the member's state vector, which is used to express the dataset state of the sender. Members receiving a sync Interest first merge the state vector in the Interest's name with their local state vector, and then send back a sync Reply containing their updated (merged) state vector.

As members move around, they encounter other members within their communication range. Encountered members synchronize their dataset state until they converge to a consistent state. For example, in Figure 3, node C encounters B and D. To sync their dataset state, one of the nodes, for example C, initiates the sync process by sending a sync Interest with its local state vector to let B and D know about its latest state.

1) Detecting State Mismatch: To resiliently detect state mismatch in a dynamic network environment, DDSN adopts both an event-driven and a periodic approach for the transmission of sync Interests. Based on the former mechanism, when a node generates a new data packet (or after it has accumulated a number of new data packets, depending on application requirements), it sends a sync Interest to members within its communication range. This Interest serves as a proactive notification of state change. Based on the latter mechanism, nodes periodically send a sync Interest to detect state mismatch with others within their communication range. This is needed, since nodes move in and out of each other's communication range, thus, we cannot assume stable state among them.

Specifically, nodes send a sync Interest periodically based on a countdown timer (t_{sync_int}) from a maximum value t_{max} to 0. The generation of new data by a node triggers the transmission of a sync Interest without delay and refreshes t_{sync_int} to postpone the periodic transmission of state information.

2) Sync State Propagation: To achieve timely synchronization of state among members, DDSN has two fundamental design goals for state propagation: (i) propagate new state information in the network with minimum delay; (ii) avoid propagating redundant or outdated state information.

DDSN's semantically meaningful message names allow nodes to interpret the received state, and prioritize their subsequent transmissions to achieve these design goals. The processing logic of a received sync Interest is illustrated in Figure 5. The receiver first assesses whether the state vector in the Interest contains the same, more recent, or older state information than its local vector. This is achieved by directly comparing the sequence number under each member prefix in the received vector to the sequence numbers in its local vector.

If the received vector contains more recent state (i.e., the sequence number under any member prefix is higher than the receiver's local vector or if the received vector contains new member prefixes), the receiver merges the received vector with its local vector (Section V-A) and sends a new sync Interest containing its updated state vector without delay. This ensures that new state is propagated as fast as possible in the network. If the received vector contains older state than the local vector, the receiver suppresses itself and does not send a new sync Interest. In this way, nodes avoid propagating outdated state information. If the received state is the same as the local state, the receiver again suppresses itself, since this is treated as an indication that some other node in the vicinity has previously transmitted the same state information.

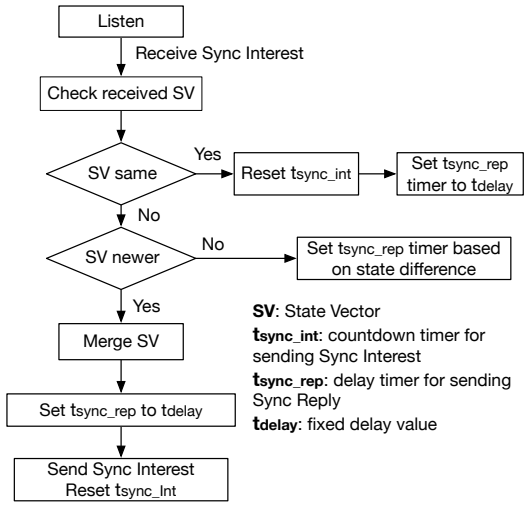


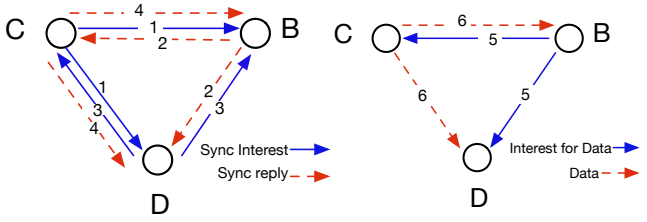
Fig. 5: Processing logic of a received sync Interest

