A Brief Introduction to NDN Dataset Synchronization (NDN Sync)

Tianxiang Li
UCLA
tianxiang@cs.ucla.edu

Wentao Shang
UCLA
wentaoshang@cs.ucla.edu

Alex Afanasyev
Florida Int'l University
aa@cs.fiu.edu

Lan Wang
Univ. Memphis
lanwang@memphis.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

***Abstract*—Previous literatures articulated that the NDN architecture may possess unique advantages in supporting battlefield communications [1], and that NDN can provide superior performance over TCP/IP under various scenarios, in particular in supporting multiparty distributed applications [2] by using a novel dataset synchronization protocol. In this paper we first describe the designs of dataset synchronization protocols (Sync in short), in particular ChronoSync and PSync, that run over NDN to support multiparty applications, and summarize the lessons from their designs. We then perform a preliminary suitability assessment of the existing Sync protocols in ad hoc mobile environments with intermitted connectivity. Our assessment suggests that Sync protocols based on state vectors can operate most resiliently under highly dynamic conditions, and identifies remaining issues for further investigation.**

***Index Terms*—Named Data Networking, Distributed Data Synchronization**

I. Introduction

Battlefields represent a hostile environment to networking where connectivities among devices are ad hoc, intermitted, and often experience heavy packet losses. Yet successes in warfare critically depend on secure and resilient communications among all participating entities (e.g. commanders, soldiers, UAVs). The TCP/IP protocol stack works well over established network infrastructure, but faces difficulties in offering secure and resilient communications under hostile conditions.

As a newly proposed Internet architecture, Named Data Networking (NDN) [3]–[5] shows a great potential in offering secure, resilient communications. NDN lets consumers request desired data by directly using application-defined names, and fetches named data packets at network layer. This is in sharp contrast to IP's use of addresses, which name hosts (data containers) or host interfaces, thus requires mapping services to convert applications names to IP addresses before communication can happen. NDN builds public-key cryptographic protection into the architecture by requiring every Data packet carry a digital signature, which securely binds its name to the content. The above features makes NDN enable NDN to work well under highly dynamic and disrupted network conditions, making NDN a particularly well-suited architecture for tactical applications.

This work is partially supported by US National Science Foundation under award CNS-1719403, CNS-1629769, and CNS-1629922.

Since applications, generally speaking, operate over transport functions, this paper describes and examines the existing transport function supports for NDN networks. Generally speaking, transport functions bridge the gap between applications needs on data delivery and what the network layer can provide. Today's transport protocols, as examplified by TCP, convert IP's *point-to-point* datagram delivery to reliable data delivery between two processes identified by pairs of IP addresses and port numbers. Similarly, NDN Sync protocols use NDN's Interest-Data exchange primitives to support dataset synchronization for multiparty applications. By focusing on data, Sync removes the limitation on how many parties may communicate in distributed applications, and enables asynchronous communication when not all parties may be online at the same time.

Over the years a number of Sync protocols have been developed [6]. Through the process we gained a number of insights into the Sync design space. In the rest of the paper, we first briefly review the previous efforts on reliable multicast protocols in §II, and summarize the NDN Sync protocol development in §III. We then examine the feasibility of the existing Sync protocols when applied to mobile ad hoc environment in §IV, and conclude the paper in §V with discussions on remaining challenges in developing secure, resilient Sync protocols for challenged environments.

II. Previous Work on Reliable Multicast

This section gives a brief summary of previous results on reliable multicast protocol designs. These results are considered relevant as they expose the challenges in multiparty communications

A large number of reliable multicast protocols have been proposed in the past, we roughly classify them into three categories based on how they achieve reliable data reliability to multiple parties. The first category is sender-initiated protocols, where the data sender collects ACKs from each of all the receivers and performs retransmissions of lost packets. RMTP [7] is such an example. It adopts a tree-based multicast approach to avoid the ACK implosion problem, thus requires the support from intermediate servers to perform ACK aggregation.

The second category is receiver-initiated protocols such as SRM (Scalable Reliable Multicast) [8] and NORM (Nack-Oriented Reliable Multicast) [9], where each receiver is responsible for its own loss recovery, by send NACKs whenever it detects packet losses. Both protocols multicast NACKs after a random delay to suppress redundant NACKs. In addition, SRM adopts an Application Level Framing (ALF) design [10] and uses unique and persistent data identifiers. This allows the nodes in the same multicast group to help each other recover from packet losses in a more effective and robust way.

