Supporting Pub/Sub over NDN Sync

Varun Patil
varunpatil@cs.ucla.edu
UCLA, Computer Science

Philipp Moll
phmoll@cs.ucla.edu
UCLA, Computer Science

Lixia Zhang
lixia@cs.ucla.edu
UCLA, Computer Science

ABSTRACT

Distributed dataset synchronization, or Sync, plays the role of a transport layer protocol in Named Data Networking (NDN). The role of Sync is to synchronize the namespace of all data productions by multiple entities running the same application. NDN application developers desire a high level API, such as the commonly used pub/sub API that hides transport and network layer details. This poster explores the design of such an API built on top of State Vector Sync (SVS), one of the NDN Sync protocols, along with a low-latency data fetching option. With this API, SVS provides fast and resilient dataset synchronization, enabling developers to work with a familiar pub/sub API while benefiting from NDN's capabilities of built-in data security, multicast data delivery, in-network caching, and consumer driven flow control.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; *Transport protocols.*

KEYWORDS

Named Data Networking, NDN Sync, Publish-Subscribe

ACM Reference Format:
Varun Patil, Philipp Moll, and Lixia Zhang. 2021. Supporting Pub/Sub over NDN Sync. In *8th ACM Conference on Information-Centric Networking (ICN '21), September 22–24, 2021, Paris, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3460417.3483376

1 OVERVIEW

Today's distributed applications running over TCP/IP are not directly built on top of the TCP transport. Instead, most applications utilize higher-level communication frameworks that offer more developer-friendly transport semantics. Examples of such higher-level frameworks are publish-subscribe (pub/sub) communication over MQTT [5] or high-throughput message queues, such as ZeroMQ [1]; both utilize TCP for packet transport. Similar to TCP, Sync, the transport service in NDN also does not directly offer high-level communication primitives that developers desire.

In this poster, we present a design for supporting such a higher-level API desired by developers. We identify the gap between the requirement and what Sync provides in §2.1. To bridge this gap, we sketch the design of a pub/sub API in §2.2, and a low-latency data fetching option in §2.3, built over State Vector Sync (SVS) [4], one of the NDN Sync protocols. This way, we provide developers with a familiar interface for data communication over NDN, while hiding network and transport layer details beneath the pub/sub library. Our open source implementation of the API is available at https://github.com/named-data/ndn-svs.

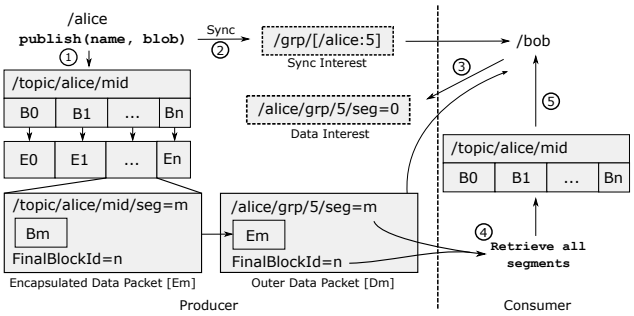

Figure 1: Data distribution over SVS

2 DESIGN

SVS uses sequence numbers to represent each participant P's data production state. Whenever P produces a new piece of data, it increments its sequence number by one. SVS keeps all the communicating parties in an application (the "Sync group") up-to-date about new data production, by propagating P's latest sequence number to the group. Developers, however, require higher-level APIs for multiparty communications, as elaborated below.

2.1 Data Distribution

One of the simplest APIs required by a distributed application is the ability to distribute data to all the other participants. We organize all the participants in a Sync group, and provide a `publish` API, which accepts a binary blob with an NDN name, and makes it available to the entire Sync group. The name provided with the blob should reflect the content of the blob semantically (we further discuss semantic naming in §2.2).

The process of data distribution over SVS is illustrated in Fig. 1, and follows the following steps:

(1) When the application calls the `publish` method, a new SVS sequence number is assigned for the blob to be published. If the application's blob is larger than the network MTU size, the blob will be segmented, with each segment fitting into an NDN Data packet, as we describe below, whose size is within the network MTU limit. An NDN Data packet E_m is created for each such blob segment B_m, named under NDN's segmented data naming conventions [7]. Each E_m is then encapsulated inside another data packet D_m, named using the name identifying the producer, the new SVS sequence number, and the segment number of the blob segment, as shown in Fig. 1. Both the inner packet E_m and outer packet D_m are signed with appropriate signing keys as defined by the application. The outer packets D_m are then published.

ICN '21, September 22–24, 2021, Paris, France

ACM ISBN 978-1-4503-8460-5/21/09.
https://doi.org/10.1145/3460417.3483376

(2) After segmenting and encapsulating the data, the publisher's sequence number is incremented. SVS propagates this change to all other participants.

(3) A consumer that receives the new sequence number constructs the name of the first outer Data packet D_0 to fetch it.

(4) The `FinalBlockId` of the segmented data blob is carried in the first data packet, following the NDN packet encoding convention, informing the consumer to fetch all segments up to D_n.

(5) Once all segments are received, the consumer reconstructs the producer's blob by concatenating the content of the encapsulated application data packets, which is then passed to the application.

This process is executed by every Sync group participant, thereby getting the published data blob to all. Note that the approach of using Sync data packets to encapsulate app data is adopted from DCT [6]. Although similar APIs have been created in the past, in particular by DCT and PSync [8], our design differs in that it delivers not only small app data blobs, but also arbitrarily large ones, without needing the Sync client to keep track of blob segmentation.

We also note that signing each data packet twice ensures (along with the naming scheme) that both the encapsulated and outer data packets are semantically complete and secured. This allows for executing different security policies for the transport protocol and the encapsulated application data[1].

