NDN Technical Report NDN-0073. http://named-data.net/techreports.html
Revision 1: May 2021

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.

1 INTRODUCTION

In Named Data Networking (NDN) [10], 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 *effectively* and *resiliently* in both infrastructure-based and infrastructure-free environments. In the latter case, network infrastructure is either non-exist, eg., ad hoc mobile, or otherwise disrupted, eg., 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 [5]) 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 dataset names, the *ChronoSync* [11] 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_1 cannot figure out the dataset difference with a received digest D, P_1 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.

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 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* [6]. VectorSync maintains two separate data structures. A *membership info object*

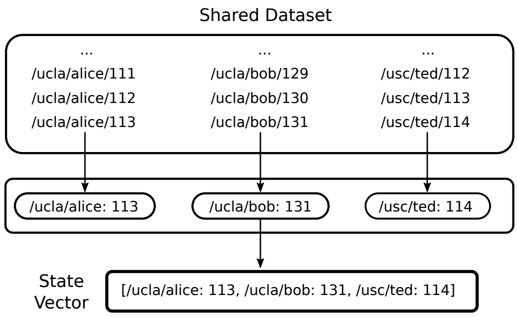
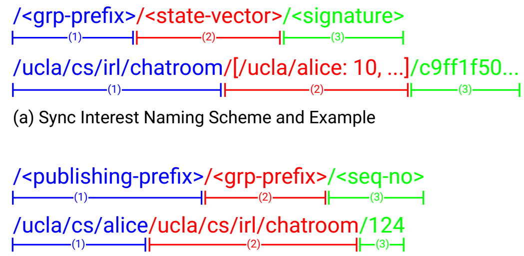


Figure 1: Relation between the Dataset State and the Representation as a State Vector



(a) Sync Interest Naming Scheme and Example

(b) Data Item Naming Scheme and Example

Figure 2: Naming for Sync Interest and Data Items

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) [9], 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 adhoc 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 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 message type which is referred to as *sync interest*, for dataset synchronization. Those 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 interest name. To prevent unauthorized parties from injecting incorrect state, sync interests are authenticated using interest signatures [3]. We illustrate the naming scheme for sync interests and an example name in Fig. 2a.

As indicated in Figure 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 Behavior

During operation, sync participants differentiate between two states: the steady state and the suppression state. In the *steady state*, a participant is not aware of any dataset change and listens to incoming sync interests. A sync interest timer is maintained to trigger sending periodic sync interests (first message in Fig. 3). These periodic sync interests act as a heartbeat mechanism and mitigate dataset changes communicated in preceding sync interests that suffered from packet loss. One can adjust the sync interest overhead in the network by adjusting the periodic sync interest timer (eg., 30 seconds with small random variation). As one of the next step, we are also looking into dynamically tune this interval based on the observed packet loss rate to balance the low overhead with low sync latency.

When receiving an incoming sync interest, the incoming sync interest can either carry i) up-to-date, ii) newer, or iii) outdated dataset state[1]. When receiving an up-to-date or newer state, the

[1] *Up-to-date* dataset state refers to the state vector of the incoming sync interest carrying the same sequence numbers for every state vector entry compared to the local records.

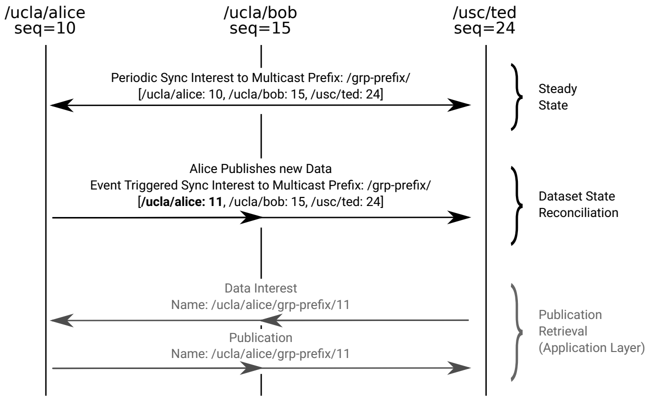


Figure 3: The SVS Protocol in Action

receiving participant assumes that all other sync participants received the same information, hence triggering a sync interest as a response is not meaningful. In this case, the periodic sync interest timer is reset without triggering a new sync interest.