The third category takes the ALC-based (Asynchronous Layered Coding) [11] approaches, such as [12] [13]. Packets are transmitted through redundant coding, so that receivers can recover lost packets individually, minimizing the need for ACK/NACKs. However, redundant coding introduces computation overhead and increases network bandwidth consumption, and cannot completely eliminate the need for retransmissions. This approach may work well in situations where loss rates are low and the number of receivers are small.

III. NDN Sync Protocol Design

In this section, we introduce the basic concepts in Sync and describe two specific Sync protocol designs, ChronoSync and PSync, in detail. Without loss of generality, we use a text chat application, ChatApp in short, as a use case to aid the reader?s comprehension.

Sync aims to reconcile the differences of a shared dataset among multiple parties. For example, let us assume a ChatApp with multiple users, each generating chat messages. Each user *U* keeps a local view of the messages generated by all other users in the same chatroom. We refer to such a local view as *U*'s *state* of the *shared dataset* (i.e. the messages that have been generated so far in ChatApp), and refer to the collection of all the users in the chatroom as a *Sync group*. The goal of Sync is to have each *Sync group* achieve and maintain consistent *state* of the shared dataset. Equipped with the knowledge of all the available data, individual users can then decide whether, or when, to fetch missing data based on local considerations such as application priority or resource constraints. [1]

More precisely, NDN treats each chat message as a named data unit, whose name takes the form of "`/<chat-app-prefix>` `/<chat-group>/<user>/<seq-no>`" (from now on we use "`/producer-prefix`" as shorthand for all the name component before seq-no). The sequence number increases monotonically, thus every chat message is given a unique name, and is signed by the producer's key at the time of production, binding the name to the content. This enables one to authenticate all received messages, independent of from where they are retrieved. Using sequence numbers in naming data provides a concise way to represent a producer's data generation state, and to inform others by simply announcing "`/producer-prefix/<seq-no>`", with <seq-no> being the sequence number of its latest data. This naming convention also allows one to describe the shared dataset state of ChatApp as a collection of "`/producer-prefix/<seq-no>`" for all users in the chatroom.

[1]The idea of a 2-step synchronization was introduced earlier in link state routing protocol design [14], where two neighbor routers first synchronize the network topology database knowledge, and then to fetch missing data from each other.

There are some basic components common for the design of NDN Sync protocols. First, there needs to be an efficient way to name the dataset, and encode the dataset namespace in certain data structure to be transmitted over the network (state representation). Second, users need to be aware of any changes in the shared dataset, through certain flow of message exchanges (state change detection). Third, every user in a Sync group needs to compare its dataset namespace with those at other users (set difference identification) and retrieve the missing data based on application need. We will illustrate each design components separately.

- **State Representation**: in Sync, each user maintains a local dataset state, e.g., for a chatroom application, each user has a local view of the messages generated in the chatroom. The first part of state representation is namespace design. Sync represents the dataset state using the corresponding data names generated by each producer in the group (e.g. the names of chat messages generated by each user in the chatroom). Namespace design is about how to efficiently represent all the data names in the dataset. The second part of state representation is state encoding, which is about how to encode the dataset namespace into a compact format to be transmitted over the network. We refer to the encoded name set as *Sync State*. Each user calculates its local Sync State and compares it with the Sync State of the other users in the Sync group. Thus the subsequent questions that the protocol needs to address are (1) how to exchange the Sync State with other users, and (2) how to calculate the difference in Sync State.
- **State Change Detection**: as producers are generating new data, it is important for other users in the Sync group to know about the change in the shared dataset. Each user needs to send its Sync State over the network to other users in the Sync group through NDN's Interest Data exchanges. This can be done by the data producer proactively notifying other users about its state, or by each user periodically querying the Sync group for any state changes.
- **Set Difference Identification**: if a received Sync State is different from the receiver's local Sync State, a user needs to identify the specific difference(s) in the corresponding dataset namespace, e.g. a chat message received by one user but not the other causing difference in users' local chat history. Depending on the Sync State representation, this information can be deduced from the Sync States or through additional message exchanges between the users.

