NDN Technical Report NDN-0073. http://named-data.net/techreports.html
Revision 3: Oct 2024

A Brief Introduction to State Vector Sync

Philipp Moll, Varun Patil, Nishant Sabharwal, Lixia Zhang
UCLA
Los Angeles, USA
{phmoll,varunpatil,nsabharwal,lixia}@cs.ucla.edu

ABSTRACT

This report provides a brief introduction to State Vector Sync (SVS), a Sync protocol for Named Data Networking (NDN). To support distributed applications, sync protocols synchronize the data names of a shared dataset among a group of participants. In this report, we explain how the SVS design is influenced by the lessons that have been cumulated over previous sync protocol designs and describe the protocol and its functions to allow experimentation with the SVS library implementations.

VERSION HISTORY

Revision 1 (May 2021): Description of the initial SVS design.
Revision 2 (July 2021): Updated the protocol description and added the processing flowchart (Fig. 3) to aid the comprehension.
Revision 3 (Oct 2024): Update for new wire format and design discussion on timer optimizations and scaling.

1 INTRODUCTION

In Named Data Networking (NDN) [14], applications communicate by requesting named, secured content chunks. To do so, one needs to know the set of available data names. This goal is easy to achieve with the traditional client-server application paradigm, where the server informs the client of the available data. However, in a distributed application with multiple participants, where any of them may produce new data items at any time, it is challenging to keep all participants synchronized with all available data.

A Sync protocol addresses this challenge for applications developed over NDN. A group of participants in the same sync group maintains a shared dataset, with each data item having a unique name. The role of sync is to keep the dataset state – i.e., the names of all data items – synchronized among all the participants.

Compared to other Sync protocol designs that preceded SVS, a distinct goal in the SVS design is the desire to operate *efficiently* and *resiliently* in both infrastructure-based and infrastructure-free environments. In the latter case, network infrastructure is either non-existent, e.g., ad-hoc mobile, or otherwise disrupted, e.g., during disaster recovery.

In this report, we start with providing a brief background on Sync protocols with a focus on lessons learned from previous developments. We then describe the SVS protocol, the ongoing efforts in extending SVS scalability, and finally wrap up the report with the remaining issues and future plans.

2 NDN SYNC PROTOCOL DESIGN

Over the years, a variety of NDN Sync protocols have been developed. The first sync protocol (CCNx 0.8 Sync [8]) supports the synchronization of datasets made of application data names that follow a hierarchically structured tree. The protocol performs hashing at each node of its direct child node data names, and uses the digest at the tree's root node to represent the dataset state. Each participant communicates its dataset state with others in the group using its root digest. Receiving a digest from another participant that differs from the local digest indicates a dataset state inconsistency. However, the different digest does not tell whether the local or the remote dataset is newer, nor exactly which data item caused the difference; the problem gets worse when multiple participants publish data simultaneously. When a digest difference is detected, the protocol walks down the tree level by level, branch by branch, to identify dataset state differences. This step may take multiple rounds.

2.1 Use of Sequential Naming in Sync

Instead of supporting the synchronization of arbitrary data names, the *ChronoSync* [15] protocol adopts the sequential data naming convention and names data items using sequence numbers, similar to TCP's use of sequence numbers in its reliable delivery mechanism. With sequential data naming, every participant publishes data under a participant-specific publishing prefix, and names individual data items with monotonically increasing sequence numbers. Knowing the participants publishing prefix and its latest sequence numbers thereby allows inferring the names of all the participant's previously published data items. Consequently, knowing the `[participant-prefix, seq#]`-tuples of all participants allows inferring the names of all data items in the dataset. Similar to CCNx 0.8 Sync, ChronoSync uses a digest to represent the dataset state, with the digest computed across all `[participant-prefix, seq#]`-tuples in the sync group. Thereby, ChronoSync inherits the limitation that additional means to identify dataset differences are required. However, using sequential naming brings advantages in dataset state reconciliation: instead of walking down the name tree to identify data item differences, ChronoSync uses a simpler recovery mechanism. If participant P cannot figure out the dataset difference with a received digest D, P requests the list of `[participant-prefix, seq#]`-tuples from the sender of D.

