

The Design of RoundSync Protocol

Pedro de-las-Heras-Quirós, Eva M. Castro, Wentao Shang, Yingdi Yu, Spyridon Mastorakis,
Alexander Afanasyev, Lixia Zhang

{pedro.delasheras,eva.castro}@urjc.es, {wentao,yingdi,mastorakis,aa,lixia}@cs.ucla.edu

Abstract—Distributed dataset synchronization (Sync in short) implemented by ChronoSync allows a group of nodes to operate on a shared dataset with eventual consistency. However, when multiple nodes in the same sync group publish new data simultaneously, ChronoSync needs to either use exclude mechanism to fetch the simultaneously produced data, or fall back to a recovery mechanism. This problem is caused by a semantic overloading on Sync Interests: a Sync Interest is used both to detect state inconsistency (by embedding the dataset state digest in the Interest name) and to retrieve update (resulting in the update being named under a specific digest). In this report, we first use a simple case study to analyze the behavior of ChronoSync under simultaneous data publications, and then introduce RoundSync, a revision to ChronoSync to fix the overloading problem. RoundSync splits data publications into “rounds” and uses two separate Interest types for state inconsistency detection and update retrieval. We have implemented the RoundSync protocol, conducted preliminary evaluation through simulations, as well as performed comparative study of the RoundSync design with other NDN dataset synchronization solutions that have been developed so far [1].

I. INTRODUCTION

Distributed dataset synchronization (or sync in short) is a communication abstraction in Named-Data Networking (NDN) [2] that enables a group of data producers to contribute to a shared dataset. Due to the unique binding between NDN names and immutable data objects, the synchronization of a shared dataset can typically be achieved by synchronizing the corresponding namespace that contains the names of all the data objects in the dataset.

ChronoSync [3] is one of the earliest sync protocols developed for the NDN architecture. It implements a multi-producer replication service with eventual consistency: each ChronoSync node in a sync group independently produces new data objects and add them to the shared dataset and notifies the others; all the updates are replicated asynchronously to all the other nodes in the group, eventually resulting in a consistent knowledge about the dataset across the sync group. However, when multiple nodes produce data around the same time (e.g., when a sync group is large in size and/or new data are produced at high rate), ChronoSync may suffer from extended delays in reaching eventual synchronization (Section II). As we explain in more detail in Sections III and IV, the main cause of this problem is due to bundling semantics of inconsistency detection and new data retrieval on the Sync Interest.

In this paper we present the design of RoundSync (Section V), a redesign of ChronoSync, that disentangles semantics into separate processes to enable fast synchronization in

face of simultaneous data productions. RoundSync splits data productions into “rounds” and uses two distinct Interest types: Sync Interest to advertise the current round number and its state, and Data Interests to retrieve updates published in each round.

II. THE ORIGINAL CHRONOSYNC DESIGN

To simplify the design, ChronoSync adopts a simple naming convention: it names each data object in the shared dataset by appending a monotonically increasing *sequence number* to the unique data prefix of each producer node. The sequence number is incremented by one for each new data object published by the same node. With this naming convention, the complete knowledge of a sync node’s data publication can be concisely represented by the node’s data prefix and the largest sequence number it has produced. The application running on top of ChronoSync may name the data in any way as needed, and the actual application data name can be encapsulated in the data packet generated by the ChronoSync layer.

ChronoSync decouples the synchronization of the namespace of a sync group, which contains the names of all the produced data objects in the dataset, from fetching the produced data objects. As we describe below, each node sends a Sync Interest periodically to solicit new update. If a node $N2$ produces a new data object, $N2$ replies to the Sync Interest with a sync reply packet. Upon receiving the sync reply, the application on each node can make an independent decision about whether, or when, to fetch the data that have been produced. If the size of the new data is small, ChronoSync may also encapsulate the whole data packet directly in the sync reply.

Each node running ChronoSync maintains an internal *sync state* about the replicated namespace in a data structure called the *sync tree*, which is a two-level tree. Each leaf node in the sync tree represents a sync node in the group, and stores the data prefix and the latest sequence number of the corresponding node. The knowledge of the entire dataset is summarized by a root digest that covers the information stored in all the leaf nodes in the sync tree.

To synchronize the namespace of the dataset, each sync node sends out *Sync Interests* periodically to other nodes in the group via multicast. A Sync Interest’s name carries the root digest computed by the sending node. When all the nodes in a group are synchronized, i.e., they all have the same knowledge about the shared dataset, they generate identical Sync Interests which are aggregated at routers, resulting in a two-way many-to-many shared tree across the network, with each node (more

precisely, the sync application in the node) facing a waiting Sync Interest.

