Improvements to PSync: Distributed Full Dataset Synchronization in Named-Data Networking

Ashlesh Gawande

IMPROVEMENTS TO PSYNC: DISTRIBUTED FULL DATASET SYNCHRONIZATION IN NAMED-DATA NETWORKING

by

Ashlesh Gawande

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Major: Computer Science

The University of Memphis

December 2019

Abstract

TCP/IP is ill-suited for modern many-to-many and distributed applications such as content distribution and IoT. Named-Data Networking (NDN) is a future Internet architecture design based on the primitive of publishing and fetching authenticated named data objects/chunks. NDNs name-based retrieval, stateful forwarding, data-centric security, and in-network caching overcome many shortcomings of today's TCP/IP networks due to their host-centric data delivery and channel-based security models.

Applications in NDN, such as chat, require distributed synchronization of all published data. This thesis evaluates PSync, a distributed dataset synchronization protocol for NDN, and proposes a set of improvements to PSync. ChronoSync is an existing synchronization protocol in NDN providing the same functionality as PSync. Named-Data Link State Routing Protocol (NLSR) is an NDN routing protocol which needs a synchronization protocol to share its state database with other instances. We compare PSync with ChronoSync to show its superior performance in supporting the data distribution in NLSR.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Synchronization is an omnipresent phenomenon in the modern Internet. A simple chat application is expected to deliver a message to all the devices of the recipients in real time. In today's centralized model of popular applications it is relatively easy to achieve this synchronization via a central server. Distributed communication is harder to materialize as there is a mismatch between the TCP/IP architecture's greed for point-to-point communication and an application's desire for content. To make communication simple for distributed applications, Named-Data Networking (NDN) [1] is designed to work on the abstraction of Named objects/chunks that are directly related to the application semantics. An Interest packet in NDN contains a Name that is used by applications to fetch a Data packet matching that name. An Interest can be forwarded to multiple destinations marking the inherent support of multicast communication in NDN. The naming and the multicast model is very useful in designing distributed synchronization protocols for NDN.

Such synchronization protocols have shown promise of wide utility in NDN. Routing protocols and chat applications are the early adopters of synchronization in NDN. Recently, PartialSync protocol [2] was proposed for entities to share only a subset of their data. It employed the power of an Invertible Bloom Filter (IBF, also known as Invertible Bloom Lookup Table or IBLT) [3] for efficiently calculating the difference between two datasets. It was later extended to support full dataset synchronization between multiple parties and the full dataset sync was renamed as PSync full sync (hereto referred as just PSync). This thesis explores the performance shortcomings of the existing PSync, suggests improvements, and compares its performance to the current state-of-the-art sync protocol in NDN.

Chapter 2

Background

2.1 NDN Forwarder

Named-Data Networking realizes the promise of networking over content chunks by using Interest and Data packets as specified in version 0.3 of the protocol [4] and visualized in Figure 1. An NDN forwarder abstracts all its network interfaces and application end points as a *Face*. Interests and Data packets are received and forwarded through these Faces by the forwarder.

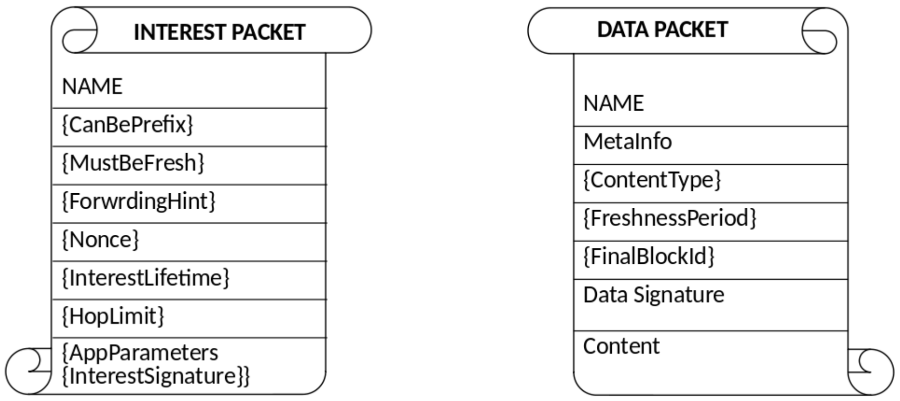


Figure 1. NDN Interest and Data Packets

Figure 2. shows the overview of forwarding and receiving of packets in an NDN forwarder as it is implemented in NFD at the time of this thesis [5]. The forwarder contains a Content Store (CS) which caches every legitimate Data packets that are received. Unsolicited data, for which an Interest was not expressed is dropped by the default policy. When an Interest arrives the forwarder tries to satisfy the Interest from the CS. Upon lookup failure, the forwarder creates an entry in the Pending Interest Table (PIT). The PIT entry has a lifetime equal to the InterestLifetime field specified by the Interest. If the Interest specifies the mustBeFresh flag then only the Data which has a freshness period that is not yet expired can satisfy it. If no Data corresponding to the Interest Name is found in the CS, the Forwarding Information Base (FIB) can be used to look up the Face where the Interest can be forwarded by the Strategy. If a PIT entry for the

Interest already existed, its lifetime is increased and the sender's Face is inserted into the entry if not present. When the Data packet comes back, the PIT is consumed and the Data is forwarded to all the Faces listed in the PIT. This is known as Interest aggregation at the PIT.

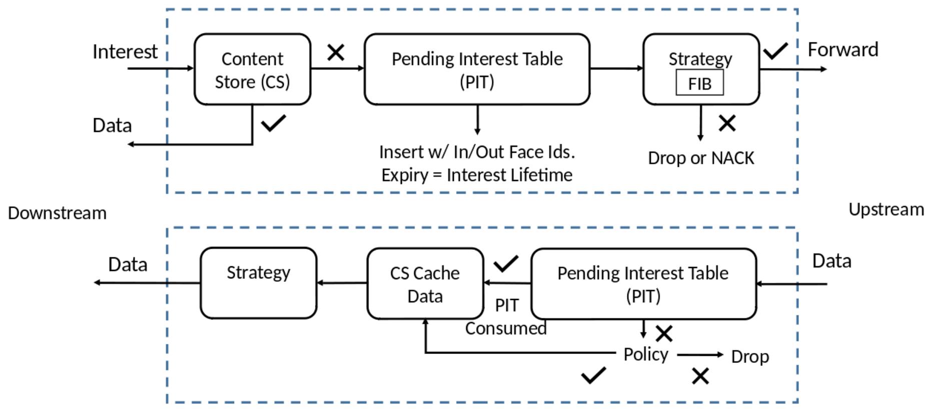


Figure 2. NDN Forwarder Overview

As mentioned above, the forwarder does not directly take the decision to forward based on the FIB. Instead, it hands the Interest to the Strategy which can forward the Interest on one or all the Faces listed in the FIB. Different Strategies can be set for handling Interests of different Names by an Operator or an application. If there is no entry in the FIB, a Strategy may choose to send a NoRoute NACK to the sender instead of forwarding the Interest and delete the PIT. The default Strategy used by the Named-Data Forwarding Daemon (NFD) [5], an implementation of NDN forwarder, is BestRoute. The BestRoute Strategy forwards the Interest to the lowest cost Face (as installed by the Operator or some routing mechanism). In contrast, the Multicast Strategy, forwards an Interest to all the Faces. When a Data packet comes through one of the Faces, it is sent to all the other Faces consuming the PIT. If a NACK is received on all the faces, the application is sent a NACK and the PIT entry is consumed.

2.2 Synchronization protocols

In a distributed synchronization protocol, a node needs to reconcile its set of data with other nodes in the network. A node here refers to sync protocol application running on a device. To communicate with other nodes on the network, a sync protocol must use the same Name prefix for their sync Interests and Data. This sync prefix should also be routable such that a sync node can reach other sync nodes. A sync protocol can then utilize the Multicast Strategy provided by NDN to send its sync Interests. Shang et al. provide a Survey of Distributed Dataset Synchronization in Named Data Networking [6] and note that all the existing sync protocols use this strategy. Using this approach, sync Interest aggregation can be done at nodes to efficiently propagate data changes by answering the pending sync Interest. Hence the sync Interest must carry a representation of the state at the sender node which will be the same as the state at other nodes when all the nodes are synchronized. This is illustrated in Figure 3. Here AppA, AppB, and AppC are applications who send the sync Interest. NFDs running on the nodes A, B, and C put this Interest in their PIT and forwards it via Multicast Strategy to all the neighbors. The PIT on node B is shown with the same Interest pending on all of its interfaces. One is from the application AppC and others are from NFDs on A and C. If AppA decides to publish some Data, these pending sync interests will be answered and sync Data will reach the applications AppB and AppC.

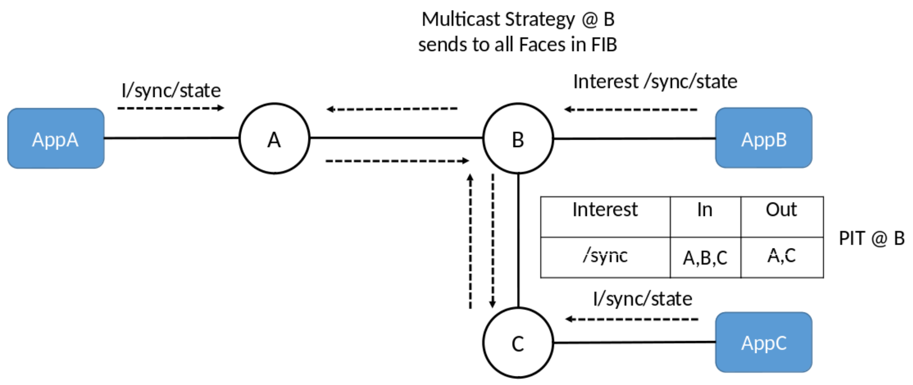


Figure 3. Multicast Interest aggregation

The sync protocols differ from each other on their representation of the state of the data at a node as well as how the protocol allows the application to insert data. The survey discovered that protocols such as CCNx 0.8 Sync, iSync, CCNx 1.0 Sync allow applications to insert and synchronize arbitrary names i.e. any form of names desired by the application. Whereas protocols such as ChronoSync and PSync limit the applications names to the format <user-prefix>/<seq-no> to discover changes based on a prefix and a sequence number. This improves efficiency as well as identifies the application instances. Therefore this thesis will base its comparison against ChronoSync. Named-Data Link State Routing (NLSR) [7] protocol has also used ChronoSync since its inception to synchronize its Link State database. Only recently was NLSR modified to provide the operator a choice of sync protocols between ChronoSync and PSync. Even though NLSR with PSync as the routing protocol has been successfully running on the NDN global testbed [8] its performance as compared to ChronoSync has been a mystery until this work.

Chapter 3

Related works

So far we have established that sync protocols use multicast communication and state representation to provide the name synchronization service to the application. We will now look at how ChronoSync and PSync achieve this synchronization.

3.1 ChronoSync

In ChronoSync [9], the state of a node is represented by a 2-level Merkle tree, called sync tree. A leaf of the tree contains a cryptographic hash or digest of a user's Name prefix and sequence number. The root of the tree is the cryptographic digest of the cumulative hashes of each leaf arranged in a lexicographical order. The order is important for each node to calculate the same root digest. Figure 4. shows an example of a node's state in ChronoSync.

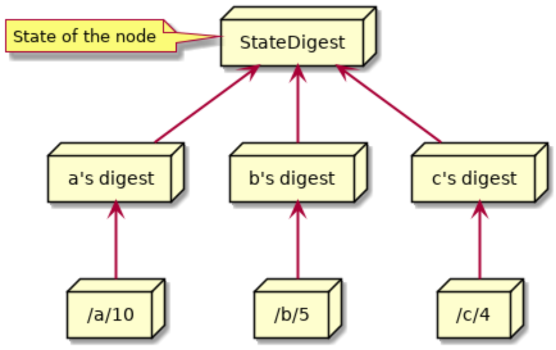


Figure 4. ChronoSync state representation

It is the state digest that is appended to the prefix of the sync Interest and multicasted to the network. Sync Interest Name as well as Sync Data Name is of the form **/<sync-prefix>/state-digest**. A simple scenario is shown in Figure 5. Initially, the state digest of both nodes A and B are empty. When node A publishes /A/1, it updates its own sync tree and its state digest becomes <12c...>. It then replies to the pending sync Interest from B and renews its own sync Interest as it would be consumed by its own reply. Upon receiving the sync Data, node B also updates its sync tree and renews its Interest with the latest state digest.