A vigilant reader might raise a question regarding the use of sequence numbers as data item identifiers: although sequence numbers simplify a sync protocol design, in general, sequence numbers cannot replace semantic names of application data. We discuss this mismatch in Section 4.1.

2.2 Vector-Based Sync Protocols

The branch of state vector-based sync protocols is inspired by the concept of Vector Clock [2]. Combined with the sequential data naming convention, this protocol family encodes the dataset state in so-called state vectors – a data structure storing the latest sequence

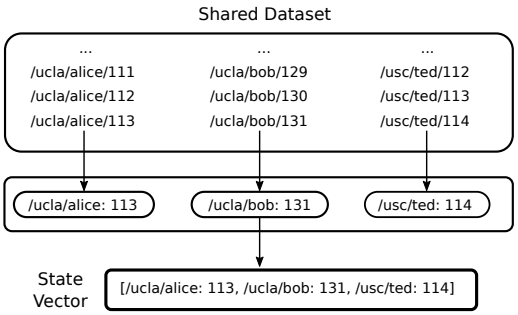

Figure 1: Relation between the Dataset State and the Representation as a State Vector

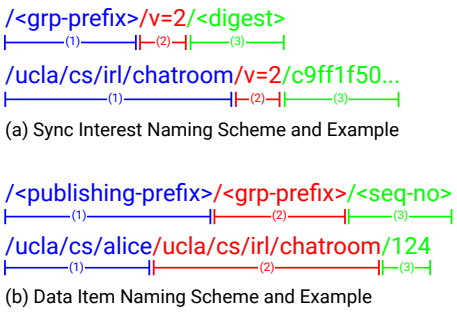

Figure 2: Naming for Sync Interest and Data Items

number of every participant as a vector. In contrast to digest-based protocols, a state vector encodes the state of the entire dataset, making it possible to directly infer the exact difference(s) when comparing two state vectors. Fig. 1 visualizes the relation between sequential data naming and the dataset encoding using state vectors.

The first state vector-based protocol is *VectorSync* [9]. VectorSync maintains two separate data structures. A *membership info object* summarizes information about all active producers in the Sync group. The *version vector* encodes the latest sequence numbers of each producer. This version vector, however, does not include participant-specific publishing prefixes and requires that all participants in a Sync group have the latest membership info object. Otherwise, the individual vector entries cannot be assigned to the correct participant.

Since NDN aims to enable asynchronous communications, requiring perfectly consistent membership information among all group members is deemed infeasible. Therefore, VectorSync was quickly followed by another sync protocol *DataSet Synchronization in NDN* (DSSN) [13], which made a simple yet significant change to VectorSync that supports dataset synchronization among a group of sensors that enter a sleeping state from time to time. DSSN changed the vector format from a list of sequence numbers to a list of `[participant-prefix, seq#]`-tuples. Each DSSN Sync Interest carries a state vector, which directly encodes the entire shared dataset state. Directly carrying the dataset state enables a DSSN message to be interpreted by *any* recipient, independent from the degree of state inconsistency between participants.

DSSN is designed to work in environments with intermittent connectivity among stationary nodes. Using DSSN as a starting point, the *Distributed Dataset Synchronization over Disruptive Networks* protocol (DDSN) [1] extended DSSN to work in wireless ad-hoc environments with high node movement dynamics. Therefore, DDSN introduces a number of features tailored for such target environments, including i) transmission prioritization that determines which messages to send first during short transient connectivity between nodes, and ii) an inactive mode to reduce traffic and to improve on energy consumption when participants detect no others within operating distance. The exclusive focus of DDSN on disruptive environments, however, makes it perform sub-optimally in well-connected networks.