When a node produces a new data object, it replies to the waiting Sync Interest with a data packet that contains the name of the new data. This data packet satisfies the waiting Sync Interest and is forwarded to the other nodes in the same sync group by following the “multicast tree” set up by their Sync Interests. This event-driven data propagation behavior is similar to that implemented by a “push-based” communication protocol. The node also sends a new Sync Interest which carries the new digest which is computed after incorporating the newly produced data.

Since a Sync Interest carrying the name N_d solicits updates from any other nodes in the same group, when multiple nodes generate new data simultaneously, they will reply to the same Sync Interest simultaneously, each sending a data packet carrying its own updates to the shared dataset. The names of all the responding data packets share the same prefix N_d which is followed by the node specific name components. However because one Interest packet can retrieve only one Data packet, only one of the sync replies is returned to each node which has an outstanding Sync Interest; different nodes may receive different replies, resulting in diverged states of the sync group.

To retrieve other data packets that may have been produced at the same time, the current ChronoSync operates in the following way.

- 1) After retrieving a data object with name N_d , a node reissues the Sync Interest again using the same name N_d , the Interest will carry an *exclude filter* to exclude the data packet(s) that have already been received.
- 2) This Sync Interest with the exclude filter will be forwarded to all nodes in the sync group.
- 3) If there is no more Sync Data published under name N_d , this Interest will expire without answer.
- 4) Otherwise one more data packet with prefix N_d is retrieved, and the process goes back to step-1 again.

This solution of using exclude filters has two known drawbacks. First, one does not know whether there is more data to fetch, the last Sync Interest in the exhaustive search is a waste. Second, the size of the exclude filter is limited, therefore this solution is effective only if the number of simultaneous data productions is below a threshold.¹ Furthermore, since either an Interest or a Data packet can be lost, the solution does not guarantee the retrieval of all the data produced under name N_d .

In addition to the above, ChronoSync provides several mechanisms to recover from the sync state divergence. First, each sync node maintains a (limited size) *digest log* to facilitate quick (re)synchronization. The *digest log* contains the historical digest values it has observed, together with the updates generated on top of the sync states identified by those digests. When a node receives a Sync Interest which carries a

digest value different from the current local digest, the node searches the digest log. If the digest is found in the log, the node returns a sync reply Data packet that contains all the updates from the log entry up to the latest state to help speed up the synchronization process.

There can also be cases where a node receives a Sync Interest carrying a digest value that it cannot locate in its digest log. This could be caused by out-of-order packet delivery of the network (i.e., when a new digest arrives before the update that generated the new digest). ChronoSync thus introduces a random delay before taking any repair actions. If the unrecognized digest cannot be resolved after a short delay, the cause could be due to packet loss or multiple simultaneous updates (see Section IV for more detail). In this case, ChronoSync falls back to a simple recovery procedure which works in the following way:

- The node who receives an unknown digest D sends a special *Recovery Interest* to the group.
- Whoever sent the Sync Interest with digest D (there could be multiple of them) can reply with its entire sync tree, which can then be merged by the recipient with its own sync tree.

This recovery procedure is considered expensive in a large group, because it involves all the nodes in the group to take actions (see more elaboration in the next section). It is also possible to address the issue with other more efficient reconciliation algorithms [4], [5], [6], but their uses have not been defined or implemented in the existing experimental prototypes.

III. CASE STUDY: STATE DIVERGENCE AND RECOVERY IN CHRONOSYNC

State divergence in ChronoSync can happen because of simultaneous data productions and/or packet losses. To resolve the differences among diverged states, individual nodes need to retrieve the missing updates from each other. One can use the exclude filter to retrieve multiple sync replies generated on top of a previously synchronized state. However, when nodes get into diverged states, for example:

- one node $N1$ in the state with digest D_1 produces a sync reply O_{D1} ,
- another node $N2$ has the state with digest D_2

Node $N2$ has no way to learn the digest value of D_1 . Consequently, $N2$ has no way of retrieving the sync reply O_{D1} . As a result, the sync nodes are forced to use the recovery mechanism to bring the group in sync again.

In the following we illustrate the above analysis using a concrete example with 3 nodes A, B, and C in a sync group. Fig. 1a shows the initial state where all the three nodes send Sync Interests with the same digest $d0$. In Fig. 1b, A and B simultaneously publish a new piece of data $A1$ and $B1$, respectively, and update their local state digests to $dA1$ and $dB1$. Node C receives only the sync reply from A, and updates its own digest to $dA1$. Note that A and B will not receive the sync reply produced by each other at

¹One could mitigate this problem by a more efficient encoding of the exclude filter, e.g. using Bloom Filter. We decide not to investigate further down that path. The current plan for the NDN protocol evolution is to remove the dependency on all Interest selectors.