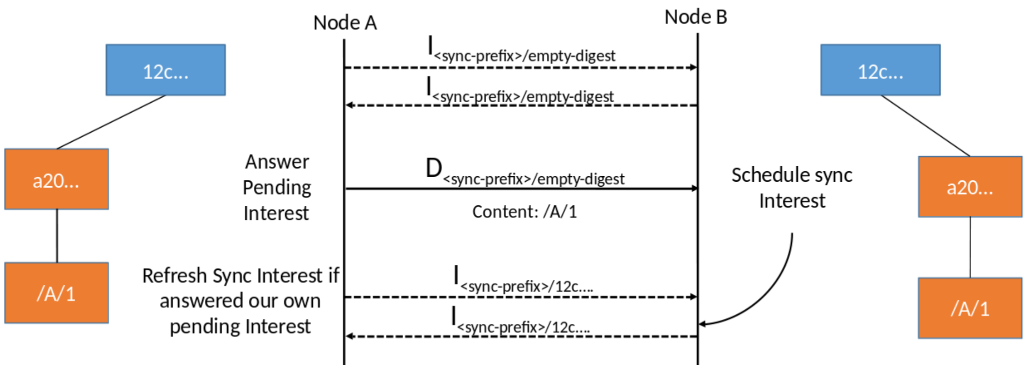


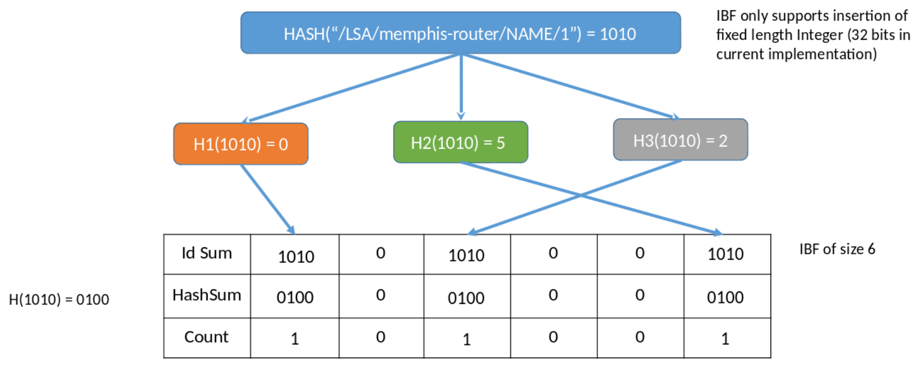
Figure 5. ChronoSync simple sync scenario

More complex scenarios, such as simultaneous publish of data by multiple nodes and network partition, result in node receiving unrecognized digest. In such a case, a ChronoSync node waits for a while before answering the Interest as some new data may render the digest recognizable. So ChronoSync no longer uses the exclude filter to exclude sync Data replies which it already has. If after the waiting period, ChronoSync cannot recognize the digest it sends a recovery Interest which has the Name of the form **/<sync-prefix>/recovery/state-digest**. A node which recognizes the digest in the recovery Interest will answer with all the data they have and bring the node up to speed. Apart from the digest tree, ChronoSync also maintains a digest log that contains a history of state digests so as to quickly recognize and reply to old digests.

There used to be an exclude filter in the Interest packet which can be used to exclude Data already received but it is removed in the latest protocol specification. Hence ChronoSync has to fall back on recovery more often.

3.2 PSync

In PSync protocol, a node represents its state with an Invertible Bloom Filter (IBF) [10]. Eppstein et al. explain IBF and the details of its operation in [10]. IBF is a probabilistic data structure containing an array of entries or cell. Each cell has an idSum, hashSum, and a count. A fixed-length data is inserted into the IBF by hashing it with a list hash functions to get the cell locations where it will be inserted. The data is XOR'd into the idSum, the hash of the data is XOR’d into the hashSum, and the count is increased by one. Figure 6. shows the insertion of the data d into an empty IBF.


HASH(“/LSA/memphis-router/NAME/1”) = 1010
IBF only supports insertion of fixed length Integer (32 bits in current implementation)
H1(1010) = 0
H2(1010) = 5
H3(1010) = 2
IBF of size 6
H(1010) = 0100

Id Sum	1010	0	1010	0	0	1010
HashSum	0100	0	0100	0	0	0100
Count	1	0	1	0	0	1

Figure 6. Insertion into an empty IBF

Removal operation is same except that the cost is decreased. If the count of the cell is 1 and the hash of the data (idSum) is equal to the hashSum then such a cell is pure and we have retrieved a data from the IBF. We can remove this data from the IBF which may unravel more pure cells. This process can be repeated until no more pure cells are found. In PSync, the IBF is directly appended to the sync Interest. When a node with IBF1 receives the sync Interest with a different IBF, IBF2, then IBF2 can be subtracted from IBF1: IBF3 = IBF1 - IBF2. This is an efficient operation where each idSum and hashSum from IBF1 are XOR’d with IBF2’s and each cell’s count in IBF2 is subtracted from IBF1’s producing a new IBF, IBF3. In IBF3 we can look for cells that are pure. Here the count can be either 1 or -1 for a pure cell. A pure cell with a count of 1 means IBF2 is missing the data which IBF1 has. Such a data is added to a positive set. A count of -1 means the opposite and is added to a negative set. If IBF3 decode i.e. the extraction of elements from IBF3 is successful then all the cells will have its constituents as zero. On the other hand, if the decode fails then we were not able to find all the differences between IBF1 and IBF2.

The chance of successful decode between two IBF depends on the number of differences between them. However, there is a high probability of successful decode if the number of cells in an IBF is equal to 1.5 times the number of expected cells [10]. Hence if the number of updated prefix between two IBFs exceeds 1.5 times the size of the IBF, it is likely that the decode will fail. Currently, the application selects the size of the IBF and it is expected to be the same across all the sync application instances.

Note that in PartialSync [2], <user-prefix>/<seq-no> is inserted into the IBF after being hashes to a fixed-length data. The previous sequence for this user-prefix is removed from the IBF. This practice remains the same in PSync. Hence the state of a node is shown in Figure 7.

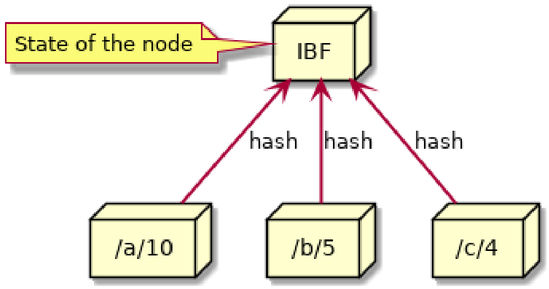


Figure 7. PSync state represented by an IBF

Like ChronoSync, PSync multicasts a sync Interest periodically. Sync Interest Name is of the form **/<sync-prefix>/IBF**. The IBF in the Interest is compressed to reduce the size of the Interest. A simple scenario with 2 nodes is demonstrated in Figure 8. Node A and B start and have a pending sync Interest towards each other with the same empty IBF, IBF1. After some time, node A publishes an update for prefix /A with sequence 1 and answers the pending Interest with the sync Data. Sync Data Name is of the form **/<sync-prefix>/IBF1/hash(IBF2)**. IBF2 is of no use to node B as it will update its own IBF1 upon receiving the data. Hence A only appends the hash to make the Data reply separate from other replies in case there are any. However, note that a second sync Data with a different hash will never make it to B since its sync Interest is satisfied. After A has sent the Data it also consumed its own pending Interest and must renew the sync Interest.

In a more complex scenario, the IBF received may not match the node's own IBF. In this case, if the subtraction of received IBF from the node's own IBF is successful, a

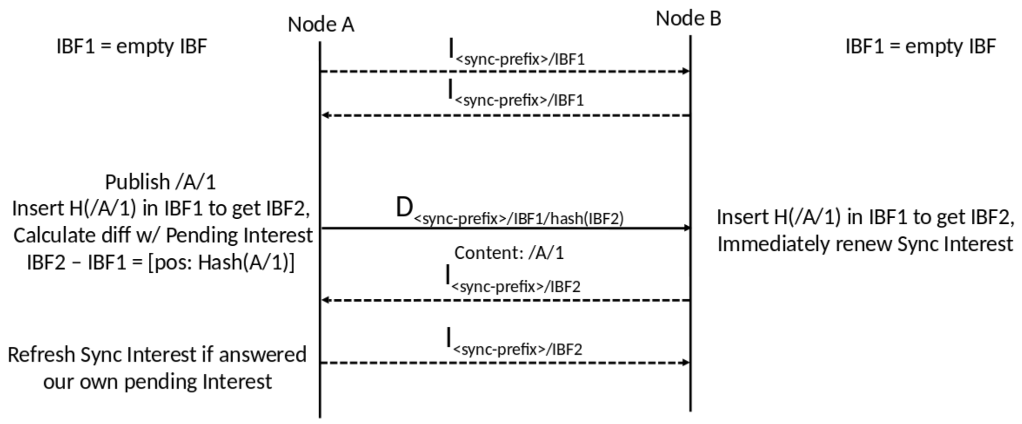

Node A
Node B
IBF1 = empty IBF
IBF1 = empty IBF
I<sync-prefix>/IBF1
I<sync-prefix>/IBF1
Publish /A/1
Insert H(/A/1) in IBF1 to get IBF2,
Calculate diff w/ Pending Interest
IBF2 – IBF1 = [pos: Hash(A/1)]
D<sync-prefix>/IBF1/hash(IBF2)
Content: /A/1
Insert H(/A/1) in IBF1 to get IBF2,
Immediately renew Sync Interest
I<sync-prefix>/IBF2
Refresh Sync Interest if answered our own pending Interest
I<sync-prefix>/IBF2

Figure 8. PSync simple sync scenario

node knows what elements the other side is missing and sends the elements to it. However, there could be a situation where a node thinks that it has found some missing data as the positive set contains one element upon decode. But it is in fact the hash corresponding to the /some-prefix/1 and the negative set contains a hash of /some-prefix/2 which the node cannot recognize. The scenario of multiple nodes publishing which leads to the detection of future hash is shown in Figure 9. A heuristic in this case that PSync applies is to check whether /some-prefix/seq+1 exists in the negative set. In such a case PSync does not send this prefix in its sync reply.

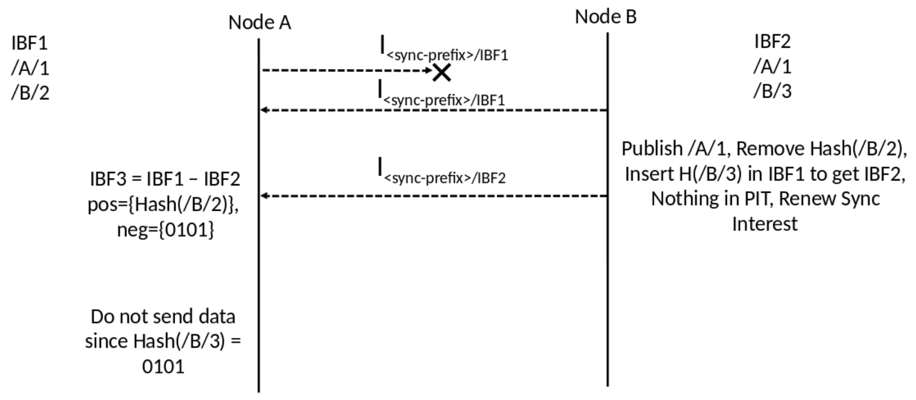

Node A
Node B
IBF1
/A/1
/B/2
IBF2
/A/1
/B/3
I<sync-prefix>/IBF1
I<sync-prefix>/IBF1
Publish /A/1, Remove Hash(/B/2), Insert H(/B/3) in IBF1 to get IBF2, Nothing in PIT, Renew Sync Interest
I<sync-prefix>/IBF2
IBF3 = IBF1 – IBF2
pos={Hash(/B/2)},
neg={0101}
Do not send data since Hash(/B/3) = 0101

Figure 9. PSync receives no new update

In another scenario such as a network partition or multiple updates can lead to the difference with the received IBF unable to be decoded successfully. In this case, shown in

Figure 10., PSync sends all of its dataset to other side. This is similar to ChronoSync sending its entire dataset on a recovery Interest. Unlike ChronoSync, PSync segments its data in case it is bigger than the NDN packet MTU of 8800 bytes. To not go over the MTU size, ChronoSync compresses the data content and recovers in multiple sync cycles if the data size does go over the MTU. PSync currently does not compress its data.