A number of Sync protocols have been proposed [6], each built upon the lessons learned from prior work. Here we focus on two representative Sync protocols– ChronoSync [2] and PSync [15]. At a high level, both protocols adopt the same sequential data naming convention, encode its dataset state, and multicast periodic Sync Interests to advertise dataset state to other users in the Sync group. This allows other users to detect state difference through digest comparison, and know whether they have out-of date, up-to-date, or newer dataset state. The protocols differ in the data structure for encoding the dataset state, operations to deduce the difference in dataset, and the message flow for set reconciliation. We will describe each protocol individually and then summarize their design commonalities and differences.

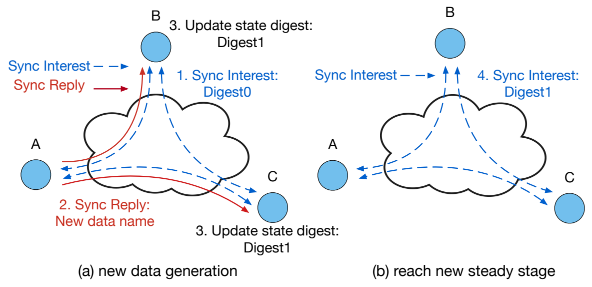
A. ChronoSync

ChronoSync [2] synchronizes a shared dataset among a distributed group of users, e.g., the chat messages of users in a chat group. For namespace design, each producer is identified by a unique producer name prefix. The data generated by each producer is named using the producer name prefix plus a monotonically increasing sequence number (starting from zero), i.e., "`/<producer-prefix>/<seq-no>`". As the sequence number is continuous, all the data generated by each producer can be represented by its producer name prefix plus the latest data sequence number rather than listing the entire set of data names, providing a condensed representation of the dataset namespace. For example, a chat group's dataset is represented by $\{(p_i, seq_i)\}$, where p_i is the chat user i's prefix and seq_i is the sequence number of the user's latest chat message. For namespace encoding, ChronoSync adopts a crypto digest data structure called *state digest*. The state digest is the concatenation of the hash value of each producer's latest data name in canonical order, and serves as a condensed summary of the dataset state.

The basic synchronization process of ChronoSync consists of the exchange of Sync Interest and Sync Reply messages. A user periodically multicasts Sync Interest containing its state digest (Sync State) to other parties in the Sync group to detect state change. If a receiver of the Sync Interest detects state difference between the received Sync State and its local Sync State, it will check whether its state is newer. If so, it will send back a Sync Reply containing its new data name. If none of the receivers detect state difference, the Sync Interest will stay pending in the network, to solicit future state changes. When a producer generates new data, it will send back a Sync Reply to satisfy the pending Sync Interest.

Once all the communicating entities are synchronized with a consistent state (stable stage), their Sync Interests will have the same name, with the same encoded state digest. Thus, the Sync Interests can be aggregated in the routers. When a producer generates new data, the Sync Reply is multicasted back to all the parties in the Sync group following the reverse paths of the pending Sync Interests.

For example in Figure 1(a), users A, B, and C are in a stable stage and have the same Sync Interest (carrying Digest0) pending in the network. When A generates new data, its ChronoSync module immediately detects its digest is newer and proceeds to satisfy the pending Sync Interest with a Sync Reply containing the new data name. This Sync Reply is multicast back to B and C to satisfy their pending Sync Interests. Then B and C each update their state digest (to Digest1), and fetch the new data based on its need. As shown in Figure 1(b), the users then enter a new stable state and each send out a Sync Interest containing the updated digest (Digest1) to solicit the next state change.



B
3. Update state digest: Digest1
Sync Interest
Sync Reply
1. Sync Interest: Digest0
A
C
2. Sync Reply: New data name
3. Update state digest: Digest1
(a) new data generation
B
Sync Interest
4. Sync Interest: Digest1
A
C
(b) reach new steady stage

Fig. 1. ChronoSync synchronization process

The stable stage is assumed to be the common case in ChronoSync – a user is either in same state as the other parties in the Sync group, or it is in a newer state after generating new data. There are a number of scenarios where this assumption does not hold.