Combining the lessons learned from the aforementioned protocols led to the development of *State Vector Sync* (SVS). SVS inherits DSSN's State Vector encoding of Sync Interests but removes features that are specifically tailored for sensor communications and disruptive environments. Also, SVS further simplifies the overall design as we explain next.

3 THE DESIGN OF STATE VECTOR SYNC

In SVS, a Sync group uses a Interest multicast group prefix that allows reaching all other participants in the Sync group. Moreover, each participant uses a participant-specific data publishing prefix under which the participant's data items are made available. The protocol uses a single type of message which is referred to as a *Sync Interest*, for dataset synchronization. Sync Interests are sent to the multicast group prefix in two cases: i) *event-driven* to inform other participants about a recent change, and ii) *periodically*, to maintain a consistent view on the dataset, even under loss of event-driven messages.

Sync Interests carry the state vector in the Application Parameters field of the Interest. To prevent unauthorized parties from injecting incorrect state, Sync Interests are authenticated using Interest signatures [5]. We illustrate the naming scheme for sync interests and an example name in Fig. 2a.

As indicated in Fig. 1, SVS's state vector contains tuples consisting of the participants' data publishing prefixes and their latest sequence numbers. The naming scheme for data items and an example name are illustrated in Fig. 2b. While the publishing prefix component (1) supports forwarding the Interest towards the data producer, the group prefix component (2) allows dispatching interests to the corresponding application on a processing host.

3.1 Sync Interest Processing and Generation

Each member of a Sync group sends a Sync Interest under two conditions: i) event driven, i.e., the node identifies a new dataset state change, or receives a Sync Interest with obsolete dataset state; and ii) time driven, i.e., periodically, in the absence of event-driven Sync Interest generation. The former is used to disseminate new dataset state information through the network with minimal delay, while the latter is to maintain the group dataset state consistency in the face of packet losses and transient network disconnections.

Event-driven Sync Interests aim to keep the group synchronized about the shared dataset state, however, a member's dataset state can potentially get out of sync with that of the others due to various unforeseen causes, such as packet losses, temporary link failures,

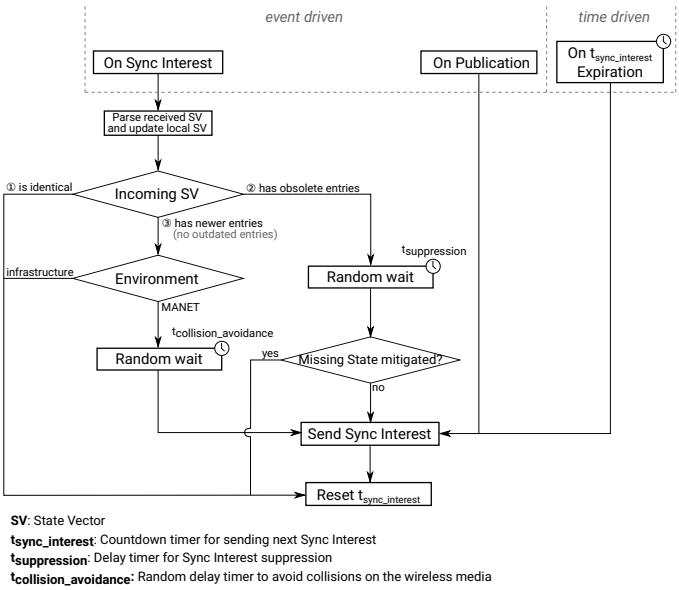

Figure 3: SVS's Sync Interest Processing Pipeline

network partitions, or even node failures and recoveries. To address this problem, each Sync entity maintains a Sync Interest timer to trigger periodic Sync Interests, which help keep the group members synchronized for the shared dataset state in the face of such unforeseen issues. The periodic timer gets reset after an event-driven interest is sent or received, as we explain below.