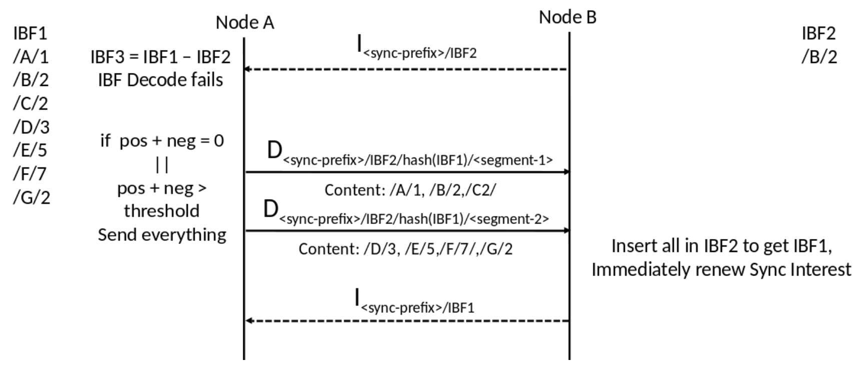


Figure 10. PSync recovery on IBF difference decode failure

The activity diagram of PSync on receiving a sync Interest is given in Figure 11. PSync extracts the compressed IBF from the Interest and decompresses it. It then subtracts the received IBF from its own and sees if the difference can be successfully decoded. If it can be successfully decoded than all elements that are missing from the other side i.e. the one in the positive set are looked up in a hash-to-prefix mapping and added to the sync Data content if the hash of prefix with the sequence number incremented by 1 is not found in the negative set. If the other side is not missing anything, then this sync Interest is saved in the pending sync Interest table for the duration of the Interest lifetime. This pending sync Interest can be answered when this node publishes some Data to immediately notify other nodes.

Figure 12. shows PSync's response on receiving sync Data. It iterates through the list of prefix/sequence received in the content and inserts the hashes of those into its own IBF for which the known sequence number is smaller than the received. Currently, PSync does not react if a NACK is received and lets the sync Interest be on its own schedule.

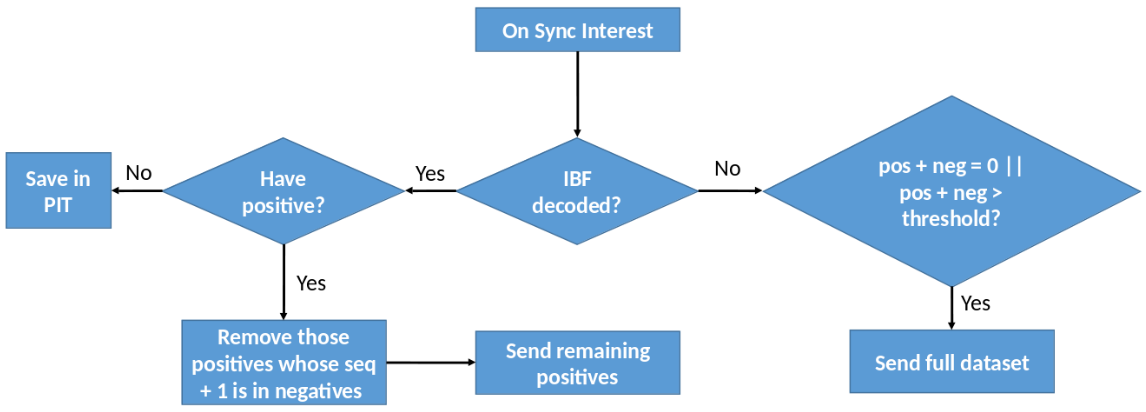


Figure 11. PSync on Sync Interest

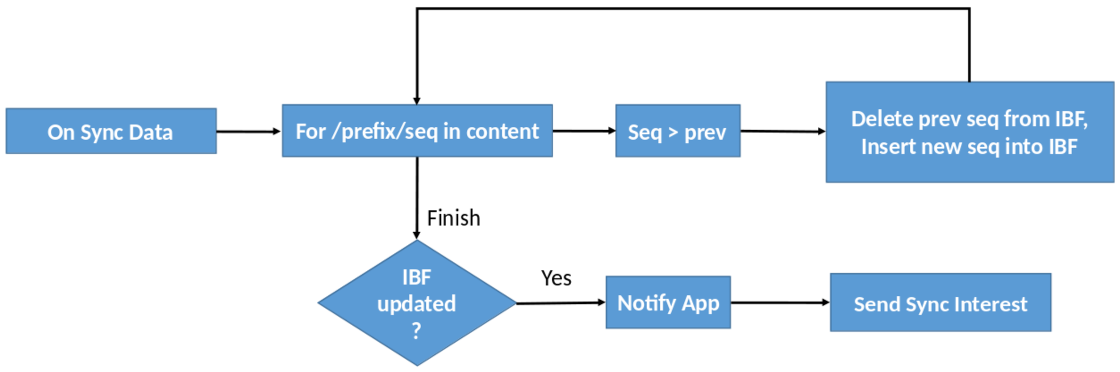

On Sync Data
For /prefix/seq in content
Seq > prev
Delete prev seq from IBF, Insert new seq into IBF
Finish
IBF updated ?
Yes
Notify App
Send Sync Interest

Figure 12. PSync on Sync Data

Chapter 4
Problems with PSync

We discovered that PSync is slow to fetch changes and aggressive in sending full dataset. As a result, it is unable to converge with low IBF sizes. Ideally we would like for NLSR to converge with a small IBF size such as 9 which means at most 6 differences can be decoded successfully between two IBFs with high probability. These problems are not discovered with the NLSR running on the testbed as the default IBF size of 80 is quite high and hence decode errors do not occur. The downside of having a large IBF is that Interest and Data packet sizes are very large increasing the overhead of the protocol significantly. This may also result in packets being fragmented on the network if the packet size is greater than the network MTU and cause other performance problems.

4.1 General problems

PSync uses a jitter value that is 20 percent of the sync Interest lifetime which can be on the order of seconds. This jitter is used when scheduling sync Interest being sent so that not all the nodes do it at the same time and for other purposes such as delaying the processing of an incoming sync Interest.

PSync does not react to NACK. A NoRoute NACK could happen if the route has not been registered yet in NFD and PSync has been started. This forces PSync to wait till the next sync period to send the sync Interest. ChronoSync shares this same problem, but recovers more quickly as it reacts on receiving unknown digest from the side by scheduling a sync Interest.

The appending of the hash of our own IBF to the sync Data Name is of no use to the receiver. ChronoSync does not append its old state digest to the sync Data Name. Doing so only increases the size of the sync Data.

4.2 Sync data with all elements are sent after decode failure by every node

Since every node replies with their full dataset on decode failure, some nodes end up sending dataset which does not contain anything new for other nodes as these nodes are actually behind. Since this data packet with no new update takes up the CS for its freshness period, the nodes which received it are unable to send any new sync Interests until the Data expires from the local CS as well as CS of other nodes in the network.

4.3 Sync Interest is not sent after decode failure or if we are behind after decode success

If the IBF size is low there are a lot of decode errors when trying to decode the difference of IBFs. In the current model, we do not send a sync Interest after a decode failure occurs, so PSync is unable to recover fast. At this point either the data is on its way or our Interest is lost in the network or discarded by other nodes. If no sync Interest from us exists, PSync has to wait till the next scheduled sync Interest whose period can be set quite high by a user. ChronoSync does not suffer from this problem, as it starts recovery after a while of not recognizing the state digest.

Similarly, if the decode is successful but the node is behind according to the elements in the negative set, PSync does not schedule a sync Interest. PSync also adds this sync Interest to its pending Interest table which is not equal to its current state.

4.4 Future hash detection is very limited

Currently, PSync detects if it is behind via calculating if the hash of prefix/(seq + 1) it is about to send exists in its negative set (the elements other side is supposed to be missing) as mentioned in section 3.2. If a sequence number greater than the increment of one exists in the negative set, the node will send the data that is behind the network resulting in no new update being received by other nodes. This again blocks other nodes from sending another sync Interest until this data expires from the CS similar to the above scenario. This scenario can be seen in Figure 13. The behavior of sending old data will continue to happen until the node that is behind receives the update and removes the old sequence number from its own IBF. Since PSync allows an application to set the sequence numbers, it can set it to the previous sequence number + 10. In fact this is what happens when NLSR recovers from a failure.

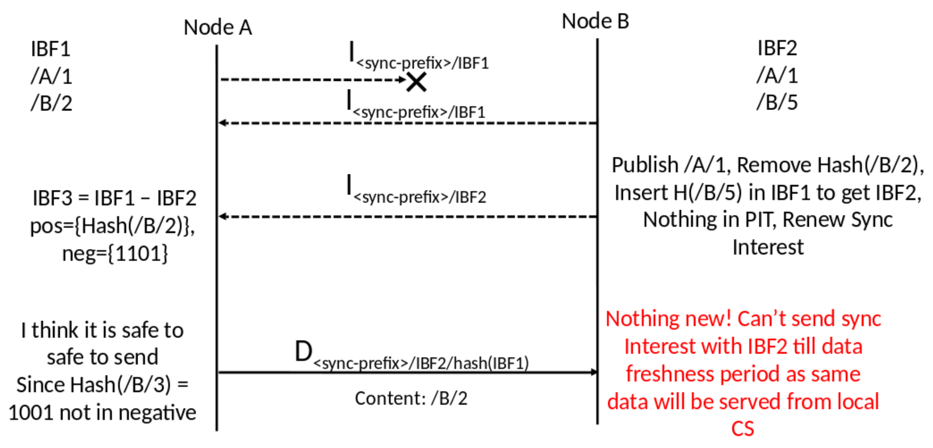

Node A
Node B
IBF1
/A/1
/B/2
IBF2
/A/1
/B/5
I<sync-prefix>/IBF1
I<sync-prefix>/IBF1
Publish /A/1, Remove Hash(/B/2), Insert H(/B/5) in IBF1 to get IBF2, Nothing in PIT, Renew Sync Interest
IBF3 = IBF1 – IBF2
pos={Hash(/B/2)},
neg={1101}
I<sync-prefix>/IBF2
Nothing new! Can't send sync Interest with IBF2 till data freshness period as same data will be served from local CS
I think it is safe to safe to send Since Hash(/B/3) = 1001 not in negative
D<sync-prefix>/IBF2/hash(IBF1)
Content: /B/2

Figure 13. PSync receives no new update due to limitations of future hash detection

4.5 Immediately sending sync Interest after new Sync Data is received

When Sync Data with new updates is received, PSync sends a sync Interest immediately. If there are multiple updates in a short period of time this will send many sync Interests. When a new sync Interest is sent, the old one is removed from the face by PSync and hence cannot bring back Data.

Chapter 5

Solution I : Arbitrary names

We started by fixing small problems to see if it we can get convergence for a 10 node topology that is an old snapshot of the NDN testbed. The IBF size we kept was 9, able to decode 5 differences between two IBFs with high success. In a run with PSync from the master branch we saw that the 10 node topology does not converge. The convergence time is set to 60 seconds which should be more than enough for the topology sizes we use. We propose the following changes that enables PSync to converge with IBF of size 9 with topology sizes ranging from 10 to 200 and under various experiments of NLSR.