3) Acknowledging State Transmission: A sync Reply acts as an acknowledgment for the reception of a sync Interest and ensures that a bidirectional exchange of state information between members take place. By comparing the received state vector and the receiver's local state vector, the receiver can deduce the missing data of the sync Interest sender, and optionally contain it in the sync Reply. Multiple members may be within the communication range of each other, thus receive the same sync Interest. DDSN adopts a prioritization mechanism to avoid redundant replies, making sure that only the member that has the most accumulative up-to-date state compared to the state vector in the received Interest will reply. That is, if a receiver of a sync Interest has more recent state than the state in the Interest, it schedules the transmission of a sync Reply based on how much accumulative new state it has under each member prefix compared to the received state vector. This is achieved by selecting an appropriate value for a sync Reply transmission delay timer t_{sync_rep}. Specifically, if the sync Interest sender's state vector contains the value pair set $\{(p_i, seq_i)\}$, where $0 < i < m$, and the receiver's state vector contains the value pair set $\{(p_j, seq_j)\}$, where $0 < j < n$, then:

$$t_{sync\_rep} = \frac{t_{delay}}{\sum_{\{i,j|p_i=p_j \wedge seq_i > seq_j\}} (seq_i - seq_j) + 1}$$

Note that t_{delay} is a fixed delay value, and m and n are the number of member prefixes in the vector of the sync Interest's sender and receiver respectively. On the other hand,

[3]A member of a sync group broadcasts a frame containing a sync Interest at the MAC layer. This is a multicast transmission at the network layer, since this Interest will be handled by the NDN forwarder of a receiving node based on the Interest's group prefix. The NDN forwarder passes the sync Interest to upper layers, including the sync transport layer, only if the Interest's group prefix matches the sync group prefix of the node.



(a) State sync process (b) Fetching missing data

Fig. 6: DDSN sync process example

if a receiver of a sync Interest has older state compared to the state vector in the Interest, it only updates its local state by merging the vector in the Interest with its local vector.

Example: In Figure 6a, C's sync Interest is received by B and D. Assuming that B has more accumulative new state than D compared to C's state vector, B will receive priority to send a sync Reply. This is achieved by having B schedule its t_{sync_rep} timer to expire before D's t_{sync_rep} timer. B will send a sync Reply first, which includes its own state vector. D will overhear this reply and cancel the transmission of its own sync Reply. Then, D sends a sync Interest to let B and C know about its own state vector. B and C receive D's Interest, and C replies first and suppresses B from replying[4]. After B, C, and D have received each other's vectors, they all converge to a consistent dataset state. As mentioned in Section II-B, we refer to this process of synchronizing the dataset state among members as *state sync*, which allows each member to have a consistent view of the shared dataset.

Once nodes exchange state vectors and converge to a consistent state, each node can identify the data it is missing based on the difference between its local dataset (i.e., the data it has already fetched) and its local state vector. Nodes start fetching missing data from others by creating a queue of missing data names and sending Interests to fetch the corresponding packets. For example, in Figure 6b, we illustrate the process of node B sending an Interest for missing data, which is received by C and D. C has the requested data and responds to this Interest. As mentioned in Section II-B, we refer to this process of fetching missing data based on established state and application requirements as *data sync*, which is decoupled from the state sync process. State sync has a higher priority to data sync in terms of transmission scheduling, as keeping an updated state information facilitates the fetching of missing data.

C. Inactive Mode

In disruptive networks with mobility and intermittent connectivity, it is a common case that nodes may endure certain isolation periods. During such periods, they will not be connected to any other nodes, thus requests for data becomes meaningless. It is also important during this period to detect encountered members to learn whether they have any newer state [14]. DDSN offers a mechanism to detect when isolation happens and when nodes get connected again.

When a node does not receive any packet transmissions for a certain period of time ($t_{activity}$), it enters the *inactive mode*. We refer to $t_{activity}$ as the *activity timer*. In inactive mode, a node suppresses transmissions, since no receivers are around, and switches to probing through periodic sync Interests that contain its latest state vector[5]. If a node receives/overhears a packet transmission while in inactive mode, it interprets that as being connected again with others. Therefore, it exits inactive mode, resets its activity timer, and continues its regular operation.

D. Communication over Multiple Wireless Hops

DDSN makes use of the event-triggered transmission of sync Interests to propagate on a hop-by-hop manner new state over multiple wireless hops. Each node that receives a state vector with up-to-date state sends a sync Interest containing its updated state vector. For the data synchronization process, requests for data are transmitted over multiple hops based on a certain forwarding probability at each hop. In the future, we plan to investigate mechanisms to build soft-state knowledge about the available data over multiple hops around nodes. This will allow us to dynamically adjust the probability of forwarding data requests based on how likely it is that the requested data is available around nodes.