When receiving an incoming sync interest carrying an outdated state, the *suppression state* is entered. In this case, the receiving participant has more up-to-date information than received and aims to reconcile the state difference by sending a sync interest carrying the participant's dataset state to the group. However, immediately responding to the outdated sync interest may lead to a high number of sync interests in the network (considering other participants also responding with sync interests). Therefore, a random suppression timer is initiated. While the timer elapses, incoming sync interests are monitored. If an incoming sync interest already reconciles the missing state, no action on timer expiry is set. Otherwise, timer expiry triggers a new sync interest carrying the participant's dataset state to the group, which mitigates the missing state. Thereafter, the participant migrates from suppression back to the steady state.

When a sync participant produces a new publication, its sequence number is increased and immediately emits an event-driven sync interest carrying the new dataset state (second message in Fig. 3). Thereby, other sync participants learn about the dataset change. Besides, the sending participant resets its periodic sync interest timer. We note that retrieving the actual data item (third and fourth message in Fig. 3) is out-of-scope of a sync protocol since application demands may vary (eg., reliability, or requirement to fetch every publication).

3.2 Why SVS Sync Interests Do Not Solicit Responses

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 as we describe below:

Outdated dataset state means that the sequence number of at least one state vector entry is lower. If the state vector of the incoming sync interest is not outdated and the sequence number of at least one state vector entry is higher compared to the local records, the sync interest carries *newer* dataset state.

1) All sync protocols use sync interests to carry the dataset state; they differ only in the encoding of the dataset state.
2) Participants in a sync group multicast sync interests to the group. Soliciting notifications of dataset state changes using multicast interests leads to three issues:
   i) 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.
   ii) A *multicast* interest solicits a reply from each of *multiple* potential producers. If multiple producers reply around the same time, due to NDN's one-interest-one-data principle only one of the replies is delivered to the interest sender.
   iii) As a consequence: different members in the group are likely to receive different updates, which leads to dataset state divergence, which will take up to multiple interest-data exchange cycles to converge.

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 removes all the above-identified issues.

4 ONGOING EFFORTS TO IMPROVE SVS

In this section, we identify a few additional issues related to the SVS design.

SVS Scalability. Looking at the design of SVS might raise scalability concerns, because the state vector design carries the entire dataset state in the sync interest name. A big number of sync participants leads to a big state vector size, and interests have a strict upper size limit by network MTU (maximum transmission unit). Efficient state vector encoding and compression schemes may help alleviate this concern to certain degree only. The most promising direction is to utilize SVS's property of each [producer, sequence number] pair is independent from other pairs, therefore each sync interest is not required to carry the full dataset state. As part of our ongoing work, we are evaluating approaches that let sync interests carry partial state vectors.

SVS Data Naming. With sequential data item naming, data item names no longer carry the *complete* application semantic information; instead a sequential name carries the producer's name, and replaced the lower part of the name by a sequence number. Doing so enables SVS to scale well with large number of data items with a compact dataset state representation. As next step, we plan to enable SVS to support pub/sub APIs with general application layer data names, by providing a mapping between each app data name and the sequence number assigned by SVS, so that when a participant fetches a data item using its sequence number, the producer can reply with the original data packet produced by the application (by encapsulating it in the content of an outer packet with the sequence number name).

This solution takes after the solution described in Nichols [4], where the reply to a sync interest contains NDN Data packet(s), with the original semantic name as produced by the application

This proposed approach should enable synchronizing arbitrary application names using SVS by requiring SVS maintain a mapping table between sequence numbers and original names.

SVS Group Membership Management. Also, 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 in upper layers only. Some higher-level libraries [8] 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 transport in such libraries.

5 WRAPPING UP

This technical report introduces the function of SVS yet not providing performance comparisons or a broad discussion of design decisions. Preliminary evaluations (not part of the report) showed a good performance of SVS on networks with no or minor packet loss. Further, SVS improves on traffic and computational overhead compared to the DDSN implementation.

With this report, we expect to provide information to render existing SVS libraries useful for NDN experimentation. We highlight the availability of the online specification of SVS [7]. Furthermore, the aforementioned reference features open-source SVS libraries in different programming languages and refers to demo applications showcasing the use of SVS.

We plan to update this report according to the SVS protocol updates over time.

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] 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.

[4] Kathleen Nichols. 2019. Lessons Learned Building a Secure Network Measurement Framework Using Basic NDN. In *Proceedings of the 6th ACM Conference on Information-Centric Networking (ICN '19)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3357150.3357397

[5] ProjectCCNx. 2012. CCNx Synchronization Protocol. CCNx 0.8.2 documentation. https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt

[6] 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/

[7] NDN Project team. 2021. Spec and API description of the StateVectorSync (SVS). NDN documentation. https://named-data.github.io/StateVectorSync/

[8] 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

[9] 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

[10] 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.

[11] 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.