Now let us examine event driven Sync Interests. An SVS event can be triggered in two cases. The first case is when a participant P produces a new piece of data. P will increase its sequence number by one to name this new piece of data, and immediately emit a new Sync Interest carrying the new dataset state. It will also reset its timer for periodic sync interests. When the other sync nodes receive the Sync Interest, they will notify their local applications of the new data name. Note that NDN Sync keeps the group synchronized in the shared dataset state; it is up to the application to decide whether, or when to fetch newly produced data.

The second case of event generation is when P receives a Sync Interest I_{recv} sent by others in the group. The processing logic of a received Sync Interest is illustrated in Fig. 3. The receiver first assesses whether the state vector carried in I_{recv} contains the same, older, or more recent state information, by directly comparing the sequence number under each member prefix in the received vector with the sequence numbers in its local state vector.

If the received Sync Interest I_{recv} is *identical* to P's dataset state (branch labeled with ①) P simply resets its periodic Sync Interest transmission timer to the original full period value. This is because someone else just notified the group of the same dataset state, there is no rush for P to repeat the same information soon after. We refer to this process as Sync Interest suppression, and it helps keep the packet overhead of SVS to a minimum.

If I_{recv} contains *obsolete* information (that is, one or more producers' sequence numbers are smaller than that of P's[1]), P needs to notify whoever sent the obsolete dataset state by sending a Sync Interest with its own latest dataset state, as indicated by the branch labeled with ②. However P must keep in mind that other members in the Sync group may have also received the same Sync Interest and planned to act on it. To minimize duplicate Sync Interest transmission, P will wait for a random time period before sending the Sync Interest, referred to as the *suppresssion* time. If P receives a Sync Interest during this waiting time which is identical to or carries newer information than the one it plans to send, P cancels its own transmission; otherwise, it sends its Sync Interest upon random timer expiration.

If I_{recv} carries information about new data that P is unaware of (branch labeled with ③), i.e., the sequence number under some member prefix is higher than P's local vector and none is lower than P's local vector, or the received vector contains new member prefixes, P merges the received vector with its own dataset state. Whether P should immediately notify others about its updated dataset state depends on the environment P is currently in. If P is infrastructure connected and has no mobile neighbors, P can trust that other members in the Sync group most likely received the new dataset state through the network's multicast delivery. In this case, it simply resets its timer for the next periodic sync interest. On the other hand, if P is in a MANET setting, P needs to help further propagate the new dataset state by passing the sync interest to mobile neighbors, or newly encountered neighbors. To reduce collisions on the wireless broadcast media resulting from multiple participants sending the Sync Interest simultaneously, we introduce a random collision-avoidance timer (in the range of a few milliseconds), as suggested by Wang et al. [12].

4 DISCUSSION

In this section, we discuss some of our important design decisions in further detail.

4.1 Sequence Numbers

A comparative evaluation [3] of Sync protocols has shown that sequence numbers are essential for scaling a Sync protocol's frequency of dataset changes in a single Sync group. Without sequence numbers, information about each dataset change must be propagated to every member in the group individually. With sequence numbers, a single packet exchange can notify a receiver about multiple dataset changes, greatly reducing sync latency and protocol overhead.

Taking the example of SVS, if one or more Sync Interests are dropped, members in the Sync group may have an older local state vector. In this case, receiving a single SVS Sync Interest is sufficient to update the state to the latest, without any further exchanges. Thus, SVS can recover from the loss of multiple packets from different group members using a single packet, which in turn may not originate at any of these members.

[1]It is possible that I_{recv} may contain some other producers' sequence numbers that are higher than that of P's.

While sequence numbers are essential to transport layer (i.e. Sync) functionality, applications typically desire to utilize and synchronize semantic application layer data names. This important requirement can be fulfilled by Sync through the use of higher layer protocols built over Sync, such as SVS Pub/Sub (SVS-PS) [4]. The SVS-PS protocol provides support to directly notify consumers about the newly produced data names in the Sync group, hiding transport layer details such as sequence numbers from application developers.