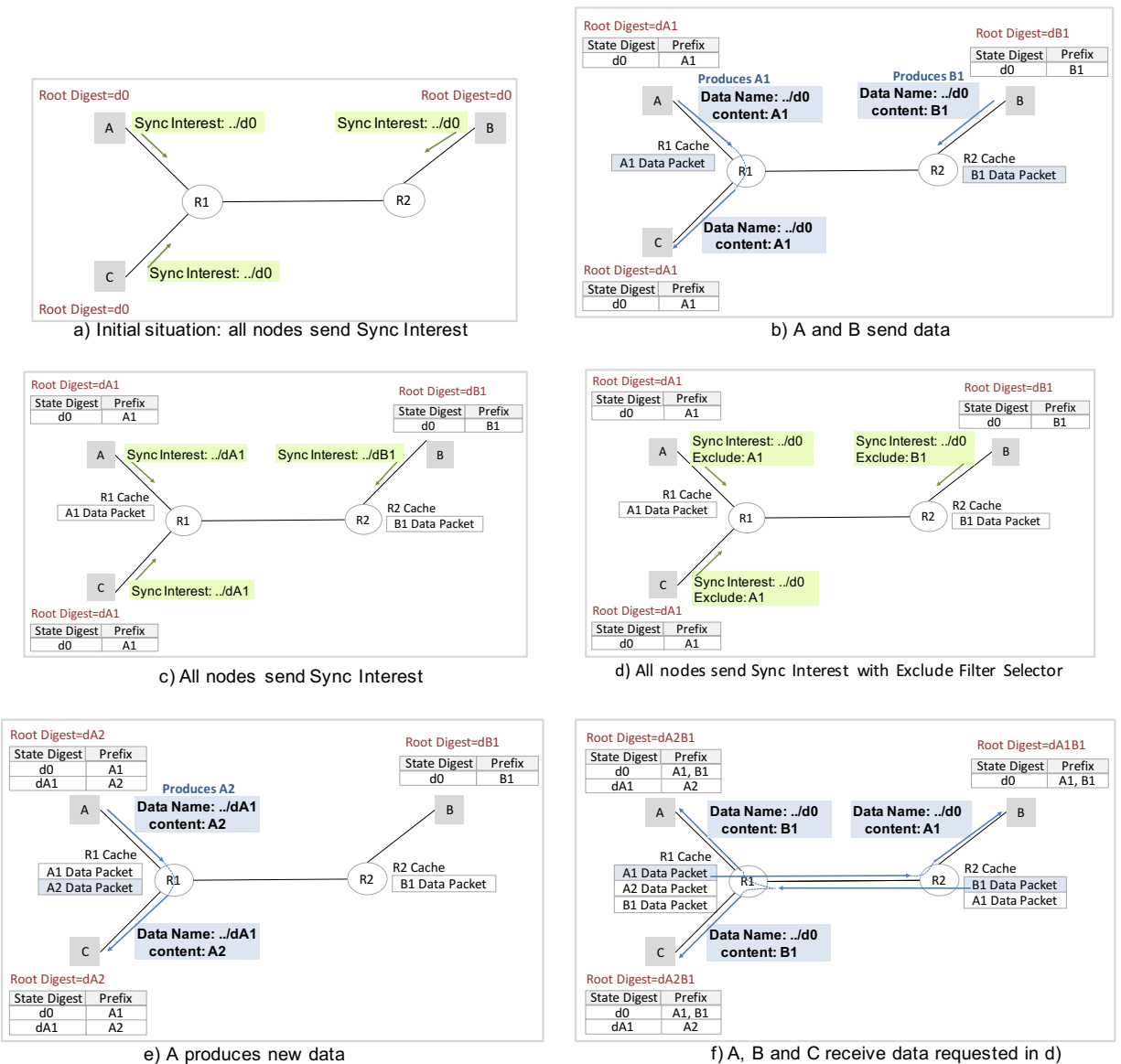


Fig. 1. Example of simultaneous updates.

this time, because their pending Sync Interests with digest d_0 has already been satisfied by their own new data production A_1 and B_1 , respectively. Due to the “one-Interest-one-Data” rule in NDN, the pending Sync Interest serves as “one-packet subscription” to any data produced under the name of that Interest.

After receiving a sync reply:

- 1) each of the three nodes sends a new Sync Interest with its updated digest, as shown in Fig. 1c. As a result, A and C receive an unknown digest dB_1 and B receives an unknown digest dA_1 . Upon receiving an unknown digest, all three nodes start a random timer to wait for potential sync data packet that may resolve the conflict. Note that in this case no sync data packet will be received.

- 2) the three nodes resend their previous Sync Interests with additional exclude filter containing the implicit digest of the sync reply packets they have received (Fig. 1d), hoping to retrieve the missing replies that caused the divergence.