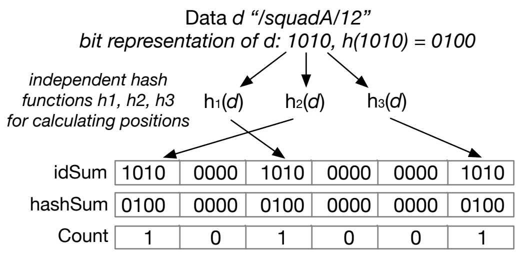
In the first case, a user might have been disconnected for a period of time before resuming connection to the Sync group, thus it will send a Sync Interest with an out-of-date state digest. To address this issue in ChronoSync, each party also keeps a log of past state digests and the corresponding changes leading to each digest. When a receiver recognizes the out-of-date digest by looking through its digest log, it sends a Sync Reply containing the names of the data that the sender is missing.

In the second case, out-of-order delivery might cause users to receive an unrecognizable state digest. For example, a user might have received a new Sync Interest containing an updated Sync State, before receiving the previous Sync Reply that leads to the updated Sync State. To address this issue, ChronoSync adopts a randomized wait timer based on the propagation delay to postpone processing of an unknown digest.

The third case is state divergence, where different nodes have accumulated different state updates locally, making the digest unrecognizable to other diverged nodes. One example for this case is network partition, where nodes are disconnected for some time, and have accumulated different state updates. Another example is when more than one producer have generated new data and reply to the Sync Interest simultaneously. In this case, only one of the replies will be received by a user, as one Interest brings back only one Data in NDN. This causes state divergence for users receiving different replies. To address this issue, ChronoSync adopts a recovery mechanism in which the receiver of the unknown digest sends a Recovery Interest to fetch from the sender of the digest the complete dataset state information. After a few rounds of recovery, the group will enter stable state again.

In summary, ChronoSync's design choices are as follows:

- **State Representation**: (a) sequential data naming; (b) crypto digest for state encoding;



Data d "/squadA/12"
bit representation of d: 1010, h(1010) = 0100
independent hash functions h1, h2, h3 for calculating positions
h1(d)
h2(d)
h3(d)
idSum
hashSum
Count

idSum	1010	0000	1010	0000	0000	1010
hashSum	0100	0000	0100	0000	0000	0100
Count	1	0	1	0	0	1

Fig. 2. PSync data name insertion into IBF. When an element is inserted, hash functions (h_1, h_2, h_3) are applied to the element to calculate the cells in the IBF which the element corresponds to. For each cell, the element's keyID (1010) is XOR'ed with the cell's idSum value, the element's hash value (0100) is XOR'ed with the cell's hashSum. The cell's count increments by one. The deletion process is similar, except the cell count is decreased by one.

- **State Change Detection**: (a) Sync Interest carrying state digest; (b) Sync Reply containing new data name; (c) Sync Interest staying pending in the network to solicit future state changes;
- **Set Difference Identification**: (a) digest log; (b) recovery mechanism that exchanges full dataset state information.

B. PSync

PSync [15] is designed to support partial dataset synchronization: it allows a consumer to subscribe to a subset of data streams, where a data stream is a sequence of data items under a common name prefix. For example, in ChatApp, the command center may generate command messages for air forces and ground squads under different data stream name prefixes. PSync can also support full sync as ChronoSync does, by making each participant both a producer and a consumer which subscribes to all data streams generated by all producers in a sync group. Since most operations are similar in both partial and full sync supports, below we focus on full sync operations.

Similar to ChronoSync, PSync names each item in a data stream using a unique stream prefix with a monotonically increasing sequence number. It also represents the dataset namespace by the collection of the latest data name of each stream in the set. PSync uses Invertible Bloom Filter (IBF [16]) to encode the dataset state by adding the hash value of each data stream's latest data name to the IBF. Figure 2 shows an example of encoding data names into the IBF. Each entity maintains a local mapping table between the data name of each stream and its corresponding hash value for decoding the IBF elements. The size of the IBF determines how many elements it can decode through subtraction operation, for a given probability of false positive output [16]. IBF encoding enables the detection of different states, and IBF subtraction operations can also discover the differences in elements contained in two different IBFs, bar the possibility of false positive.

Each PSync participant multicasts Sync Interests, which contain the sender's IBF, to the group periodically. Upon receiving a Sync Interest, one performs IBF subtraction operation to decode the differences between the local IBF and the received IBF. This operation outputs the hash values of the different data names between the local IBF and the received IBF. One then looks up its mapping table to find the corresponding data names of those hash values, and sends them out in the Sync Reply. With multiple receivers in the same sync group, each of them may send back a Sync Reply. However due to NDN's one Interest for one Data exchange rule, only one of these replies will be received by the original sender of the Sync Interest. Thus after processing the Sync Reply and updating its IBF, the original sender will immediately multicast a new Sync Interest to the group. Any user in the Sync group may respond again if detecting state difference between its own IBF and the one in the received Interest. This process repeats until all the parties are synchronized.