4.2 Multicast Interests without reply

As a significant departure from earlier vector-based Sync protocol designs, SVS Sync Interests are used as one-way notification only, and do not trigger reply Data packets. This design decision is based on the lessons learned from previous Sync protocol designs. All participants in a Sync group multicast Sync Interests to the group. In previous designs, these multicast Interests solicit Data in reply, and this Data notifies the Interest sender about dataset changes. Such a design that pulls dataset changes leads to three issues.

(1) Given the time of next dataset state change is unpredictable, a reply-soliciting Sync Interest stays pending on all forwarders (long-lived) until its lifetime expires, and gets refreshed by a follow-up sync interest. This creates a persistent PIT state from every member to every other member in the Sync group, impacting scalability.
(2) A *multicast* Interest solicits a reply from each of *multiple* potential producers. If multiple producers reply around the same time, only one of the replies is delivered to the Interest sender due to NDN's one-Interest-one-Data principle.
(3) As a consequence of the above, different members in the group are likely to receive different updates with a high update frequency, leading to dataset state divergence. This diverged state can now take multiple Interest-Data exchange cycles to converge due to the same reason.

Instead of using multicast Sync Interests to solicit replies, SVS uses multicast Sync Interests to let each participant *notify* the rest of the group of its own dataset state. Removing replies to multicast Interests from Sync design resolves all the above-identified issues.

4.3 Security & Group Membership

One might consider the management of group membership as part of Sync. However, we argue that membership management should not be part of the transport layer. Deciding whether a Sync Interest sender is authorized requires information that is available only at the application layer. Some higher-level libraries [11] can provide support for this verification using standard NDN security mechanisms. Although not part of the conceptual design of SVS, we aim to integrate SVS as the transport in such libraries.

4.4 Timer Tuning

SVS utilizes a *single* timer as the loss recovery mechanism. This timer can take on different values depending on the state of a given Sync participant.

(1) In steady state, the timer is set to the relatively longer periodic timer value. The longer period reduces the number of frivolous packets sent across the network when all participants in the Sync group are already converged.
(2) When an outdated Sync Interest is received, the timer is set to the short suppression timer value. The random suppression timer ensures that the overhead of SVS stays low, by reducing the number of Sync Interests sent in response to an outdated Sync Interest. In the ideal case, only a single Sync Interest would be triggered in response to an outdated Sync Interest, and all others would be suppressed.

At any time, the SVS timer is reset to the periodic timer value when an up-to-date Sync Interest is received at a participant. If the timer expires, the node sends out a Sync Interest with its local state vector.

The precise values that participants pick for this timer can greatly affect the performance of the protocol [6]. The specification of SVS details several optimizations for timer settings and Sync Interest generation. We briefly describe some of these optimizations below.

4.4.1 *Periodic Timer Period.*

The periodic timer of SVS is used to correct any inconsistent state due to lost Sync Interests, in the absence of any new data production. If a SVS participant sets the timer to the periodic value, it gets reset on receiving *any* Sync Interest from another participant, regardless of the contents. As a result, any data production in the group automatically resets the periodic timer.

The ideal periodic timer value at a Sync participant depends on the application and network conditions. The discussion below serves as a reference for developers to choose parameters suitable for their application.

(1) In all cases, it is recommended to randomly choose a different value at each timer period with a moderate random jitter. For example, an application may choose a period of 30s ± 10%. Such a range guarantees that all participants usually spend time approximately equal to the period before triggering any Sync Interest. Due to the random jitter, the participant that picks the shortest value sends the first Sync Interest, resetting the timer at all other participants.
(2) A shorter period is more likely to correct any inconsistent state, at the cost of higher periodic packet overhead. However, if the frequency of data publication is high, the overhead would likely not increase significantly, since the periodic timer would never expire.
(3) A shorter period would greatly benefit SVS participants in ad-hoc networks with poor connectivity. If a group member can transiently communicate with another group member, a shorter period increases the likelihood of a state exchange during this transient connectivity period, improving dataset consistency.