5.1 Reacting to NoRoute NACK and shortening the jitter

The first problem that we discovered and fixed was PSync not reacting to NoRoute NACK. So we made the change to send a sync Interest if a NACK is detected after a jitter. Jitter was set to 20 percent if the sync Interest lifetime. Since NLSR sets a sync Interest lifetime of 60 seconds, our jitter was 12 seconds. We now use the jitter of 100 milliseconds to 500 milliseconds. Anything shorter than that would put extra pressure on NFD by sending repeated Interest before the route is up. Anything larger tends to slow down convergence. Figure 14. shows the change with the NACK reaction.

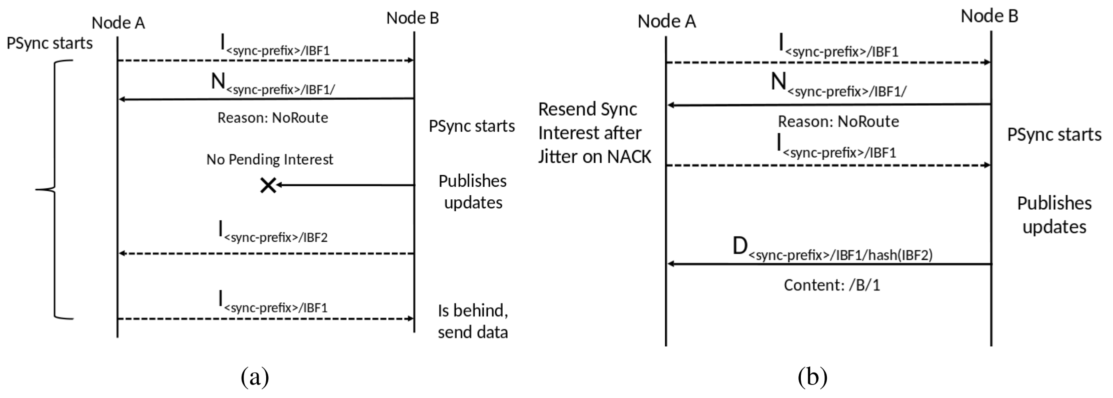

Node A
Node B
PSync starts
$I_{\text{<sync-prefix>/IBF1}}$
$N_{\text{<sync-prefix>/IBF1/}}$
Reason: NoRoute
PSync starts
No Pending Interest
Publishes updates
$I_{\text{<sync-prefix>/IBF2}}$
$I_{\text{<sync-prefix>/IBF1}}$
Is behind, send data
(a)
Node A
Node B
$I_{\text{<sync-prefix>/IBF1}}$
Resend Sync Interest after Jitter on NACK
$N_{\text{<sync-prefix>/IBF1/}}$
Reason: NoRoute
$I_{\text{<sync-prefix>/IBF1}}$
PSync starts
Publishes updates
$D_{\text{<sync-prefix>/IBF1/hash(IBF2)}}$
Content: /B/1
(b)

Figure 14. (a) With no NACK reaction (b) With NACK reaction

5.2 Not appending IBF hash to sync Data Name

To reduce the sync Data overhead PSync was changed to no longer append the hash of its own IBF to the sync Data Name given that the hash does not serve any purpose to the receiver.

5.3 Reacting to incoming sync Interests

We then started to react to sync Interest when the decode was successful and the presence of elements in negative set told us that we are behind. This was done by waiting for a while and then sending the Interest if we are still behind. This is shown in Figure 15. When the decode fails, we need to react in a similar fashion as when the decode is successful but we usually find the negative set to be empty when the decode fails. If we always react, we increase our chance of ending up with no new updates. We needed a better idea of whether we are ahead or the sender is ahead.

5.4 Adding the number of elements in the IBF to the Interest Name

It is hard to know the number of elements that has been inserted into an IBF. The count fields can put bounds on the number but cannot know the specific pattern of insertion of elements to get the exact count without exhaustively going through all the permutations. But if we do know the number of elements on the received IBF, a node can take a more educated guess on a decode failure based on the number of elements it has in

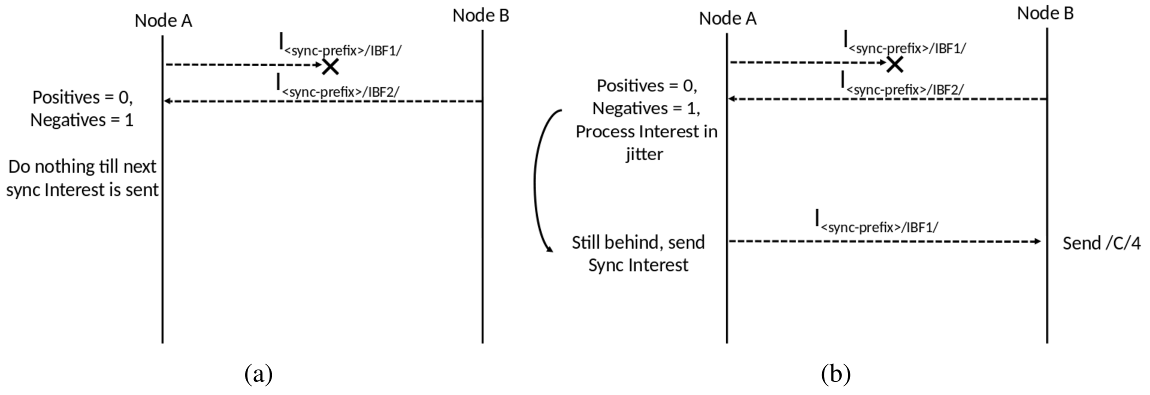

Node A
Node B
I<sync-prefix>/IBF1/
I<sync-prefix>/IBF2/
Positives = 0, Negatives = 1
Do nothing till next sync Interest is sent
(a)
Node A
Node B
I<sync-prefix>/IBF1/
I<sync-prefix>/IBF2/
Positives = 0, Negatives = 1, Process Interest in jitter
I<sync-prefix>/IBF1/
Still behind, send Sync Interest
Send /C/4
(b)

Figure 15. (a) No reaction on negative detection (b) Processing and reacting in jitter on negative detection.

its own IBF. If a node has more number of elements it can answer an incoming IBF which failed to decode immediately with all its data. But if it has fewer number or equal number of elements it can delay the processing of the sync Interest to a later time in the hope to recover. This is effective in reducing the number of sync Data being sent, especially by nodes who just joined the network and are way behind. Hence the sync Interest Name is now changed to: **/<sync-prefix>/IBF/<num-elements-in-IBF>**. The changes are shown in Figure 16.

There is a hidden problem with this approach. A node can be behind from its neighbor but think it is ahead because of the future hash problem (section 3.2). So its number of elements can be equal to the number of elements received. The trick to detect the future hash cannot be applied here because there is a decode failure and we lack information on positives and negatives. In this case, the node will process the Interest later. The other node would also process this Interest later. If the reply of the first node makes it back to the other node, we again run into the problem of the other node not receiving anything new. This makes the convergence slow as multiple nodes receive this sync Data carrying no new elements and become silent for a while. We could add more random jitters in places to minimize this problem, but this is not reliable 100 percent of the time and convergence is flaky as topology size increases. This also makes the protocol more complicated, more non-deterministic and even slower in some cases.

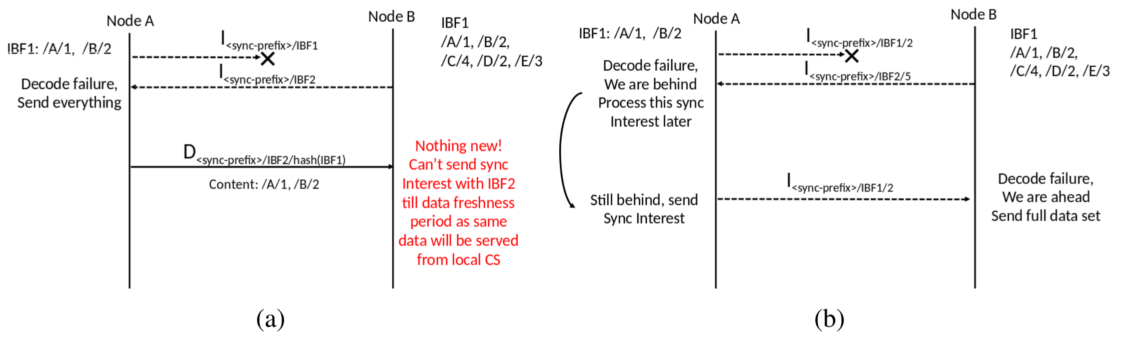

Node A
Node B
IBF1: /A/1, /B/2
IBF1 /A/1, /B/2, /C/4, /D/2, /E/3
I<sync-prefix>/IBF1
I<sync-prefix>/IBF2
Decode failure, Send everything
D<sync-prefix>/IBF2/hash(IBF1)
Content: /A/1, /B/2
Nothing new! Can't send sync Interest with IBF2 till data freshness period as same data will be served from local CS
(a)
Node A
Node B
IBF1: /A/1, /B/2
IBF1 /A/1, /B/2, /C/4, /D/2, /E/3
I<sync-prefix>/IBF1/2
I<sync-prefix>/IBF2/5
Decode failure, We are behind Process this sync Interest later
Still behind, send Sync Interest
I<sync-prefix>/IBF1/2
Decode failure, We are ahead Send full data set
(b)

Figure 16. (a) Problem with sending data if a node is behind (b) Fixing the problem through number of elements

5.5 Adding every prefix and sequence number to the IBF and not erasing anything from it

Currently, the future hash problem is solved by a simple heuristic that fails in case the sequence number of other side has been updated by more than one. In decode failure, the heuristic cannot be applied. This happens because the state of the IBF in PSync only represents the current state of a node. It does not have knowledge about the previous states because we specifically remove them from it. In ChronoSync, this problem does not occur as only those nodes who recognize the digest answer. Nodes in ChronoSync maintain historical digest to quickly determine if they recognize it.

For PSync to solve this problem, it must only insert the hashes of /prefix/seq-numbers into the IBF. This way the IBF of a node not only represents the current state, but all the past states too. After this change, future hash problem is impossible to occur. So the solution of sending the sync Data immediately if our number of elements are greater works wonderfully without anyone sending data that has lower sequence numbers. The solution is illustrated in Figure 17. solving the problem shown in Figure 13. The solution is effectively equal to the **arbitrary** names solution where any names can be inserted into the IBF without the structure of **/**<**user-prefix**>**/**<**seq-no**>.

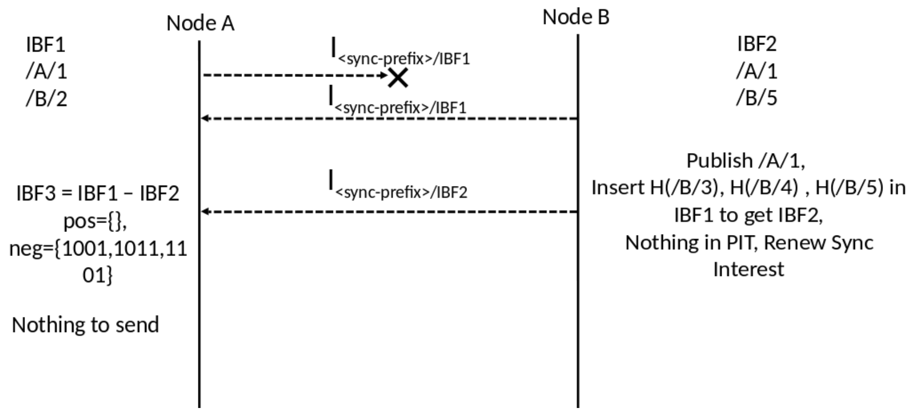


Figure 17. PSync no longer has future hash problem

Note that this solution does not affect the decoding failure probability of the difference between the two IBFs which is still proportional to the number of elements changed between them and not to the number of elements inserted into the IBFs. The downside of this solution is that the IBF can reach its limits of number of elements much faster. In the current implementation, the IBF uses a 32-bit integer to store the count. In the future this could be increased to 64 bits to avoid blowing up the IBF for a long time. Another downside is that the decode failures might increase.