E. Partial Data Synchronization

For distributed applications it is common that users may be interested only in data generated under specific member prefixes (e.g., users may subscribe to specific news resources). In such pub-sub model, users can be notified when a member they are interested in has generated new data, and fetch the data accordingly. We call this process *partial data synchronization*. DDSN offers a set of general-purpose transport layer abstractions to cover the needs of *any* distributed multi-party application under intermittent connectivity by supporting both full data synchronization and partial data synchronization. In adverse environments, each member carries and propagates the state of the data generated by all other members in a sync group. In this way, DDSN allows as many members as possible to learn in a timely manner the latest state of the data they would like to fetch. Each member can then fetch the data based on its subscription.

The key challenge in this scenario is that members may hold only a subset of the full dataset (data they have subscribed to). However, members propagate information about the state of the entire dataset (state of all other members). This mismatch between a member's local dataset and the propagated state information triggers members to send requests for data that may not be available around them.

To address this issue, we can add an additional field in the state vector called *preference flag*, under each producer prefix.

[4]In this example, both B and C receive equal priority to respond to D's sync Interest, since they both have the same state. To avoid collisions in such cases, B and C use a timer to randomize their sync Reply transmissions.

[5]Alternatively, probing can be done through lightweight probing Interests under a predefined probing namespace. When a node receives a probing Interest, it sends a sync Interest to initiate the state sync process.

When a node is subscribed to a particular producer prefix, it will set the preference flag to 1 for that prefix or 0 otherwise. For example, the ith pair of values in the state vector would be in the form of $\{(p_i, seq_i, flag_i)\}$, which represents: the latest data sequence number produced under producer p_i's prefix is seq_i, and $flag_i$ indicates whether this consumer is subscribed to producer p_i's data. The preference flag reflects what data is held by each member, and is used to prioritize the requests for data. Each node can thus build up a local table regarding which prefixes its neighbors are subscribed to, by aggregating the preference flags it received from its neighbors' Sync Interests. To ensure the timeliness of the information, each flag in the table would have a life timer which expires after certain period (then the flag value is set to 0). A node will refresh the life timer of a preference flag if it receives a state vector with same preference flag set to 1. Nodes will prioritize the transmission of Interests for data under the producer prefix subscribed to both by itself and by at least one of its neighbors. This allows higher chance of fetching available data, and the data fetched is more likely to benefit others neighbors as well.

F. Sync Interest Authentication

Received sync Interests change the receiver's state. This can cause the receiver to fetch data perceived as missing based on the received state. Sync Interests could be abused by malicious nodes for bogus state injection in a sync group, which would cause nodes to fetch bogus or non-existent data. To deal with bogus state vectors, nodes sign sync Interests and attach the signature to the Interests. To sign Interests, each node either has a pre-configured shared symmetric key per sync group or a pair of dynamically generated public and private keys. Nodes authenticate received sync Interests to verify the validity of the carried state. They also decide whether they trust the node that signed each Interest based on pre-established trust anchors.

VI. Experimental Evaluation

In this section, we present our experimental evaluation. We first describe the experimental setup (Section VI-A), and then present our experimental results under adverse network conditions (Section VI-B). We first compare the DDSN performance with a data dissemination solution based on epidemic routing [2] (Section VI-B1), then compare the DDSN performance with two existing NDN-based sync protocols (Section VI-B2).