We note that under dynamic conditions, it may be possible to further optimize the timer range depending on factors such as the current data production rate in the group.

4.4.2 *Suppression Timer Period.*

The function of the suppression timer is to ensure that only one Sync Interest is sent to the network in response to an outdated Sync Interest. On receiving an outdated Sync Interest, each participant individually picks a random timer value in a given range. We observe that, in the ideal case, this suppression time should be –

(1) Very short or zero at exactly one participant P_a, and much longer at all other participants. Such a case would result in most participants being very likely to receive P_a's Sync Interest before their own suppression timer value expires.
(2) At least as large as the network diameter latency, for all participants except P_a. This would ensure that P_a's Sync Interest can reach all other participants in time.

In practice, we cannot achieve this ideal case under dynamic conditions, and thus we must make some approximations. To help only a small fraction of participants pick a small suppression time, we suggest using the follwing exponential curve to pick the random value.

$$T = C \times (1 - e^{\frac{V-C}{C/F}})$$

where:

T is the calculated suppression timer value
V is a uniform random value in $(0, C)$
C is a constant *Suppression Period*
F is a constant *Decay Factor*

The two constant parameters in the curve must be approximated depending on network conditions. C is the maximum suppression time that any node may choose, and thus must be equal to at least the network latency diameter. F must be positively correlated to, and can be set equal to, to the number of active participants in the Sync group.

Picking the right suppression timer value is crucial for good performance during loss recovery. Learning parameters such as the constants in the equation above from network conditions dynamically is part of ongoing SVS research.

4.4.3 *High Frequency Optimization.*

Some Sync groups may produce data at very high rate, so that each producer's sequence number changes multiple times every second. SVS generates a Sync Interest for each change in the sequence number, which can flood the network with multicast Interests.

To optimize for high data production rates, we suggest setting the SVS timer to a short period T_h on any dataset change instead of triggering a Sync Interest immediately. During this period, any further dataset changes should be ignored and incoming Sync Interests should be ignored for timer calculations. When the timer expires, the participant should send a Sync Interest with the latest known state vector and reset the timer to the periodic timer value.

This optimization ensures that a much smaller number of Sync Interests are transmitted in the network, at the cost of increasing Sync latency by T_h. This cost is often acceptable and can be compared to buffering in TCP/IP, and applications can explicitly indicate any latency-sensitive data production.

4.5 Scalability

Looking at the design of SVS might raise scalability concerns, because the state vector carries the entire dataset state in the Sync Interest packet. A large number of Sync participants leads to a big state vector size, and Interests have a strict upper size limit defined by the network's Maximum Transmission Unit (MTU).

We first note that efficient state vector encoding and compression schemes may help alleviate this concern to a certain degree. Since a large number of producers in a group may have a common name prefix, we have observed that simply compressing the entire state vector can be highly effective in certain cases.

However, the most promising direction is to utilize SVS's property that each `[prefix, seq#]`-tuple is independent from other pairs. Therefore, each Sync Interest is not required to carry the full dataset state, and can carry a *partial* state vector instead, as proposed in p-SVS [7]. Scaling SVS using such a partial state vector strategy and its interplay with the Sync Interest suppression mechanism is currently part of ongoing SVS research.

5 WRAPPING UP

This technical report introduces the functioning of SVS and provides discussion on various performance and scalability optimizations that applications can leverage. Evaluations [3] have shown that SVS can improve on traffic and computational overhead compared to other Sync protocol implementations.

We highlight the availability of the online specification of SVS [10]. Furthermore, the aforementioned reference features open-source SVS libraries in different programming languages and refers to demo applications showcasing the use of SVS.