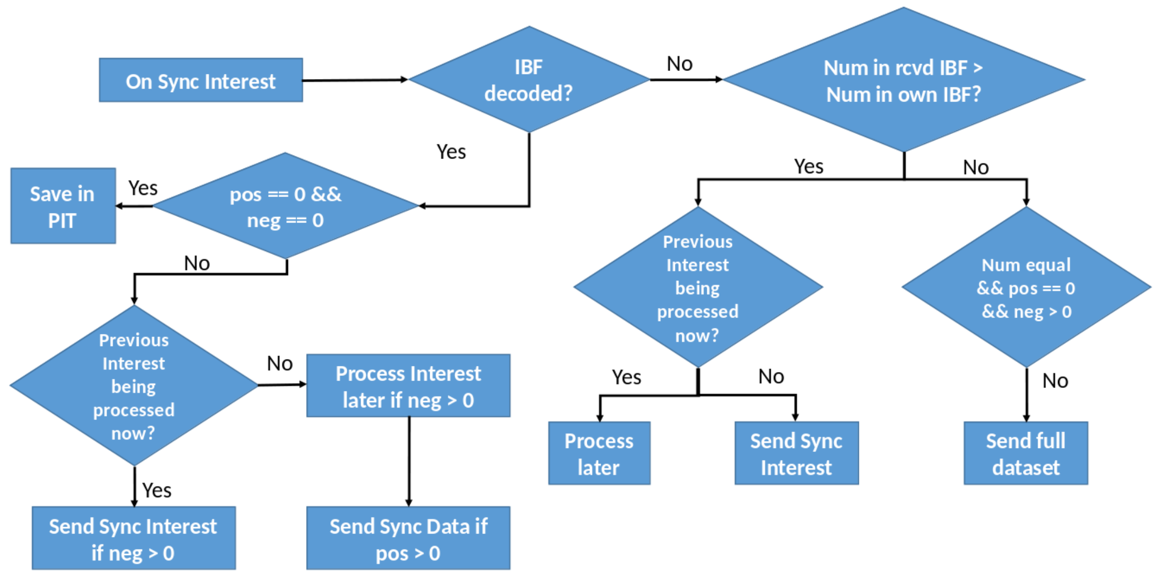
5.6 Reacting to incoming sync Interest reliably

We can now reliably respond to incoming sync Interest on decode failure as we are possessed with the correct knowledge of the number of elements in the IBF and devoid of future hashes. If the decode of the IBF in incoming sync Interest is successful, but the node is behind it should process this sync Interest later in the hopes of recovering from incoming sync Data. If the node also discovers that it is ahead it should send the sync Data now. When the node processes the sync Interest later and it is still behind, then it should send a sync Interest immediately but not send any sync Data if the positive set was not empty as it has already been sent the last time we processed this Interest. Similar case should happen on decode failure if the number of elements in the received IBF is greater than the number of elements the node itself has. But in the case of decode failure we do not send all of our dataset but wait to hear from the other side first.

If a decode failure happens and the number of elements in our IBF is more than or equal to that of the received IBF, then the node should send the entire dataset. But in case the number of elements in our IBF is equal to the number of elements in the received IBF, even after the decode failure there could be some elements that show up in the negative set. In that case we should not send our entire dataset and wait for incoming sync Data to get us up to speed.

Finally, we should only save those sync Interest in the pending Interest table for which the decode is successful and our IBF is equal to the received IBF i.e the positive and negative sets are empty.

Figure 11. is updated as Figure 18. to reflect these changes. The number of elements in the IBF added to the sync Interest along with inserting every prefix/seq into IBF guides the recovery efficiently in case of a decode failure.



On Sync Interest
IBF decoded?
No
Num in rcvd IBF > Num in own IBF?
Yes
Save in PIT
Yes
pos == 0 && neg == 0
Yes
No
No
Previous Interest being processed now?
Num equal && pos == 0 && neg > 0
Previous Interest being processed now?
No
Process Interest later if neg > 0
Yes
No
No
Process later
Send Sync Interest
Send full dataset
Yes
Send Sync Interest if neg > 0
Send Sync Data if pos > 0

Figure 18. Solution 1: PSync on Sync Interest

5.7 Waiting before sending a new sync Interest after sync Data is received

To reduce the number of sync Interest, every time a sync Data is received we cancel the existing scheduled sync Interest and schedule a new one after a small jitter. Figure 19. shows this change along with the change to insert all sequences leading up to the new sequence into the IBF. Although a sync Data that comes with no new updates is no longer expected, we keep the behavior of sending sync Interest after a while in case it does happen through some obscure scenario.

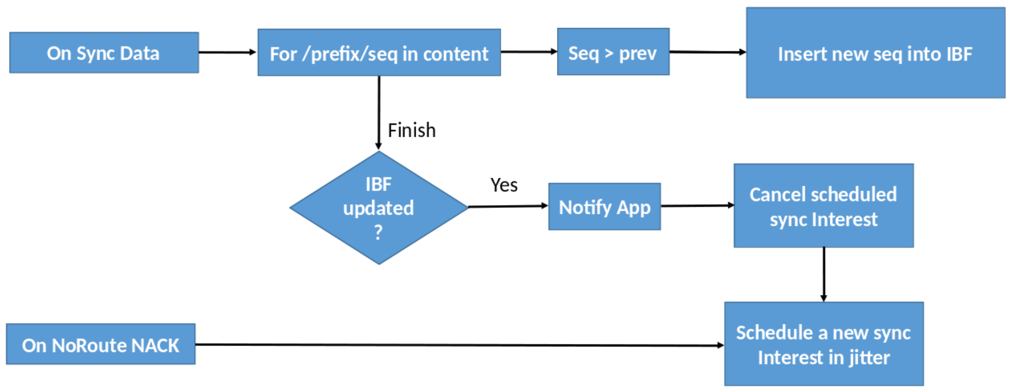

On Sync Data
For /prefix/seq in content
Seq > prev
Insert new seq into IBF
Finish
IBF updated ?
Yes
Notify App
Cancel scheduled sync Interest
Schedule a new sync Interest in jitter
On NoRoute NACK

Figure 19. Solution 1: PSync on Sync Data

Chapter 6

Solution II : Latest sequence with cumulative number of elements

Inserting everything into the IBF means keeping track of all the names in the network. This defeats the purpose of having to keep track of only the latest sequence numbers of a name by each node. We would like to keep the latest sequence numbers and still not send stale data. The problem with keeping track of the number of elements in IBF is that it leads to stale data as described in Section 5.4 But we can solve this problem by counting the number of total elements that had been inserted into the IBF and not just the number of elements an IBF currently has. This metric is enough to avoid sending stale data successfully in most cases when there is a decode failure. In cases it does send stale data when there are future hashes we set the sync data freshness very low so that it does not stop convergence by blocking sync Interests. Hence the following changes are made on top of solution I to support this use case:

- Only insert the latest sequence in IBF by removing the older sequence
- Keep count of the number of elements ever inserted into the IBF and append that to the sync Interest
- If sending full dataset set the freshness of the data to 10 milliseconds
- Add future hash checking back to on Sync Interest when decode is successful before sending data back

The on sync Interest flow diagram remains the same as Figure 18. except that we also remove any future hashes when sending back data on positive detection after successful decode as shown in Figure 20. The on sync Data flow diagram is reverted to remove the older sequence as shown in Figure 21.

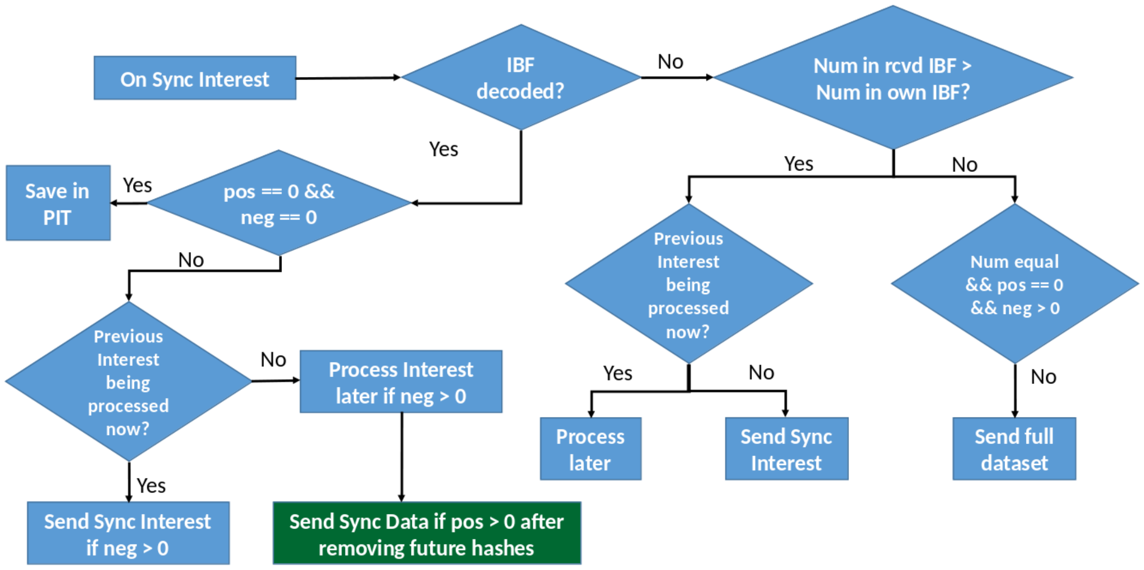

On Sync Interest
IBF decoded?
No
Num in rcvd IBF > Num in own IBF?
Yes
Save in PIT
Yes
pos == 0 && neg == 0
Yes
No
No
Previous Interest being processed now?
Num equal && pos == 0 && neg > 0
Previous Interest being processed now?
No
Process Interest later if neg > 0
Yes
No
No
Process later
Send Sync Interest
Send full dataset
Yes
Send Sync Interest if neg > 0
Send Sync Data if pos > 0 after removing future hashes

Figure 20. Solution II: PSync on Sync Interest

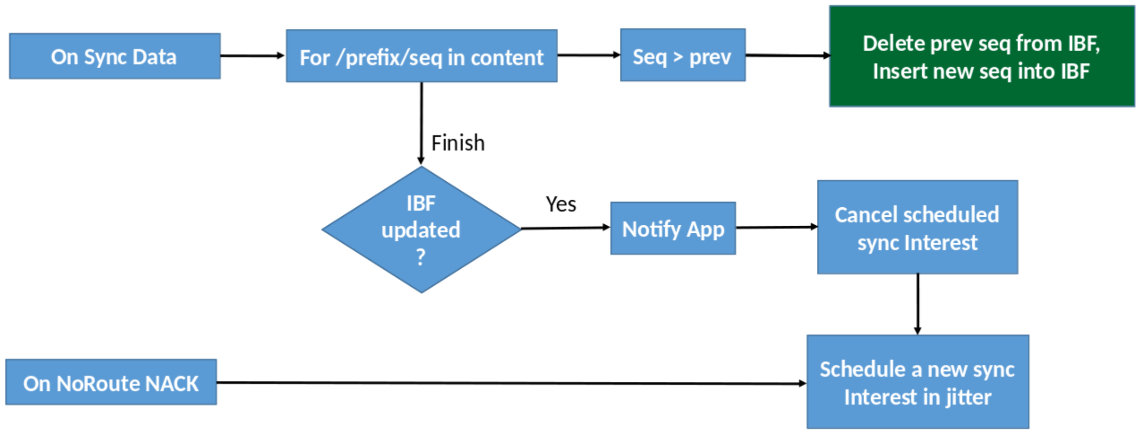

On Sync Data
For /prefix/seq in content
Seq > prev
Delete prev seq from IBF, Insert new seq into IBF
Finish
IBF updated ?
Yes
Notify App
Cancel scheduled sync Interest
On NoRoute NACK
Schedule a new sync Interest in jitter

Figure 21. Solution II: PSync on Sync Data

Chapter 7

Solution III : Alternate PSync design

Building upon the lessons learned in the solutions before a new design for PSync was explored. The aim of the design was to avoid situations where nodes do not have a pending sync Interest to answer when they publish and reduce the delay and the number of packets. In this design, we still only insert the latest sequence number but do not answer sync Interests immediately. Instead, each node has a periodic timer to answer pending sync Interests. We also send a sync Interest immediately after receiving sync Data instead of waiting unlike solution I and solution II. This is because other nodes will be collecting the sync Interests and answer when the periodic timer expires. The on Sync Interest flow diagram is updated as Figure 22.