The state differences between the sender and receiver of a Sync Interest can be decoded as long as the differences do not exceed the maximum that is determined by the IBF size. If a large amount of state differences gets accumulated, only some of the differences may be decoded. In such cases, PSync sends back the decoded differences in Sync Reply and continues with additional iterations. Evaluating PSync performance in this case is part of our future work.

In summary, PSync's design choices are as follows:

- **State Representation**: (a) sequential data naming; (b) an IBF that contains hashes of (data prefix, latest sequence number) for all the data streams;
- **State Change Detection**: (a) Sync Interests carrying IBFs; (b) Sync Reply containing differences in data names; (c) Interests being sent periodically to detect state changes;
- **Set Difference Identification**: IBF subtraction.

C. Sync Security

Sync protocols raise a unique security requirement as we explain below. By default, NDN Interest packets are not signed, as sending an Interest to fetch data does not have side-effect to producers in general. [2] However, Sync Interests used in Sync protocols signal dataset state changes, which trigger actions on the receiving ends. Therefore Sync Interests need to be authenticated to prevent malicious attackers from injecting false information into the system.

1) Sync Interest Authentication: If symmetric key is used in Sync Interest authentication, each Sync Interest would be signed by the public key of its sender. This public key signature makes each Sync Interest name unique, thus preventing Interest aggregation in the network.

To use symmetric key to authenticate Sync Interests, one may distribute a shared symmetric group key to all the parties in a Sync group, which can then be used to generate HMAC for each Sync Interest, with the HMAC as part of the name carried in the Interest. All the parties with the same state representation will generate the same HMAC value, thus their Sync Interests can be aggregate. HMAC computations are also much cheaper compared to public key signature. One

[2]When an Interest is used to carry out actions, e.g. a command Interest to turn a light on, it will be signed [17]

main issue of using symmetric key authentication is the key generation and distribution process. Below we describe a simple design of having a group manger to handle the key management.

2) Group Key Management: A simple way to manage a shared group key is to have a single group manager responsible for the generation, distribution, and update of the shared key. In distributed applications such as chatroom messaging, a leader (e.g., the creator of a chatroom) can act as the group manager and distribute keys to others in the Sync group, an approach used in [18]. The group manager can securely distribute the symmetric key to all parties in the Sync group, by using each participant's public key to encrypt the shared symmetric key, and then publishing these encrypted keys to let participants fetch as any other published data.

The symmetric key can be updated periodically or on demand depending on actual need. When a new key is generated, the group manager can notify the Sync group through key advertisement messages. In case of intermittent connectivity, where some parties fail to be notifed, they may receive Sync Interests with unrecognizable HMAC values, which serves as a signal for them to fetch the new key.

The above solution is simple and straightforward to implement, however it raises new questions of the group manager selection and failure recovery, that need to be addressed.

IV. Sync in Mobile Ad Hoc Networks

Networking in mobile ad hoc environment (MANET) with continuous node movement and intermittent connectivity has been a decade long research topic. Our earlier work [1] suggests that NDN may possess unique advantages in enabling MANET. In this section we investigate what are the fundamental challenges for a Sync protocol to operate in such highly dynamic environments.

The basic difference between networking over a stable infrastructure and through ad hoc mobile connectivity is the lack of persistent connectivity in the latter case. We illustrate the impact of this lack of persistent connectivity through a battlefield communication example.

Fig 3 shows an example battlefield scenario, where different mobile entities need to coordinate and communicate with each other. The satellite, aircraft gateway, and HMMWV gateway act as NDN packet forwarders, offering intermittent connectivity to other nodes in the figure. Let us assume that each of the other entities runs a chat application ChatApp. The command center, ships, fighter squadron, and squads send chat messages to exchange commands and battlefield information with each other while moving around, each acting as both chat message producers and consumers.

A. State Divergence Recovery

Intermittent connectivity in MANET makes Sync Interest losses as the norm instead of low probability events, resulting in state updates not being propagated to all, or most of, the other nodes within the Sync group. Consequently, their dataset states are more likely to be inconsistent than not.