However, before they are able to reconcile the states, node A produces a new data packet, A_2 , and generates a sync reply to the Sync Interest with digest dA_1 (Fig. 1e). Note that the Sync Interests sent by nodes A and C with the same digest dA_1 are still pending in the network. Therefore, the new sync reply will be returned to C, resulting in both A and C updating their state digests to a new value dA_2 . Unfortunately, node B will not be able to calculate the digest dA_2 which was generated on top of dA_1 : the data object leading to dA_1 has not been received by B. After B receives the missing data A_1 from R1

cache using the Sync Interest with exclude filter, B will update its state digest to $dA1B1$, which is unrecognizable by either A or C. Similarly, after receiving the missing data B1 from R2 cache, A and C will update their state digests to $dA2B1$ which is unrecognized by node B (Fig. 1f). At this point, three nodes will execute the recovery process to sync up with each other again.²

The recovery mechanism of ChronoSync can be considered expensive for two reasons. First, every node that receives a Recovery Interest needs to inspect its digest log to find the digest carried in that Interest. If the digest is found in the log, the node produces a sync reply containing all updates since the digest was generated in the group. This searching and responding process takes time and may affect the application performance. For example, a node may need to lock the digest log data structure during the recovery, which stops the protocol from processing new sync replies; one may also stop producing new data to avoid making other nodes further out-of-sync. Second, the reply to a Recovery Interest contains the data prefix and the latest sequence number of every node that has published new data since the corresponding digest was generated. Therefore the size of the reply packet can be rather large in a large sync group with high data publishing rate per node.

Given the cost of ChronoSync’s recovery mechanism, its invocation must be minimized. To that end, we first identify the causes that lead to the invocation of the recovery mechanism, and then develop the RoundSync as a means to mitigate the identified issues.

IV. IDENTIFYING THE ROOT CAUSE IN CHRONOSYNC STATE DIVERGENCE

We identify the root cause to the inevitable invocation to ChronoSync’s recovery process, as described in the previous section, to be the fact that ChronoSync uses a Sync Interest to serve two different purposes: (1) it lets each node to retrieve updates as soon as they are produced by any other nodes, and (2) it lets each node detect whether its knowledge about the shared dataset conflicts with anyone else in the sync group. Overloading these two functions on the same Sync Interest affects both data retrieval and publishing.

First, a sync reply packet (data production) is named under the digest D of the state on top of which the update is applied. However, a node does not know how many sync replies get generated under D . After receiving a sync reply under D , the node can use exclude filter persistently to look for next reply, but this can be either unnecessary (when data production rate is low, only a single reply under each D) or ineffective—when either a Sync Interest from node N or a reply to be delivered to N is lost, causing N to be out of sync, therefore N has to fall back to the recovery mechanism.

²In this specific case, it seems the recovery process might be avoidable, if A holds back new data production A2 until the unknown digest $dB1$ gets resolved. However, the Sync Interest carrying $dB1$ could be lost, hence A and C may not be aware of the state divergence.

Second, if a sync node publishes new data before it is synchronized with all the other nodes in the group, i.e., reaching an agreement on the latest digest of the shared dataset, then the digest in the name of the sync reply may not be known by other nodes, making it impossible for other nodes to retrieve the reply, eventually resulting in a sync recovery. Therefore, a node should hold back from publishing new data when inconsistency is detected, but this will halt the applications running over ChronoSync until the conflicts are resolved.

V. ROUNDSYNC DESIGN

In this section we describe RoundSync, a revision to the original ChronoSync protocol to address the issues identified in the previous section. Like ChronoSync, RoundSync provides the synchronization support with eventual consistency among a group of distributed nodes sharing a common dataset. Different from ChronoSync, RoundSync uses the following two means to mitigate the identified issues in the previous section.


First, RoundSync divides the data publishing and synchronization process into *rounds* (hence the name of the protocol). Each round is identified by a monotonically increasing *round number*. Each node may publish *at most one* piece of data in each round. Data synchronization in each round is independent and does not affect data publishing in other rounds. For example, a node can publish new data in round k while trying to sync up with other nodes in round $(k - 1)$ or even earlier rounds.

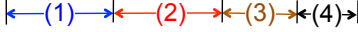
Second, RoundSync decouples the two functions overloaded in the ChronoSync’s Sync Interest: detecting inconsistency in the sync group, and retrieving updates on the sync state. RoundSync defines separate types of Interests. The *Data Interest* is used for fetching updates generated by any node in a sync group. The *Sync Interest* is now used solely for detecting inconsistent states within a round. Note that the semantics of the Sync Interest in RoundSync is different from that in ChronoSync.


RoundSync enables each node to synchronize with others on a per round basis. However, the state of each round can change, say when a disconnected node gets reconnected again. To ensure the overall dataset consistency without having to repeatedly checking every round, RoundSync introduces a new term *cumulative digest*, which is semantically equivalent to the state digest in the original ChronoSync design, and develops a new recovery mechanism to allow the node to reconcile the different states quickly and efficiently (see Section V-F).