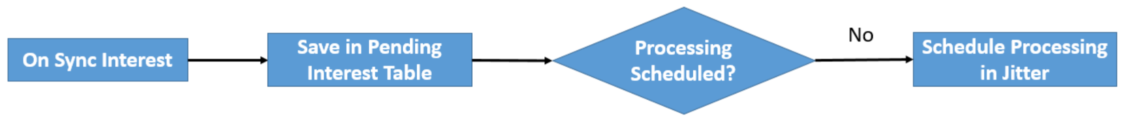


Figure 22. Sol III: PSync on Sync Interest

When the processing timer expires, all sync Interests are processed as shown in Figure 23. This builds upon the solution presented in Solution I and II only difference being the processing is done in a separate function, called satisfyPendingSyncInterests. On Sync data remains the same as solution II, Figure 21.

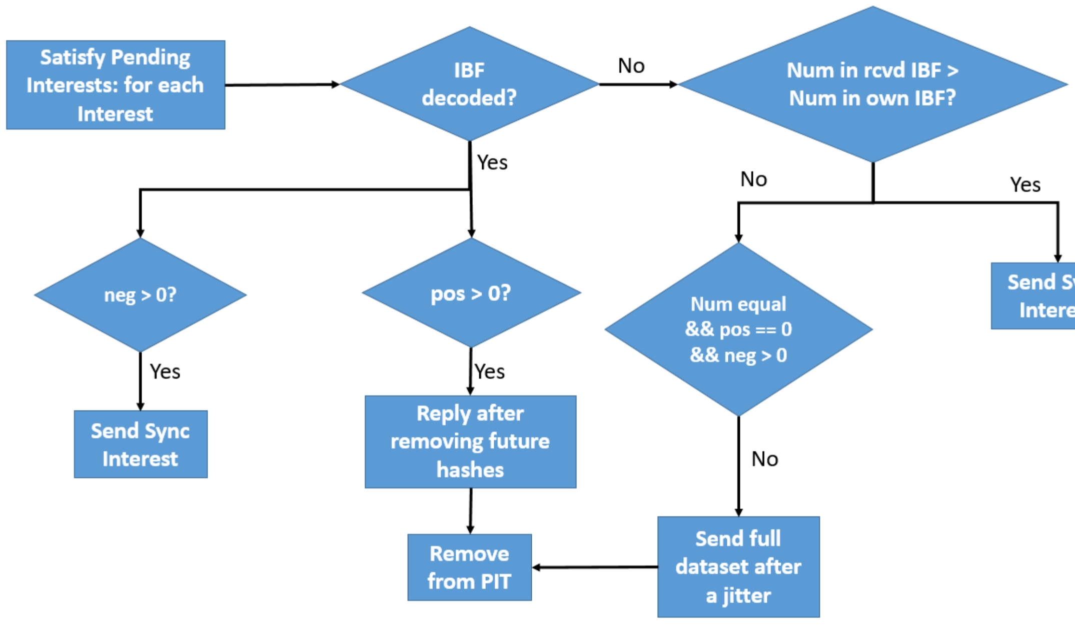

Satisfy Pending Interests: for each Interest
IBF decoded?
No
Num in rcvd IBF > Num in own IBF?
Yes
No
Yes
neg > 0?
pos > 0?
Num equal && pos == 0 && neg > 0
Send Sync Interest
Yes
Yes
Send Sync Interest
Reply after removing future hashes
No
Remove from PIT
Send full dataset after a jitter

Figure 23. Sol III: PSync Satisfy Pending Sync Interests

Chapter 8

Evaluation

NLSR is an NDN routing protocol the design of which requires the services of a sync protocol. NLSR needs to synchronize its link-state database in link-state (LS) mode and its hyperbolic coordinates in hyperbolic routing (HR) mode. NLSR provides the option to the user to choose the sync protocol at run time via its configuration file. Sync Interest lifetime is 60 seconds by default as not many updates are expected. NLSR is deployed on the NDN testbed and provides the important function of routing Names to the testbed users. It is important that the performance of the sync protocol is optimized to allow NLSR to quickly synchronize its state between instances so that it can calculate the routing table and install the routes in the FIB.

When NLSR starts up, it queries the Face dataset of the neighbors specified in its configuration file from NFD. Once it has a list of the neighbors associated with a Face in NFD, NLSR can register prefixes on the Face. NLSR registers the name of its neighbors to send Hello Interests to determine whether they are alive. NLSR also registers the sync prefix to all its neighbors and sets the Multicast Strategy in NFD for the sync prefix.

8.1 Changes to NLSR startup

Currently, the synchronization module within NLSR is instantiated before the prefixes are registered and even before the Face dataset is received from NLSR. Hello Interests are also being sent before registration. Hence we fix this behavior to send Hello Interests after routes are registered and register the routes for sync to a neighbor after the neighbor's Hello Data is validated. We also added a function to PSync that lets NLSR send the first sync Interest when it wants to after sync route registration and not when the sync is initialized.

If Interests are sent with no routes installed, Multicast Strategy returns a NoRoute NACK and deletes the PIT entry. Hence this change stops these NACKs from occurring.

8.2 Changes to ChronoSync

Even after the change above, if some neighbors start late, NoRoute NACK can occur from their NFD instead of local NFD and hence the sync protocol must deal with. We modified ChronoSync to send a sync Interest after jitter if NoRoute NACK is received. This change has been merged into the ChronoSync master branch.

8.3 Changes to PSync

Unlike PSync, ChronoSync does not Segment the sync Data content but compresses it. To avoid the segmentation for as long as possible, we also changed PSync to compress the data. We use zlib as opposed to ChronoSync's use of bzip2 as we had found zlib more effective in compressing the IBF before.

8.4 Mini-NDN experiment setup

Mini-NDN [11] is an NDN network emulator based on Mininet [12]. It was based on a legacy code base that did not allow easy changes. Some of the pain points of it were:

- Every time a Mini-NDN experiment was updated it was needed to be installed in the system
- Every time Mini-NDN itself was updated, an install to the system was needed
- Many aspects of the experiment was not directly controlled by it, but by Mini-NDN core

Mini-NDN was redesigned to overcome these shortcomings. Further Mini-NDN was known to not scale beyond 100 nodes easily, even on big machines. This was because Mini-NDN uses real instances of NFD and NLSR with security. To disable security we took a patch from ndnSIM [13] that makes ndn-cxx library [14] (which is used by NFD, NLSR, ChronoSync, and PSync) use a dummy key by default and puts a precomputed dummy signature on packets to scale beyond 100 nodes. On smaller node topologies we saw faster emulations as security is not used, interaction with file system where keys are stored is eliminated. We also use a RAM disk to store our logs so that the speed of file I/O during emulation is much faster than disk.

In Mini-NDN experiments, the nodes are connected via virtual Ethernet pairs and we use UDP faces between nodes. Sync data, especially in PSync, can go over the default Ethernet MTU size of 1500 bytes which results in the fragmentation of UDP packets. Tshark (command-line version of Wireshark) is used to collect the data and then later processed to de-fragment it. This is time consuming and sometimes not reliable. So we increased the default MTU size to be 9000 to circumvent this problem. We also process the Tshark pcap files with ndn-tool's [15] ndndump instead of the tshark ndn.lua script because ndndump is written in C++ and yielded faster results than ndn.lua.

8.5 Experiments

Topologies are chosen from various old and current testbed topologies as well as potential future topologies. We chose the topologies listed in 1. to run NLSR under various conditions and check the speed and overhead of PSync and ChronoSync. Only the partition experiments uses a different topology to do network partition.

Table 1. Topologies used for NLSR

Topology	Links	Description
10 node	24	Snapshot of NDN testbed
22 node	50	Snapshot of NDN testbed
42 node	117	Current NDN testbed
100 node	294	Proposed topology for testbed
200 node	790	Proposed topology for testbed

For PSync IBF of size 9 is used as it shows the lowest overhead across experiments. Anything higher increases the packet size and anything lower increases the number of packets. IBF9 means the IBF is set with a size of 9 and is expected to successfully decode a difference with a received IBF as long as they only differ by up to 6 elements. Each experiment is repeated thrice and the results are averaged for each scenario.

The following abbreviations are used in presenting the results of various experiments:

Table 2. Abbreviations used in results

Abbreviation	Description
chrono	ChronoSync
sync-arb	PSync Solution I (arbitrary, inserting all names in IBF)
psync-lat	PSync Solution II (inserting only the latest sequence in IBF)
psync-alt	PSync Solution III (alternative design)

8.5.1 Start Up Experiment

Let NLSR start up on each node with a gap of 100 milliseconds between each startup and run the experiment for 60 seconds after all NLSRs have started. Then check whether NLSR has converged before stopping the experiment. After that, we calculate how much time does it takes for each prefix to be received by all the nodes in the network since it was published. For each publish only those nodes are considered to have received the update which were already started at the time of publishing. We calculate the minimum, median, and maximum over this data. Additionally, we calculate the overhead in terms of number and size of Interest and Data packets.

Figure 24. shows the median delay of ChronoSync vs various designs of PSync. Table 3. lists the minimum, median, and maximum delays for all the protocols with various topologies. All the designs of PSync performed much better than ChronoSync in terms of delay. PSync's solution II and III have the smallest delay, followed by solution I as compared to ChronoSync. As more frequent decode failures are expected in putting everything into IBF, the larger delay of solution I is expected.

Table 3. Delays for Normal Startup

Protocol	ChronoSync (sec)			PSync-arbitrary (sec)			PSync-latest-seq (sec)			PSync-alternate (sec)		
Topology	Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
10-node	0.015	1.24	7.46	0.016	0.85	2.80	0.015	0.59	2.77	0.023	0.65	2.93
22-node	0.007	2.04	8.39	0.004	1.23	6.73	0.003	1.01	6.02	0.049	0.83	7.08
42-node	0.012	3.15	11.89	0.003	1.75	8.6	0.003	1.56	8.23	0.014	1.51	9.07
100-node	0.03	5.52	36.65	0.026	3.29	35.87	0.027	2.88	35.13	0.034	2.82	36.55
200-node	0.029	6.47	73.19	0.025	4.30	69.84	0.025	3.40	70.89	0.030	3.29	71.55

In the normal startup, although solution II and III outperform ChronoSync in terms

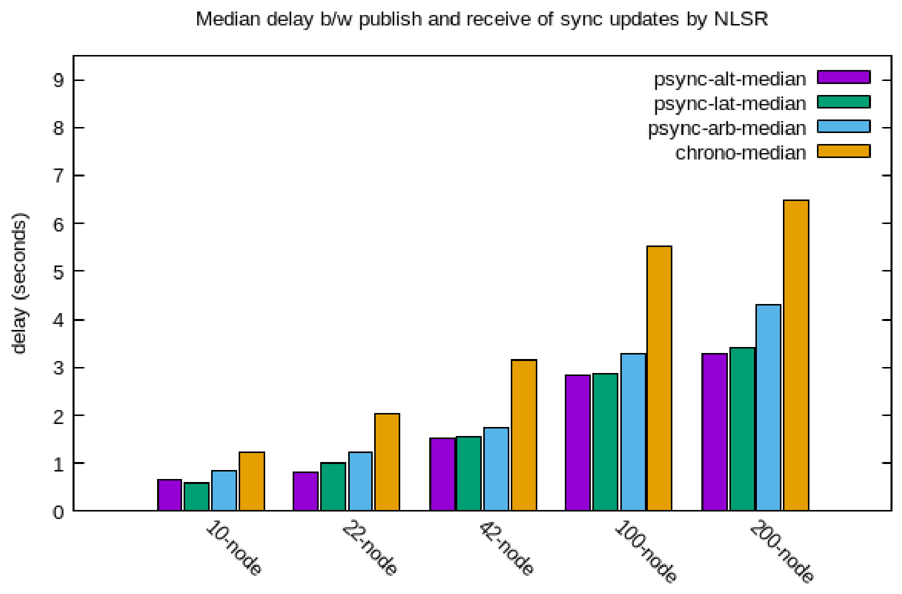


Figure 24. Median Delay in Normal Startup

of delay, their overhead for number of Interest and Data is much higher than ChronoSync's as topology size increases. Solution I's overhead is much closer that of ChronoSync as the topology size increases because it does not suffer from the future hash problem. Solution I has 1.34 times the total number of packets of ChronoSync for 200 nodes, followed by solution II with 1.64 times and solution III with 1.94 times. For the total bytes of 200 node, solution I uses 1.56 times more bytes than ChronoSync, followed by 1.69 times for solution II and 1.83 times for solution III. For topologies up to 42 nodes, the overhead is comparable for all.