A. Experimental Setup

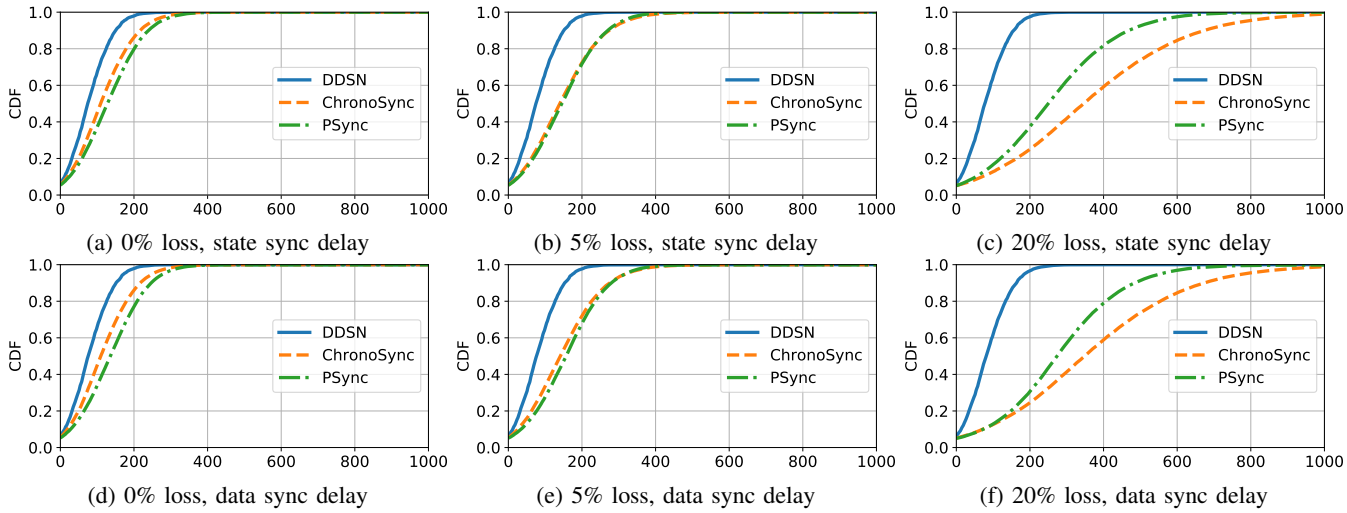
We have implemented a DDSN prototype in C++ and makes use of the ndn-cxx library [15] to ensure compatibility with NFD [16]. To evaluate DDSN under a variety of settings and conditions, we ported our prototype into the ndnSIM network simulator [17] (our simulation code can be found at https://github.com/JonnyKong/DDSN-Simulation). ndnSIM offers integration with the NDN prototype implementations (ndn-cxx library and NDN forwarder). We use a grid topology of 800m$\times$800m with 30 mobile nodes (20 sync nodes that belong in the same sync group and 10 other nodes which only forward packets) that move based on the Random Walk Mobility Model. Each node randomly chooses its direction and speed. The speed ranges from 1m/s to 20m/s, representing both humans and vehicles, and the direction ranges from 0 to 2π. Each node moves along the same path for 20s before changing its direction and speed. Nodes communicate through NDN over IEEE 802.11b 2.4GHz (transmission rate of 11Mbps) with a WiFi range of 60m (unless otherwise noted). Each node generates data following a poisson distribution with $\lambda = 40$s on average and its data generation process lasts for 800s. The payload of each data packet consists of random text of size 100-1024 bytes. Note that we perform 10 trials for all the experiments mentioned below and we present the 90th percentile of the collected results (with the exception of CDF plots that show the full distribution of the results).

We consider the following evaluation metrics: (i) *state sync delay*: the time needed for an updated state vector to reach all the members in the group, (ii) *data sync/retrieval delay*: the time needed for newly generated data to reach all members in the group, and (iii) *overhead*: the traffic volume (in terms of bytes or Interests) generated for all the members to retrieve all the generated data.

Comparison to an epidemic dissemination solution: We implemented a data dissemination solution based on epidemic routing in an IP-based network setup, which we compare to DDSN. Epidemic routing relies on periodic transmissions of beacon messages to detect when nodes are connected to others. The reception of a beacon triggers the encountered nodes to exchange their *summary vectors*. A summary vector contains a list of data packets that its sender has. After nodes exchange their summary vectors, they send to each other data that they have, but others are missing. Each node also has a buffer to store and carry data it overhears to other nodes. For epidemic routing, we use UDP multicast running on top of a broadcast IP address and we assume that every node has adequate resources to store and carry all the generated data packets. For a fair comparison with DDSN, we set the period of beacons to 8s, i.e., the same as the period of the Sync Interest transmissions.

Comparison to existing sync protocols in NDN: We ported the prototype ChronoSync and PSync implementations into ndnSIM, which we compare to DDSN. To perform a fair comparison, we let each protocol transmit a sync Interest every 8s. and each member in the sync group fetch the data generated by all the other members.