A. Naming

1) *Data Interest*: Each node sends a Data Interest to fetch updates published by others in each round. The reply to a Data Interest contains the name of the data produced by the application running on top of RoundSync. RoundSync follows the same naming convention as used in ChronoSync, i.e., a piece of application data is named by the concatenation of the

`/ucla/alice/appPrefix/108`

 (a) An example of application Data name

`/multicast/appPrefix/DATA/2006`

 (b) An example of Data Interest name

`/multicast/appPrefix/SYNC/2005/d0509a...`

 (c) An example of Sync Interest name

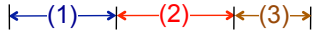
`/ucla/alice/appPrefix/RECO`

 (d) An example of Recovery Interest name

Fig. 2. Naming conventions in RoundSync.

application prefix with the sequence number of that data, as shown in Fig. 2a.

Fig. 2b shows the structure of a Data Interest name: the name starts with a multicast prefix that is announced by every node in the sync group,³ followed by the component that identifies the application for demultiplexing purpose, and a special marker component, “DATA”, that indicates the type of the Interest; the last component in the Data Interest name carries the round number, e.g. Fig. 2b shows a Data Interest requesting data produced in round 2006. Note that the Data Interest name contains *no* digest, because the detection of state inconsistency is now done separately by the Sync Interest. This separation enables a node to construct a Data Interest name by following the well-defined naming convention to retrieve any missing updates in a given round. To avoid retrieving the same data back from router caches, a Data Interest may also include an exclude filter. The data publishing and fetching process is described in Section V-C.

2) *Sync Interest*: All the nodes in a sync group exchange Sync Interests to detect inconsistent sync state in each round. Different from the original ChronoSync design, a Sync Interest in RoundSync only informs others of the sender’s state digest, but does not trigger any reply packet. Fig. 2c shows the components in a Sync Interest name. Similar to a Data Interest, the name starts with a multicast prefix, followed by the application identifier and a special marker component, “SYNC”, which distinguishes Sync Interests from Data Interests. The last two components in the Sync Interest name carry the round number and the *round digest* which covers the names of all the application data received by the sender in that round.

The round digest is calculated as follows: for each application data name received in round N , with node prefix p_j and

³Such per group multicast may raise concerns about routing scalability. We are currently exploring effective solutions to address this concern.

Application Dataset	
p_j (node prefix)	s_j (latest seq#)
“/ucla/cs/alice”	20
“/ucla/cs/bob”	1
“/ucla/remap/daniel”	1
“/ucla/remap/karen”	1

TABLE I
AN EXAMPLE OF APPLICATION DATASET.

sequence number s_j , a digest d_N^j is calculated:

$$d_N^j = H(p_j | s_j) \quad (1)$$

where H is a pre-configured hash function used by all sync nodes. The round digest rd for round N is then calculated by applying a hash function to the concatenation of the digests d_N^j computed from each data name (ordered lexicographically based on the actual names $p_j | s_j$):

$$rd_N = H(d_N^1 | d_N^2 | \dots | d_N^K) \quad (2)$$

where K is the total number of data published so far in round N .

A node sends out a Sync Interest for a round whenever its corresponding round digest changes due to new update(s), independent from whether the update(s) generated by the node itself or others. Upon receiving a Sync Interest, a node compares the round digest with its local digest for the same round. If the digests are different, the node will issue Data Interests for that round to retrieve the missing updates. The detection and reconciliation of inconsistency in a single round is described in more detail in Section V-D.

3) *Recovery Interest*: When a sync node detects inconsistency in the cumulative digest which is carried in all data packets (see Section V-C), it may send a unicast *Recovery Interest* to a specific node to retrieve all updates generated since the inconsistent round. The name of the Recovery Interest contains the unicast prefix of the target node, the application identifier, and the special marker component, “RECO”, as shown in Fig. 2d. The detail of the recovery mechanism is described in section V-F.

B. Protocol State

Similar to ChronoSync, each node in RoundSync maintains three data structures as part of its internal state:

- The *Application Dataset* contains the namespace of the shared dataset that RoundSync replicates. It is equivalent to the sync tree data structure in ChronoSync but without the root digests (see Figure-4 in [3]). Table I shows an example.
- The *Round Log*, indexed by round numbers, stores the digest of each round together with the names of the application data produced in that round. It is similar to the digest log in ChronoSync, except that each digest value covers only the data published in the corresponding round, rather than the entire application dataset. Table II shows an example.

Rounds Log		
Round#	rd _N	Application data names
1	1a...a	[["/ucla/cs/alice", 1]]
2	b2...b	[["/ucla/cs/bob", 1]]
3	4c...c	[["/ucla/remap/daniel", 1], ["/ucla/remap/karen", 1]]
...
32	12...f	[["/ucla/cs/alice", 20]]

TABLE II
AN EXAMPLE OF ROUNDS LOG.