ACKNOWLEDGMENTS

We would like to thank Justin Presley from Tennessee Tech for his efforts in developing the Python implementation of SVS. This work is partially supported by the National Science Foundation under award CNS-1719403.

REFERENCES

[1] Tianxiang Li, Zhaoning Kong, Spyridon Mastorakis, and Lixia Zhang. 2019. Distributed Dataset Synchronization in Disruptive Networks. In *16th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (IEEE MASS)*. IEEE, 10. https://doi.org/10.1109/MASS.2019.00057
[2] Barbara Liskov and Rivka Ladin. 1986. Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada) *(PODC '86)*. ACM, 29–39. https://doi.org/10.1145/10590.10593
[3] Philipp Moll, Varun Patil, Lan Wang, and Lixia Zhang. 2022. SoK: The evolution of distributed dataset synchronization solutions in NDN. In *Proceedings of the 9th ACM Conference on Information-Centric Networking* (Osaka, Japan) *(ICN '22)*. Association for Computing Machinery, New York, NY, USA, 33–44. https://doi.org/10.1145/3517212.3558092
[4] Philipp Moll, Varun Patil, Lixia Zhang, and Davide Pesavento. 2021. Resilient Brokerless Publish-Subscribe over NDN. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 438–444. https://doi.org/10.1109/MILCOM52596.2021.9652885
[5] Named Data Networking (NDN) project. 2021. NDN Packet Format Specification version 0.3 – Signed Interest. https://named-data.net/doc/NDN-packet-spec/current/signed-interest.html accessed: 2021-05-20.
[6] Varun Patil, Seiji Otsu, and Lixia Zhang. 2023. Timers in State Vector Sync. In *Proceedings of the 10th ACM Conference on Information-Centric Networking* (Reykjavik, Iceland) *(ACM ICN '23)*. Association for Computing Machinery, New York, NY, USA, 115–117. https://doi.org/10.1145/3623565.3623754
[7] Varun Patil, Sichen Song, Guorui Xiao, and Lixia Zhang. 2022. Scaling state vector sync. In *Proceedings of the 9th ACM Conference on Information-Centric Networking* (Osaka, Japan) *(ICN '22)*. Association for Computing Machinery, New York, NY, USA, 168–170. https://doi.org/10.1145/3517212.3559485
[8] ProjectCCNx. 2012. CCNx Synchronization Protocol. CCNx 0.8.2 documentation. https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt
[9] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. 2018. *VectorSync: Distributed Dataset Synchronization over Named Data Networking - Named Data Networking (NDN)*. Technical Report. Named Data Networking. 9 pages. https://named-data.net/publications/techreports/ndn-0056-1-vectorsync/
[10] NDN Project team. 2021. Spec and API description of the StateVectorSync (SVS). NDN documentation. https://named-data.github.io/StateVectorSync/
[11] Jeff Thompson, Peter Gusev, and Jeff Burke. 2019. NDN-CNL: A Hierarchical Namespace API for Named Data Networking. In *Proceedings of the 6th*

ACM Conference on Information-Centric Networking (Macao, China) *(ICN '19)*. Association for Computing Machinery, New York, NY, USA, 30–36. https://doi.org/10.1145/3357150.3357400

[12] Lucas Wang, Alexander Afanasyev, Romain Kuntz, Rama Vuyyuru, Ryuji Wakikawa, and Lixia Zhang. 2012. Rapid Traffic Information Dissemination Using Named Data. In *Proceedings of the 1st ACM workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*. 7–12. https://doi.org/10.1145/2248361.2248365

[13] Xin Xu, Haitao Zhang, Tianxiang Li, and Lixia Zhang. 2018. Achieving resilient data availability in wireless sensor networks. (2018), 1–6. https://doi.org/10.1109/ICCW.2018.8403581

[14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014), 66–73.

[15] Zhenkai Zhu and A. Afanasyev. 2013. Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*. 1–10.