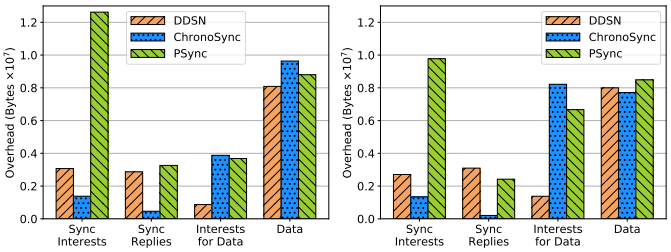
Retransmission Strategy: We adopted a data request retransmission strategy for lossy network environments. It is based on the characteristics of sync protocol operations. For sync, initially a state update is sourced from the data producer, then it is propagated to nearby nodes which in turn fetch the new data. In other words, the state and data propagate together when there is adequate connection time among nodes, thus the sender of new state is likely to carry the corresponding data. To facilitate the exchange of new data under intermittent connectivity, we place newly generated data requests in a transmission queue with a short transmission interval (0.5s in our simulations). If such a request does not bring data



(a) 0% loss, state sync delay (b) 5% loss, state sync delay (c) 20% loss, state sync delay

(d) 0% loss, data sync delay (e) 5% loss, data sync delay (f) 20% loss, data sync delay

Fig. 7: State and data sync delay, comparison between DDSN & existing NDN sync protocols



(a) 0% Loss Rate (b) 20% Loss Rate

Fig. 8: Overhead, DDSN & existing NDN sync protocols

back after a certain number of retransmissions (10 in our simulations), it is placed to a queue with a longer transmission interval (5s), so that nodes avoid retransmitting repeatedly to the same neighbors. Compared to a baseline approach of periodic request retransmissions every 5s, our strategy reduces the number of transmitted data requests by 35% and the data sync delay by 45% under 50% loss rate. This retransmission strategy is implemented for each sync protocol in our simulation setup.

B. Experimental Results

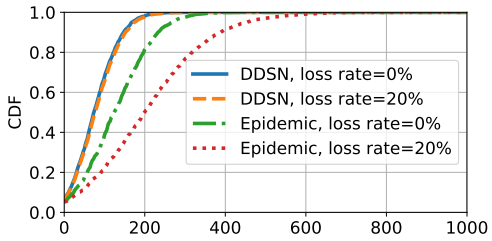


Fig. 9: Data retrieval delay, DDSN & epidemic dissemination

1) Comparison with Epidemic Data Dissemination: In this subsection, we compare the performance of DDSN to a data dissemination design based on epidemic routing, which opportunistically disseminates data to nodes in the network through pair-wise exchange of messages between nodes [2].

Data retrieval delay: In Figure 9, we present the CDF of the data retrieval delay for DDSN and epidemic data dissemination. The results demonstrate that DDSN nodes retrieve 90% of the generated data in less than 140s for loss rates of 0% and 20% respectively. On the other hand, in epidemic dissemination, nodes need around 220s and 400s to retrieve 90% of the generated data for loss rates of 0% and 20% respectively. That is, DDSN achieves 35-56% lower delays than epidemic dissemination. This is due to the fact that DDSN follows an event-based approach that propagates without any delay up-to-date state information multiple hops in the network, so that nodes can retrieve missing data as quickly as possible. Epidemic routing relies on the periodic transmission of beacon messages to trigger message exchange between a pair of nodes; only in cases that another node receives a beacon, the encountered nodes exchange their summary vectors and send data to each other.

Overhead: DDSN generates 1.49 and 1.44 bytes$\times 10^7$ of overhead for loss rates of 0% and 20% respectively, while epidemic dissemination generates 2.48 and 2.58 bytes$\times 10^7$ of overhead for loss rates of 0% and 20% respectively. These results show that DDSN achieves 40-44% lower overheads. The main reason is that DDSN's uniform sequential naming convention of each data (member prefix + sequence number) allows the state of the entire dataset to be encoded in the state vector by enumerating all the member prefixes and the newest sequence number under each prefix. The state vector size remains the same as the number of data generated continues to increase. Epidemic routing's summary vector enumerates all the data generated, and is not scalable as the number of generated data increases. The state vector size is 150-200 bytes during the experiments, however, the summary vector grows up to 1600 bytes. Especially when data generation rate is high, summary vector needs to remove some of the older data information to maintain its size limit. Under disruptive environments, this can result in some of the data to be never received by certain nodes.