- The *Recovery Data Names Collection* is used for processing recoveries from long-term inconsistencies, as described in section V-F.

Whenever a sync node publishes a piece of new data, or receives new data from other nodes in round N , it makes the following updates to the protocol state:

- it updates the latest sequence number of the producer node in the Application Dataset to the new value;
- it adds the data name into the lexicographically ordered set of application data names in the corresponding entry for round N in the Rounds Log, and
- it recalculates the round digest rd_N .

C. Data Publishing

When the application running on top of RoundSync publishes new data, it passes the data name to the RoundSync service module. RoundSync then updates the local sync state (described in the previous subsection) and generates a data packet to reply to the pending Data Interest in the current round n , with the name of the application data encapsulated in the content. The producer node then *immediately* moves to the next round by increasing the round number to $n + 1$, and sends out a Data Interest for the new round.

When other nodes in the same sync group receive the reply in round n , they update their sync states based on the encapsulated name and then move to the next round by sending out a Data Interest with round number $n + 1$.

When a node updates the digest in round n , either because it publishes new data itself or because it receives a new update from others in the group in that round, it sends out a Sync Interest for round n immediately carrying the updated round digest, to inform others about the new change. Different from ChronoSync, after receiving a reply to a Data Interest in round n , a node does not need to retransmit the Data Interests in round n with an updated exclude filter containing the implicit digest of the replies the node has received in round n so far. Instead, it only does so after state inconsistency is detected from received Sync Interests, as we describe in the next subsection.

Fig. 3 illustrates the data publishing process with a simple example where node A produces a data packet in round 1 which is propagated to node B and prompts both A and B to enter round 2.

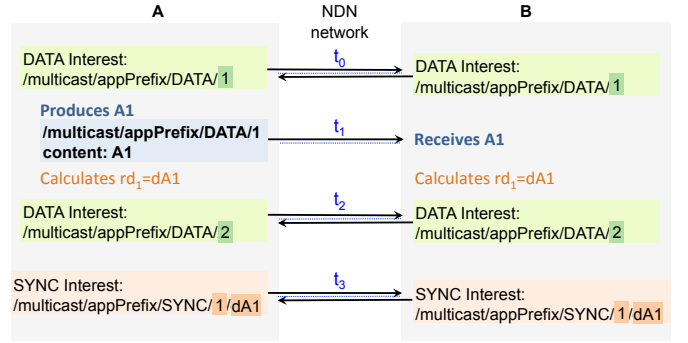


Fig. 3. Publishing and synchronizing data in round 1.

D. Per Round Inconsistency Detection

RoundSync detects inconsistent states in each round separately. When a sync node N , who enters round $n + 1$ and sends Sync Interest for round n , receives a Sync Interest for round m , it compares m with n :

- If $n = m$, the node compares the received round digest with its local digest for round n . If the digest values are different, the receiving node will issue a Data Interest for round n to retrieve potentially missing updates, carrying an exclude filter to exclude any data pieces it has received in round n .
- If $n < m$, the node knows that it may be missing updates in rounds $n + 1$ up to m . Therefore the node sends out Data Interests immediately for each of those rounds, and moves itself to round $m + 1$. While the node is still synchronizing for the earlier rounds, it may publish new data in round $m + 1$. That is, the synchronization process in earlier round does not block data publishing in later round.
- If $n \geq m$, N has already moved past round m . Therefore it compares the received round digest in the Sync Interest with the local digest for round m . If they are different, N determines that there is inconsistency in round m , and then send a Data Interest for that round with an exclude filter containing the implicit digest of all the data pieces produced in that round which the node has already received, to resolve the inconsistency by retrieving any missing updates in round m .⁴ We would like to highlight the fact that node N can send a Data Interest to retrieve any potential missing data as soon as inconsistency is detected, without knowing the state digest of the node who produced the missing update. This is because the Data Interest name is solely determined by the round number, a benefit from decoupling update retrieval from inconsistency detection.
- If $n \ll m$, the node is lagging many rounds behind other nodes, thus the round-by-round synchronization will be

⁴Note that the difference in the digest could also be due to the sender of digest(m) lagging behind, therefore another design option is for node N to reply to the Sync Interest with digest(m) with a data packet containing N 's known data names for round m .