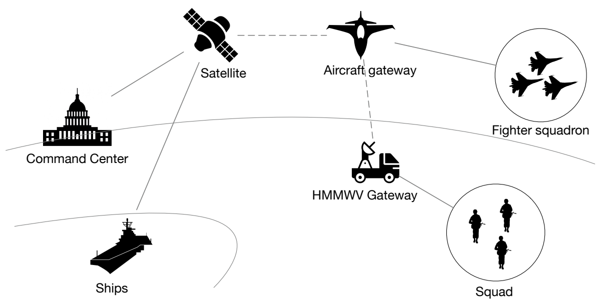


Fig. 3. Example Scenario

State Change Detection: The periodic transmission of Sync Interest by ChronoSync and PSync can act as a periodically detection of state inconsistencies. This detection works effectively even when nodes move around, and one's connectivity to a new neighbor is rather short-lived. Periodic Sync Interest transmissions offer a robust solution to state change detection. However recovering from state divergence becomes more challenging. For example, when disconnected, the airforce and ground squads might have accumulated a number of different data messages regarding battlefield status, causing their local chat message sets to diverge. Once they are connected again, they need to reconcile quickly the diverged dataset over (most likely) short-lived connectivity.

Reconciliation via State Digest: ChronoSync uses a crypto digest as a compact representation of state, allowing efficient state difference detection. But since crypto digest comparison does not directly identify the difference in each nodes' dataset namespace, and the engineered solutions of change logs and short random wait are ineffective for large divergence, receiving an unrecognizable state digest will trigger the recovery process mentioned previously. The recovery process likely requires multiple rounds of packet exchanges, that may not finish when the short-lived connectivity breaks. What we can learn from the above is that, if a Sync Interest depends on the receiving node's internal state to derive the data name differences, the protocol can be made ineffective by highly dynamic environment.

Reconciliation via Invertible Bloom Filter: IBF allows two nodes to identify the differences in dataset namespace through IBF subtraction and mapping table lookup. However as mentioned earlier, the size of the IBF limits how much state difference can be decoded from the IBF operation. In case not all the differences can be decoded from the IBF calculation, multiple rounds of Sync Interest and Reply exchanges may be needed in order to enter a new consistent state eventually, leading to the same problem as ChronoSync. We can conclude that advanced encoding of dataset state can be advantageous in well understood environments and may show its limitations in highly dynamic environments.

a) Reconciliation via State Vectors: [19] proposed an alternative approach to dataset reconciliation, which is further elaborated by [20]. The basic idea is to represent a dataset state in the form of a version vector, with each component being the latest data name of each producer. Note this State Vector solution also adopts the sequential naming convention used by ChronoSync and PSync, where each node is identified by an unique producer name prefix and the data generated by that node is named using the producer prefix and a monotonically increasing sequence number.

Because a State Vector directly enumerates the producer prefix and latest data sequence number for each producer in a Sync group, as shown in Figure 4, the difference in dataset names can be directly identified by comparing the sequence number of each producer prefix. A node then updates its State Vector by taking the higher value of each producer's sequence number in the State Vector.

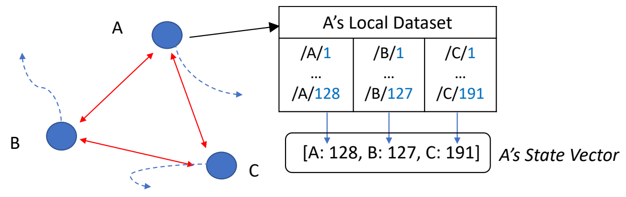


Fig. 4. State Vector Example. Chat user A has a local dataset of the messages generated by chat users A, B, and C. The chat user producer prefix and the latest chat message sequence number is contained directly in the State Vector as two-tuple value pairs.

State Vector based state representation allows direct dataset state comparison without any assumptions on the receiving ends state, and tolerates all degrees of state divergence. By comparing State Vectors, nodes can directly deduce the data names it is missing without any additional message exchanges. This is ideal in the mobile ad hoc environment where data diverges, and even large divergence, can be the norm instead of exceptions.

The size of a State Vector goes up proportionally with the number of producer states to be exchanged. However, explicit listing of producer states also allows one to exchange a partial State Vector under packet size constraints, to include only the state information of those most important producers (e.g. chat messages from the commanders).