2) Comparison with Existing NDN Sync Protocols: In this subsection, we compare DDSN with ChronoSync and PSync

under different loss rates.

State and data sync delay: Figure 7a, 7b, and 7c show the CDF of the state sync delays under packet loss rate 0%, 5%, and 20% respectively. Overall, DDSN achieves lower state sync delays compared to ChronoSync and PSync due to its prioritized propagation of up-to-date state information in the network. The results also demonstrate that DDSN is more resilient than ChronoSync and PSync to different rates of packet loss. For low packet loss rates (0-5%), DDSN achieves 27-38% lower state sync delays (90th percentile) than ChronoSync and PSync. For high loss rates (20%), DDSN achieves 67-77% lower state sync delays (90th percentile) than ChronoSync and PSync. When loss rate is high (50%), it becomes difficult for Chronosync and PSync to propagate the state to more than 50% of the nodes, while DDSN performance remains stable. This is due to a number of design factors. First, DDSN utilizes the successful receipt of a sync Interest carrying the state vector to ensure state propagation. The receiver can directly interpret the state information and resolve state inconsistency based on the received state vector. On the other hand, ChronoSync and PSync require multiple rounds of message exchanges for state divergence recovery. As the loss rate increases, the increase in state sync delay becomes significant for ChronoSync and PSync due to lower success rates of completing the recovery process. Secondly, each DDSN member prioritizes the propagation of new state information to its neighbors, which disseminates state update more effectively under intermittent connectivity.

The data sync delay shown in Figure 7d, 7e, and 7f demonstrate that DDSN is again more resilient than ChronoSync and PSync to packet loss. DDSN is able to distribute data to more nodes than the other two protocols. This is due to the fact that data sync follows tightly after state sync. Receiving new state information enables nodes to send Interests for the new missing data. For low packet loss rates (0-5%), DDSN achieves 26-36% lower data sync delays (90th percentile) than ChronoSync and PSync. For high loss rates (20%), DDSN achieves 68-76% lower data sync delays (80th percentile) than ChronoSync and PSync. Under 50% loss rate, ChronoSync and PSync did not distribute more than 80% of the generated data, due to the large state synchronization delay.

Overhead: In Figure 8a and 8b, we present the overhead results for DDSN, ChronoSync, and PSync for loss rates of 0% and 20% respectively. The results show that DDSN achieves 3-17% and 5-36% lower overheads than ChronoSync and PSync respectively. ChronoSync uses a digest to compactly encode the latest state, thus sync Interests are of small sizes. However, ChronoSync results in a large number of requests for data, since its design does not offer detection of members within the communication of each other. In PSync, the dominating overhead factor is the compressed IBF size, which increases with the size of the encoded state information.

VII. Discussion

A. State and Data Mismatch

As we mentioned in Section II-B, sync decouples the state and data sync process by allowing nodes to first sync the shared dataset state, then fetch data if it is needed by applications. This is an important design choice considering the requirements of different applications, and works well in infrastructure networks, where nodes are typically connected and have all the data that their state indicates. However, in environments with intermittent connectivity, nodes might not have all the data their state indicates. As a result, nodes may propagate state for data they do not have. Nodes receiving this state will try to fetch potentially unavailable data, resulting in a large number of transmitted requests. One feasible solution would be to extend the state vector design, by adding an additional data sequence number under each member prefix to reflect the actual dataset of the state vector sender. Thus, the state vector reflects the newest sequence generated under each member prefix and the actual dataset of the sender. As a tradeoff, this approach would introduce additional overhead for the state vector.

B. Scalability and Maintenance of State Vector

Depending on the number of sync members (dataset size) encoded in a state vector, the vector size might grow. For example, if we assume that the size of each member prefix is 8 bytes, each sequence number is 8 bytes, and the IEEE 802.11 WiFi MTU is 2304 bytes, a state vector will be able to encode up to about 120-130 members. Although the state vector can be encoded in binary, its size can still exceed MTU when there are a few hundreds of nodes. For the state synchronization process, members can transmit a partial state vector, containing only the dataset state information of the members which have recently generated new data. Approaches to enhance the state vector scalability can also be explored. For example, compression can be employed to reduce the vector size. Another candidate approach might be a multi-dimensional vector structure [18], which offers multiple dimensions of state aggregation. To address cases of ever growing state vectors, members need to have a mechanism to clean up their local vector over time. Members remove a member prefix from their local vector after a certain amount of time, if they do not receive a vector containing a greater sequence number for this prefix compared to the sequence number in their local vector.