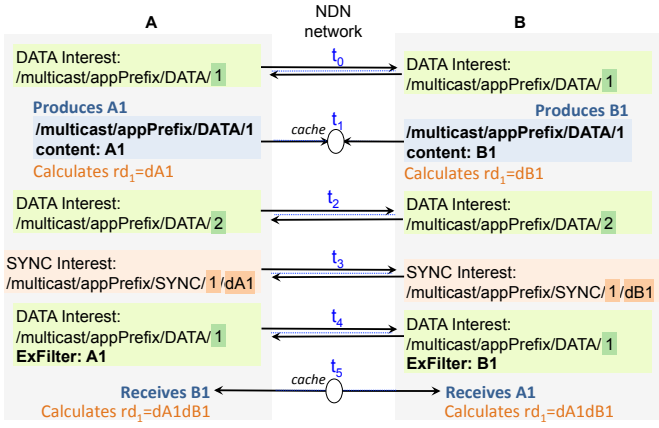


Fig. 4. Simultaneous data publishing in round 1.

slow. In this case, N may decide to jump to the latest round it is aware of (i.e., round $m + 1$), and use the recovery mechanism (described in Section V-F) to catch up with the missing data in the earlier rounds.

E. Simultaneous data publishing

The inconsistency detection mechanisms described in the previous subsection automatically handle simultaneous data production scenarios. When two sync nodes happen to produce data simultaneously in the same round n , both of them will generate replies to the same Data Interest for round n . Depending on the underlying network topology, other nodes in the group will receive one of those replies but not both, causing the group to diverge into two inconsistent states.

According to the described rule for $n = m$ case: each node will issue a new Sync Interest carrying one of the two possible round digests. Upon receiving the Sync Interest with a different digest, a node will detect the inconsistency and send Data Interests with exclude filter to retrieve the missing reply. After each node receives the reply, they will converge on the same state for round n and send out Sync Interest with the same round digest. This description can be generalized to the cases where k out of N nodes ($1 < k \leq N$) publish new data simultaneously in the same round, as long as k is within the limitation of the exclude filter size.

Fig. 4 shows an example with two nodes simultaneously publishing data in the same round and eventually resolving the conflict.

F. Recovery from Cumulative Inconsistency

In addition to round-by-round synchronization, RoundSync provides a simple way to detect the overall dataset inconsistency using *cumulative digests*, and designs a recovery mechanism to reconcile the inconsistency discovered by the cumulative digests.

1) *Cumulative digests*: Different from a round digest which covers only the data published in a specific round, the cumulative digest for round n covers the whole dataset observed by each sync node in that round. Whenever a node produces

Rounds Log			
Round N	rd _N	cd _N	User data names
1	1a...a	1a...a	[Alices's prefix,1]
2	b2...b	cd...6	[Alices's prefix,2], [Bob's prefix,1]
3	4c...c	7a...4	[Daniel's prefix,1], [Karen's prefix,1]
4	12...9	32...5	[Karen's, prefix,2]
5	23...d	ac...4	[Alices's prefix,3], [Daniel's prefix,2]
6	cd...3	89...1	[Daniel's prefix,3]
7	65...2	bb...3	[Karen's prefix,3]
8	ab...9		[Bob's prefix,2]
9	32...9		[Alice's prefix,4]
10	23...4		[Karen's prefix,4]

Fig. 5. Stable round and calculation of cumulative digests.

new data, it piggybacks the cumulative digest for the latest stable round n in the reply packet to the Data Interest in the current round m . The piggybacked information in the data reply includes the following three pieces:

- the node prefix of the producer who generates this data reply,
- the value of the cumulative digest for round n , and
- the round number (n) that the cumulative digest is generated for; note that $n < m$, as the current round m is yet to stabilize.

Upon receiving a Data Interest reply with the cumulative digest for round n ,

- if the node has not calculated a cumulative digest for that round yet (e.g., because it is still actively synchronizing in the rounds up to n), it may delay the processing of the received cumulative digest until it has calculated one itself.
- if the node has calculated its own cumulative digest(n), it verifies the consistency of the entire application dataset up to the round n . If the values differ, the node starts the recovery process as described next.

Note that sending cumulative digests for recent rounds that are yet to achieve complete synchronization may unnecessarily trigger multiple nodes to take recovery actions. To reduce the chance of premature recovery actions, a sync node should only include the cumulative digest for the latest round n up to which no change has been made for a sufficiently long time. That is, all the rounds up to n have already stabilized. For example, in Fig. 5 the Rounds Log includes rounds 1 through 10, and the previous highest stable round number is 4 whose cumulative digest has been piggybacked in the Data Interest replies generated for rounds 7 through 10. After the node detects that no more data has been produced for rounds 5 through 7 for a long enough time, it will mark those rounds as stable, calculate the cumulative digest up to round 7, and piggyback that information in future Data Interest replies (for the rounds after 10).

2) *Recovery process*: To recover from a cumulative digest inconsistency with node R and R 's cumulative digest $D_{R(n)}$, a node A first determines whether the value of n is very close to A 's current round m ; if so, it means that A is following the group's data production closely and the inconsistency may be

recovered by using round-by-round synchronization for a few round right before round n .