However, the above mechanism has two limitations. First, a consumer may wish to see the actual application data name to decide whether it wants to fetch the blob or not, however it cannot see the inner data name before the using sequence number to retrieve the outer packet. Second, retrieving a newly produced data takes 1.5 round trip times (0.5 for learning the new sequence number from Sync Interest, and another RTT to retrieve data). Our pub/sub API and low-latency enhancement address these issues.

2.2 The Pub/Sub API

To allow participants selectively retrieving only the data they are interested in, we provide a `subscribe` API, which accepts a name prefix, and selectively fetches the data objects that match the prefix.

SVS notifies new data productions of a producer with sequence numbers, without providing the application data names that correspond to these sequence numbers. We address this issue by providing a mapping between the two. To get this mapping, the consumer sends a mapping query Interest as shown in Fig. 3a. This query includes the producer's identifier and the sequence number range for which the mapping is requested. The pub/sub library at the consumer decides the requested range of sequence numbers by identifying the published sequence numbers of each publisher for which the mapping information is missing. The publisher replies with the corresponding application data names. Based on these names, the consumer can now selectively retrieve the data objects it subscribes to, using the mechanism described in §2.1.

To further optimize the mapping retrieval, when a publisher sends out a Sync interest to notify others after producing a new data blob, we can use this Sync Interest to piggyback the mapping

[1]Note that the overhead for signing the packet twice depends on the used signature algorithm, and different algorithms may be used for both packets. (e.g. asymmetric signature for application data; symmetric algorithms for outer data, e.g. AEAD [3])

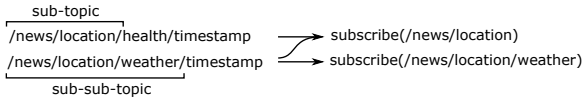

Figure 2: Semantic naming of application data

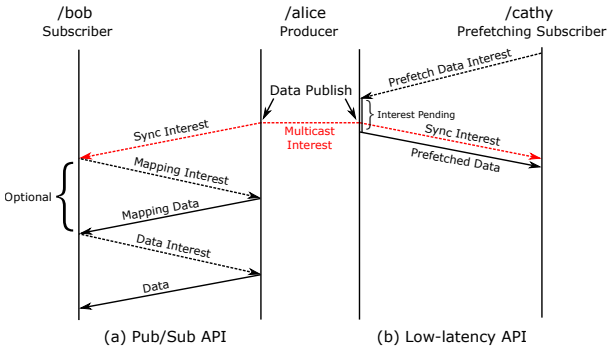

Figure 3: Sequence of protocol messages

between the new sequence number and the newly produced data name, by carrying the data name in the `ApplicationParameters` field, as illustrated in Fig. 3a, which marks the mapping retrieval optional. A consumer receiving this Sync Interest receives the mapping, without having to retrieve the mapping separately.

While the pub/sub API allows fetching objects matching certain prefixes, semantic naming by applications for the published data objects is critical for best results. Hierarchically structured names allow fetching narrow subsets of data, allowing consumers to fetch only the data they require. An example of a semantic naming scheme for a news application and how pub/sub can be used to selectively retrieve its subsets is illustrated in Fig. 2.

2.3 Low-Latency Data Fetching

To support applications with low-latency requirements, we adopt a data prefetching approach introduced by PLI-Sync [2]. We provide a `subscribeToProducer` method, accepting a producer name prefix and a callback. This call assumes that a consumer wants to receive all data matching a producer prefix with low latency, and the consumer would filter the received data objects if it only requires a subset of the publications. On receiving this call, the pub/sub library prefetches the next Data packet by sending an Interest with the next sequence number, before knowing the data is produced, as shown in Fig. 3b. The Interest lifetime of the prefetching Interest can be determined by the application based on the data production pattern, with a default of 4sec in our implementation. This prefetching Interest will be pending at the producer until it publishes the packet, and may need to be refreshed before its Interest lifetime expires. Data prefetching is made possible by SVS's usage of sequence numbers, enabling the consumer to construct the interest for next outer data packet before it is generated. Depending on the frequency of data publication, the consumer could also prefetch multiple future sequence numbers, forming a prefetching pipeline.

ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under award 1719403, and by Operant Networks.

REFERENCES

[1] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications.* O'Reilly Media, Inc.

[2] Yi Hu, Constantin Serban, Lang Wang, Alex Afanasyev, and Lixia Zhang. 2021. PLI-Sync: Prefetch Loss-Insensitive Sync for NDN Group Streaming. In *IEEE International Conference on Communications (ICC).* IEEE.

[3] D. McGrew. 2008. *An Interface and Algorithms for Authenticated Encryption.* RFC 5116. RFC Editor. https://www.rfc-editor.org/info/rfc5116

[4] Philipp Moll, Varun Patil, Nishant Sabharwal, and Lixia Zhang. 2021. *A Brief Introduction to State Vector Sync.* Technical Report NDN-0073, Revision 2. Named Data Networking. 1–4 pages.

[5] mqtt.org. 2020. MQTT: The Standard for IoT Messaging. https://mqtt.org/ accessed: 2021-07-19.

[6] Kathleen Nichols. 2021. Trust Schemas and ICN: Key to Secure IoT. In *Proceedings of the 8th ACM Conference on Information-Centric Networking.* ACM.

[7] NDN Project Team. 2019. *NDN Technical Memo: Naming Conventions.* Technical Report NDN-0022, Revision 2. Named Data Networking. 2–3 pages.

[8] Minsheng Zhang, Vince Lehman, and Lan Wang. 2017. Scalable name-based data synchronization for named data networking. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications.* 1–9. https://doi.org/10.1109/INFOCOM.2017.8057193