VIII. Conclusion & Future Work

In this paper, we presented DDSN, a distributed dataset synchronization protocol in NDN designed to operate under adverse network conditions. This work is based on long trials of design refactoring and performance analysis. Through this process, we were able to get an in-depth understanding of the challenges of data synchronization in adverse environments, as well as how to effectively address these challenges. This paper not only summarized the lessons learned from countless experimental attempts, but also the design merits and limitations of previous distributed dataset synchronization protocols.

Throughout our design and experimentation process, we were able to discover aspects useful for future protocol development. First, we experimented with different mechanisms to minimize redundant transmissions and we concluded that, in highly lossy environments, maximizing the utility of each message becomes vital to ensure protocol robustness. Second, we confirmed the resilience of using a state vector to synchronize the distributed dataset, as long as it is feasible for a given dataset size. Third, we reconfirmed that the old simple soft-state approach (periodic retransmission of state vector) offers resilience under adverse conditions, in lossy wired networks before and now in disruptive wireless networks as well.

Our next steps include more thorough simulation and real-world experimental analysis on the protocol performance in various settings. We also plan to build new pilot applications over DDSN to use as driver examples for its further evaluation. Finally, we plan to compare DDSN with further DTN solutions, explore adaptive ways for communication over multiple wireless hops, as well as investigate alternative approaches to enhance the scalability of the state vector structure.

REFERENCES

[1] L. Zhang *et al.*, "Named Data Networking," *ACM Computer Communication Review*, July 2014.

[2] A. Vahdat, D. Becker *et al.*, "Epidemic routing for partially connected ad hoc networks," 2000.

[3] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "A Survey of Distributed Dataset Synchronization in Named-Data Networking," NDN Project, Technical Report NDN-0053, April 2017.

[4] T. Li, W. Shang, A. Afanasyev, L. Wang, and L. Zhang, "A brief introduction to ndn dataset synchronization (ndn sync)," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 612–618.

[5] W. Fu, H. B. Abraham, and P. Crowley, "Synchronizing namespaces with invertible bloom filters," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015, pp. 123–134.

[6] Z. Zhu and A. Afanasyev, "Let's chronosync: Decentralized dataset state synchronization in named data networking," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE, 2013.

[7] M. Zhang *et al.*, "Scalable Name-based Data Synchronization for Named Data Networking," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, May 2017.

[8] W. Shang, A. Afanasyev, and L. Zhang, "Vectorsync: distributed dataset synchronization over named data networking," in *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 2017.

[9] P. de-las Heras-Quirós, E. M. Castro, W. Shang, Y. Yu, S. Mastorakis, A. Afanasyev, and L. Zhang, "The design of roundsync protocol," Technical Report NDN-0048, NDN, Tech. Rep., 2017.

[10] X. Xu, H. Zhang, T. Li, and L. Zhang, "Achieving resilient data availability in wireless sensor networks," in *Communications Workshops (ICC), 2018 IEEE International Conference on*. IEEE, 2018, pp. 7–11.

[11] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference?: efficient set reconciliation without prior context," in *ACM SIGCOMM*, 2011.

[12] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE transactions on Software Engineering*, no. 3, pp. 240–247, 1983.

[13] Z. Zhang, Y. Yu, H. Zhang, E. Newberry, S. Mastorakis, Y. Li, A. Afanasyev, and L. Zhang, "An overview of security support in named data networking," Technical Report NDN-0057, NDN, Tech. Rep., 2018.

[14] T. Li, S. Mastorakis, X. Xu, H. Zhang, and L. Zhang, "Data synchronization in ad hoc mobile networks," 2018.

[15] NDN Team, "ndn-cxx," http://named-data.net/doc/ndn-cxx.

[16] A. Afanasyev, J. Shi *et al.*, "NFD Developer's Guide," NDN, Tech. Rep. NDN-0021, 2015.

[17] S. Mastorakis, A. Afanasyev, and L. Zhang, "On the evolution of ndnsim: An open-source simulator for ndn experimentation," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, 2017.

[18] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.