If $m \ll n$, or if A fails to re-synchronize after trying a few round-by-round synchronization, A will invoke a recovery process by sending a Recovery Interest using R 's unicast prefix learned from the data reply, concatenated with the cumulative digest value $D_{R(n)}$. In addition, node A also produces a Data Interest reply to the current round with no update but its own prefix and its cumulative digest for round n . This reply is sent after a small random-wait time; the node suppresses its own reply if another node sends out a similar reply before it does. Doing so is to inform the remote node R about the inconsistency, triggering R to send a Recovery Interest to A as well.

Note that, except special cases where every node publishes at each round, in general nodes do not keep a complete list of every node's sequence number, i.e. the entire application dataset state, for every round. Therefore when a node receives a Recovery Interest for round n , it cannot reply with the application dataset state for that round. Instead, the reply to a Recovery Interest contains the current application dataset state and the current round number m of the sender.

Node A who receives the recovery reply data packet updates its own application dataset with the received one and stores the updated dataset temporarily in a data structure called *recovery data names collection*. Node A also moves to round m if m is greater than A 's current round number. The new round is designated as the *recovery round*. Since its previous sync state is unsynchronized, the node will delete all the cumulative digests it has calculated before, and purge the Rounds Log to reclaim storage. Once the recovery round becomes stable (i.e., no data is published in that round for sufficiently long time), the node calculates the cumulative digest for the recovery round using the dataset stored in the recovery data names collection, and starts calculating cumulative digests for the future rounds.

Fig. 6 shows an example of the recovery process. When a recovery reply containing the application dataset and round number 103 is received, the node who is currently in round 71 updates its own application dataset and jumps to round 103 immediately, marking it as the recovery round. It also truncates its original Rounds Log and removes all entries before round 71. After round 103 through 106 is stabilized, the node uses the recovery data names collection and the additional updates received in each round to calculate the cumulative digests for rounds 103 to 106.

VI. CONCLUSION

RoundSync is motivated by the observed semantics overloading on the Sync Interest in the original ChronoSync design. Because a Sync Interest with name M is used both for dataset state inconsistency detection (hence M carries the state digest D), and for update retrieval (hence all the new updates produced in the same state generated in the state represented by D will share the same prefix M), this semantic overloading on the name prohibits retrieval of new updates if nodes are

Rounds Log			
Round N	rd _N	cd _N	User data names
70	54...1		[Alice's prefix, 50]
103		dc...4	
104	cd...3	12...7	[Daniel's prefix, 31]
105	65...2	ad...f	[Karen's prefix, 26]
106	ab...9	43...7	[Bob's prefix, 21]
107	32...9		[Alice's prefix, 61]
108	23...4		[Karen's prefix, 27]
109	17...8		[Bob's prefix, 22]

Recovery Data Names Collection	
pi	sj
Alice's prefix	60
Bob's prefix	20
Daniel's prefix	30
Karen's prefix	25

Fig. 6. Updating Rounds Log after receiving recovery reply

out of dataset synchronization, and the recovery process can be expensive. This once again shows the importance and challenge in designing a proper naming scheme for network protocols running over NDN.

RoundSync modifies the ChronoSync design in several aspects. First, it divides the data productions into rounds and restricts each node to publish at most one data packet in a single round, reducing the chance of too many updates produced in the same round. Second, each round can be synchronized independently from any other round, and does not affect data publication in any round. Third, RoundSync uses separate Interest packets to perform state inconsistency detection (Sync Interest based on the round digest) and update retrieval (Data Interest based on the round number), enabling a node to retrieve updates without knowing other nodes dataset state. Finally, RoundSync redesigned the recovery mechanism from cumulative digest inconsistency, which is performed between node pairs without involving the whole group.

The above modifications improve RoundSync's scalability of data synchronization for the scenarios with high data production concurrency among the distributed data producers. We would also like to express a concern about the usage of RoundSync's Sync Interests: they solicit no return data packets, which is inconsistent with NDN's Interest-Data flow balance. Please see [1] for further discussions on the impact of such Interest usage.

We have implemented the RoundSync protocol and conducted preliminary comparison studies between ChronoSync and RoundSync through simulations; the results will be reported in a future revision of this report.

REFERENCES

- [1] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "An Overview of Distributed Dataset Synchronization in Named-Data Networking," NDN Project, Technical Report NDN-0053, April 2017.
- [2] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2656877.2656887>
- [3] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, Goettingen, Germany, October 2013. [Online]. Available: <http://icnp13.informatik.uni-goettingen.de/index.html>
- [4] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," *Proc. of SIGCOMM*, 2011.

- [5] Y. Minsky et al., "Set reconciliation with nearly optimal communication complexity," *IEEE Trans. Info. Theory*, 2003.
- [6] J. Feigenbaum et al., " l^1 -different algorithm for massive data streams," *SIAM Journal on Computing*, 2002.