B. State and Data Synchronization

As mentioned previously, NDN Sync keeps applications up to date about the latest dataset state (state synchronization), and lets applications decide what data to fetch based on its need (data synchronization). Considering the heterogeneity among members of a large sync group, this decoupling of state and data synchronization is an important design decision.

However in MANET, the lack of persistent connectivity between nodes makes it difficult to fetch missing data after a node learns a piece of new data has been generated. As a result, nodes are unclear what data it can fetch from its connected neighbors, causing excessive Interests to be sent requesting for data which is not available. This mismatch between a node's newest state and its actual dataset gets exacerbated as the newest state is propagated in the network. One possible solution to this issue is to couple the state and data synchronization process together, by which nodes to send Sync State reflecting its actual dataset in the Sync Interests. Another possible solutions is to deploy distributed repos to increase data availability. It is an ongoing work to address the issues of state and data synchronization in MANET, and is an area worth further exploring.

V. Conclusion

In this paper we summarized the key design features of existing NDN Sync protocols, assessed their suitability of operating hostile environments such as battlefields, and identified both new design directions and remaining challenges. We observed that, in an ad hoc mobile environment with intermittent connectivity, most resilient communications can be achieved by Sync protocols that can fully utilize intermittent connectivity by operating in an itempotent way.

We also examined remaining challenges in Sync protocol design, which include scalability and security. Regarding scalability: using IBF to encode sync state can identify exact differences in shared dataset, however the effectiveness of IBF depends on its size and the amount of state differences; it also remains open whether iterative difference discovery would work well in highly dynamic environment. State vector based designs support itempotent operations, however the vector size can be a concern. Regarding Sync security, we need robust solutions to manage group shared keys. We are actively developing solutions to address these two challenges.

References

[1] C. Gibson, P. Bermell-Garcia, K. Chan, B. Ko, A. Afanasyev, and L. Zhang, "Opportunities and Challenges for Named Data Networking to Increase the Agility of Military Coalitions," in *Proceedings of Workshop on Distributed Analytics InfraStructure and Algorithms for Multi-Organization Federations (DAIS)*, 2017.

[2] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proc. of IEEE ICNP*, October 2013.

[3] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking Named Content," in *Proc. of CoNEXT*, 2009.

[4] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM Computer Communication Reviews*, Jun. 2014.

[5] A. Afanasyev, J. Burke, T. Refaei, L. Wang, B. Zhang, and L. Zhang, "A Brief Introduction to Named Data Networking," in *IEEE MILCOM 2018*.

[6] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "A survey of distributed dataset synchronization in Named Data Networking," NDN, Technical Report NDN-0053, May 2017.

[7] J. C. Lin and S. Paul, "RMTP: A reliable multicast transport protocol," in *IEEE INFOCOM*. IEEE, 1996.

[8] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *ACM SIGCOMM*, 1995.

[9] B. Adamson, C. Bormann, M. Handley, and J. Macker, "Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol," Tech. Rep., 2004.

[10] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4. ACM, 1990, pp. 200–208.

[11] M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, and J. Crowcroft, "Asynchronous layered coding (alc) protocol instantiation," Tech. Rep., 2002.

[12] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 56–67, 1998.

[13] J. Nonnenmacher, E. W. Biersack, and D. Towsley, “Parity-based loss recovery for reliable multicast transmission,” *IEEE/ACM Transactions on Networking (TON)*, vol. 6, no. 4, pp. 349–361, 1998.

[14] J. Moy, “Ospf specification,” Tech. Rep., 1989.

[15] M. Zhang, V. Lehman, and L. Wang, “Scalable Name-based Data Synchronization for Named Data Networking,” in *Proceedings of INFOCOM*, 2017.

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

[17] NDN Project Team, “Signed Interest,” https://named-data.net/doc/ndn-cxx/current/specs/signed-interest.html, 2018.

[18] Z. Zhu, J. Burke, L. Zhang, P. Gasti, Y. Lu, and V. Jacobson, “A new approach to securing audio conference tools,” in *Proceedings of the 7th Asian Internet Engineering Conference*, 2011, pp. 120–123.

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

[20] X. Xu, H. Zhang, T. Li, and L. Zhang, “Achieving resilient data availability in wireless sensor networks,” in *IEEE ICC Workshops*, 2018.