Table 4. Number of Packets for Normal Startup Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	192.80	106.30	151.40	112.80	149.39	110.30	181.90	117.60
22 node	281.36	148.05	208.96	158.77	225.91	170.86	290.68	195.82
42 node	520.21	272.88	430.10	342.12	443.48	346.83	563.62	392.81
100 node	733.14	388.95	745.67	606.11	861.09	681.87	1046.39	719.31
200 node	1467.88	880.16	1708.92	1440.65	2130.09	1728.23	2604.05	1948.37

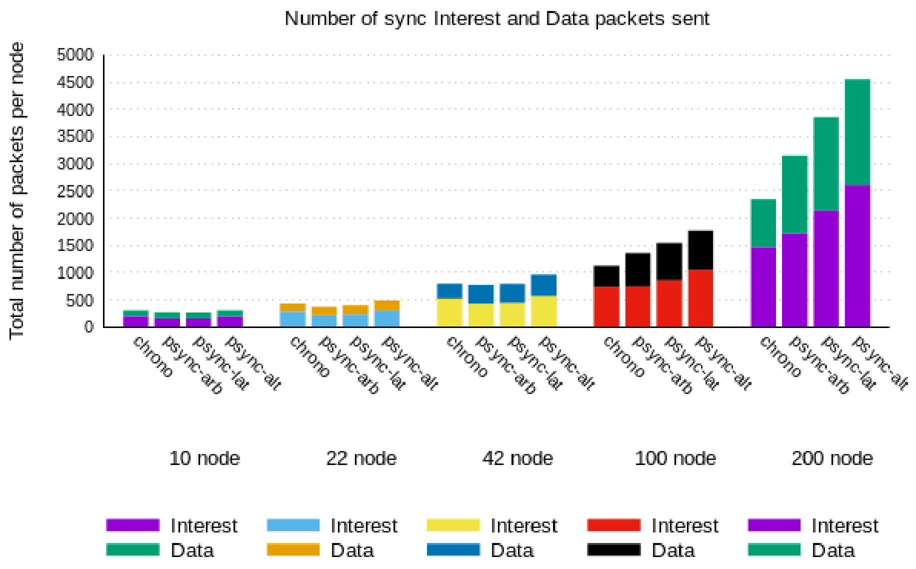


Figure 25. Total Number of Packets Per Node in Normal Startup

Table 5. Number of Bytes for Normal Startup Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	16619.8	60223.8	123941.0	62491.0	23660.4	62273.7	28656.6	66197.9
22 node	23526.1	89712.5	33389.7	93205.3	36140.8	99856.3	46507	113923
42 node	44402.4	199282.0	69565.5	211436.0	71677.1	217705.0	91063.2	236207
100 node	63500.4	342071.0	120753.0	465367.0	139468.0	519128.0	169312	540621
200 node	126491.0	937873.0	279754.0	1386540.0	348583.0	1448750.0	426155	1602390

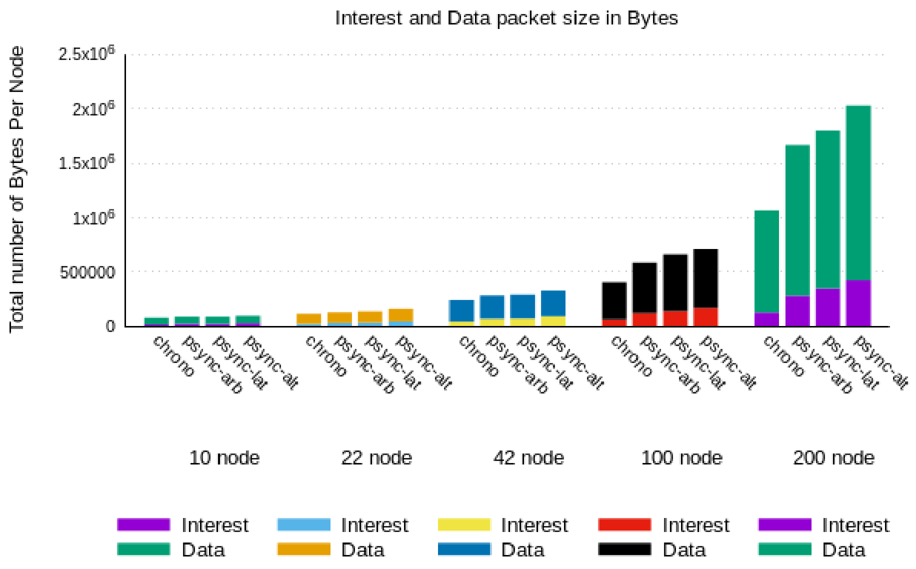


Figure 26. Total Number of Bytes Per Node in Normal Startup

8.5.2 Delayed Startup Experiments

Delayed startup experiment is identical to normal startup except that one node starts 60 seconds after all the nodes have been started. The experiment results and conclusion are same as those of normal startup experiment above.

Table 6. Delays for Delayed Startup

Protocol	ChronoSync (sec)			PSync-arbitrary (sec)			PSync-latest-seq (sec)			PSync-alternate (sec)		
Topology	Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
10-node	0.024	1.54	6.07	0.013	0.84	3.05	0.014	0.62	2.91	0.044	0.79	2.53
22-node	0.004	2.23	7.84	0.003	1.28	6.0	0.003	1.15	6.46	0.046	0.9	6.01
42-node	0.009	3.0	11.91	0.002	1.79	8.67	0.003	1.53	8.33	0.034	1.52	8.88
100-node	0.029	5.39	36.39	0.028	3.33	36.33	0.028	2.88	34.68	0.032	2.82	35.64
200-node	0.028	6.76	70.12	0.025	4.09	70.44	0.025	3.3	69.56	0.030	3.38	72.29

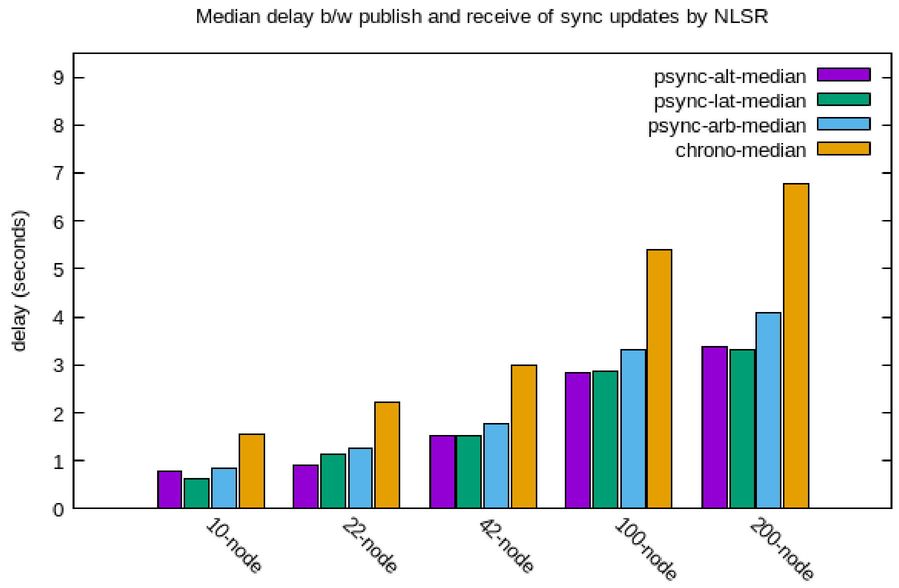


Figure 27. Median Delay in Delayed Startup

Table 7. Number of Packets for Delayed Startup Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	222.10	115.00	167.60	111.20	168.30	111.90	197.70	112.60
22 node	285.45	131.14	239.41	168.59	245.23	172.41	301.41	192.82
42 node	522.69	268.41	444.86	334.79	470.26	352.26	576.93	385.79
100 node	743.82	386.87	773.07	607.12	883.34	680.85	1145.68	780.37
200 node	1507.41	895.09	1721.66	1431.49	2105.40	1685.84	2617.46	1933.33

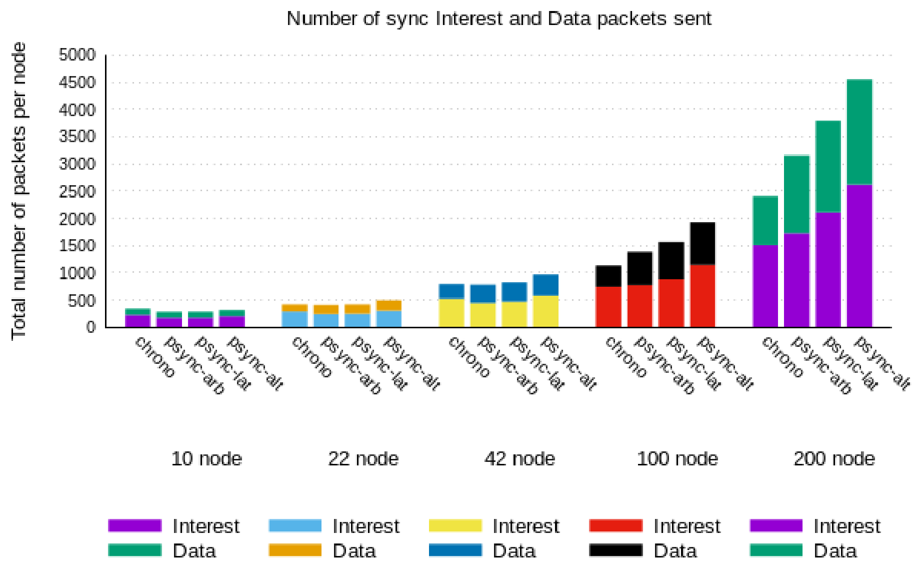


Figure 28. Total Number of Packets Per Node in Delayed Startup

Table 8. Number of Bytes for Delayed Startup Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	19075.0	65274.1	26592.9	61733.1	26667.3	63271.4	31302.2	63487.6
22 node	24717.6	86368.1	38296.0	97377.1	39274.8	101207.0	48263.4	112173.0
42 node	45080.3	194689.0	71975.6	207983.0	76115.7	219708.0	93151.6	233530.0
100 node	64289.5	337951.0	125289.0	473885.0	143169.0	508606.0	185516.0	569191.0
200 node	129685.0	933026.0	281846.0	1403300.0	344622.0	1456480.0	428247.0	1579090.0

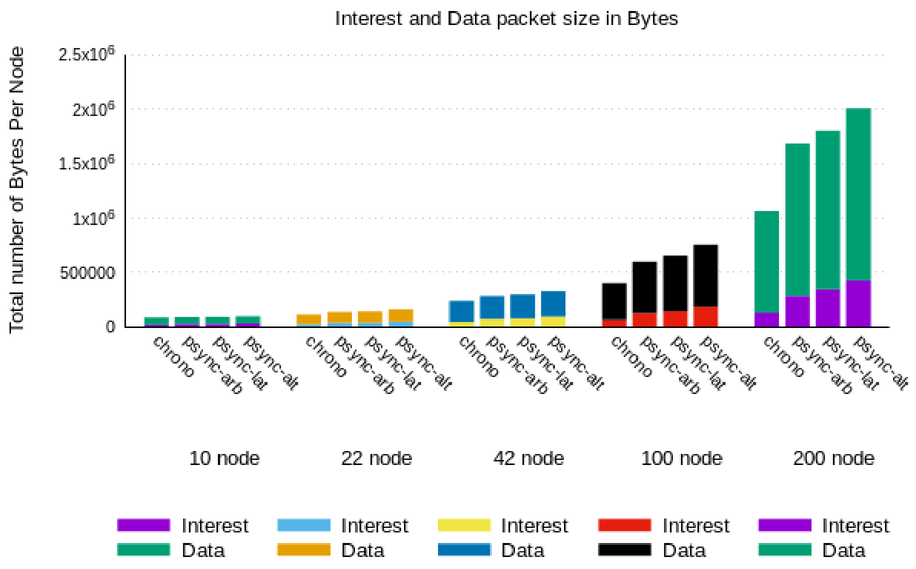


Figure 29. Total Number of Bytes Per Node in Delayed Startup

8.5.3 MCN Failure Recovery Experiments

In this experiment we let all the nodes of the topology start similar to the normal startup. After 60 seconds we then fail the Most Connected Node (MCN) for 60 seconds. Then we start the MCN node again. Finally, we look at the delays between the MCN node publishing its new sync update after recovery and other nodes receiving it. NLSR during recovery increments its sequence numbers by 10 and forces it on sync. Hence we see that solution II and solution III do not perform well in this scenario as future hash detection of sequence greater than one fails. And nodes which are behind think they are ahead and send stale data to other nodes. On the other hand, since solution I has the older sequence number in the IBF and not just the newest it does not show up as false positive and the performance is same as ChronoSync.

Table 9. Delays for MCN Experiment

Protocol	ChronoSync (sec)			PSync-arbitrary (sec)			PSync-latest-seq (sec)			PSync-alternate (sec)		
Topology	Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
10-node	0.016	1.04	3.39	0.016	1.32	2.64	0.803	1.74	18.17	0.231	1.5	2.77
22-node	0.019	1.28	4.50	0.019	1.33	2.69	1.315	2.67	20.86	1.392	2.51	13.26
42-node	0.005	1.35	5.39	0.004	1.37	2.74	0.681	2.65	32.17	1.125	2.81	23.34
100-node	0.03	1.49	6.09	0.027	1.66	3.32	1.244	2.98	48.72	1.132	3.07	43.54
200-node	0.04	1.71	3.47	0.034	1.46	2.88	1.113	2.61	52.16	1.195	3.26	57.3

The overhead for solution II and solution III is also much higher as nodes are spreading old information to the network and it takes many rounds for the network to converge on the new sequence. The overhead for the solution I is slightly lower than that of ChronoSync's in terms of the number of packets but slightly higher in terms of total bytes.

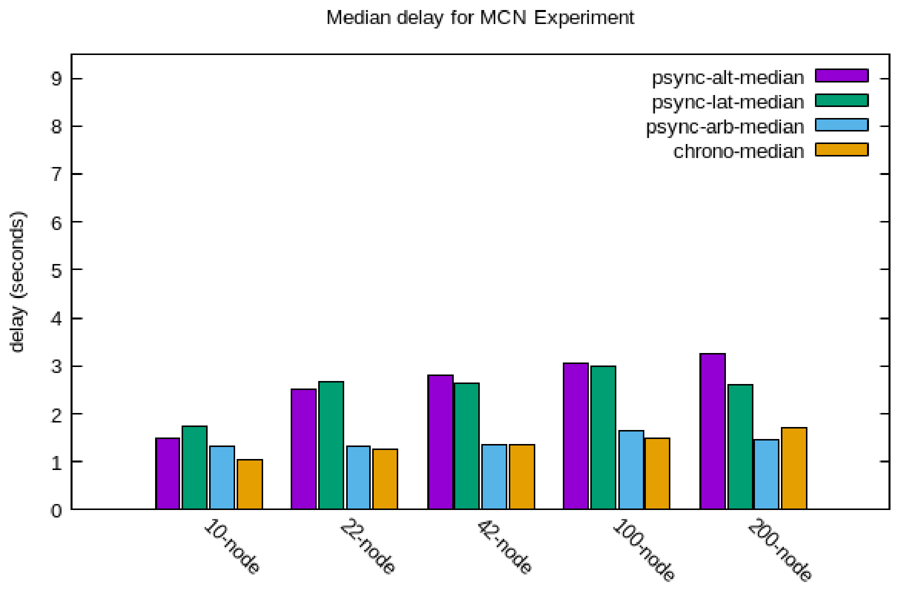


Figure 30. Median Delay for MCN Experiment

Table 10. Number of Packets for MCN Experiment Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	38.60	14.60	29.70	12.70	34.5	16.5	46.20	25.10
22 node	36.72	14.04	28.31	13.72	400.64	376.86	236.68	204.77
42 node	44.85	18.07	33.90	17.92	491.14	465.91	408.59	369.38
100 node	47.36	18.73	36.12	19.44	519.82	490.23	435.82	390.34
200 node	62.55	29.10	61.15	27.52	788.76	752.24	655.39	601.11

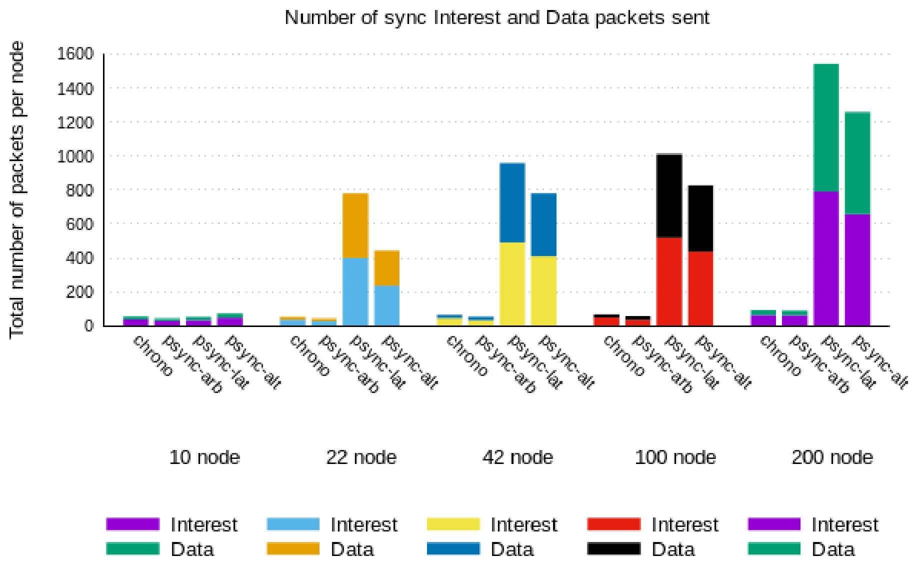


Figure 31. Total Number of Packets Per Node in MCN Experiment

Table 11. Number of Bytes for MCN Experiment Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
10 node	3291.1	8085.0	4770.9	7022.5	7402.8	14130.7	5513.7	9112.9
22 node	3138.7	9416.0	4555.7	7669.8	38498.1	116584.0	65112.1	214307.0
42 node	3834.3	14717.9	5489.8	10008.1	66243.2	208947.0	79653.8	263522.0
100 node	4049.5	17791.3	5826.3	10822.1	71340.4	221342.0	85027.2	277421.0
200 node	5223.9	17475.9	10134.9	15399.8	107076.0	340389.0	129574.0	425642.0

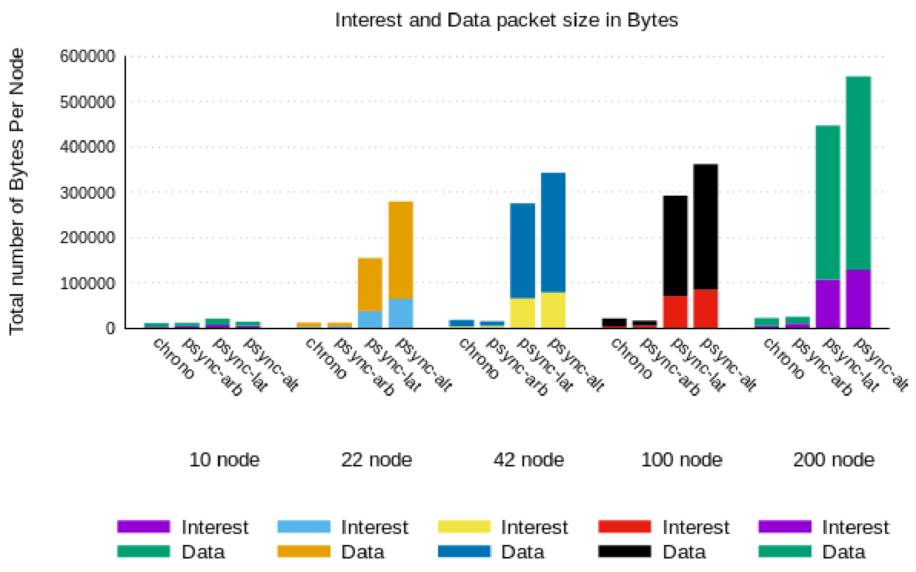


Figure 32. Total Number of Bytes Per Node in MCN Experiment

8.5.4 Network Partition Experiment

In network partition experiment we generate a random topology of 100 node (named as 100 node* to differentiate from 100 node in the above experiments) which has two clusters connected by a single link. Each cluster had 25 percent of the nodes forming links with other nodes in the cluster. We start the topology such that the single link is down. We let the experiments run for 60 seconds so that the two clusters converge and then bring up the single link. We then look at the delays and overhead after 60 seconds are given to the network to recover from the partition.

Solution II and III are best in terms of delays, followed by solution I. In terms of overhead, the solution I is the closest to ChronoSync's overhead. It is followed by solution II and then solution III. This is similar to results obtained in startup experiment and delayed startup experiment.

Table 12. Delays for Partition Experiment

Protocol	ChronoSync (sec)			PSync-arbitrary (sec)			PSync-latest-seq (sec)			PSync-alternate (sec)		
Topology	Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
100-node*	0.022	9.94	104.76	0.013	7.84	105.67	0.014	6.83	105.6	0.035	6.87	105.91

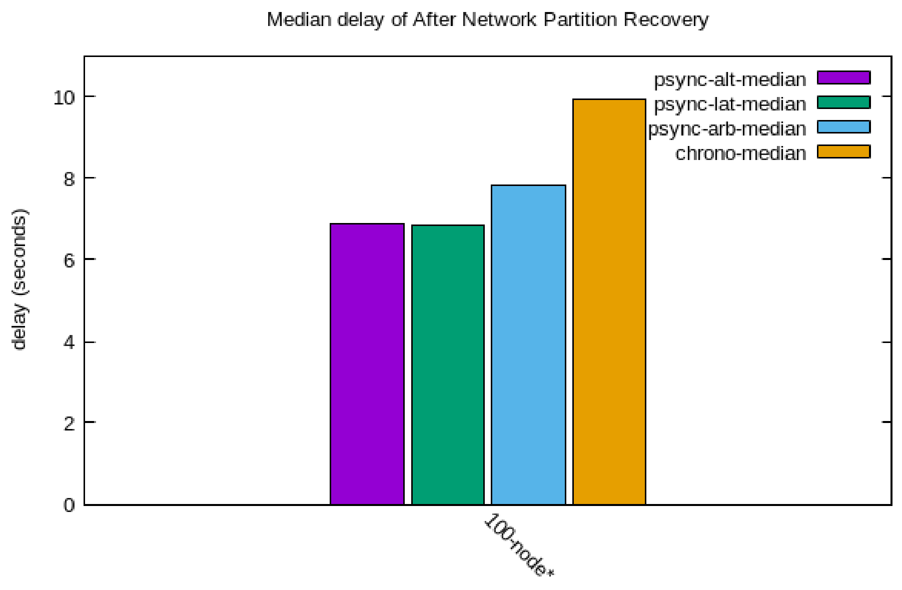


Figure 33. Median delay for Partition

Table 13. Number of Packets for Partition Experiment Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
100 node*	791.17	364.10	736.29	598.61	894.33	732.13	1054.87	751.25

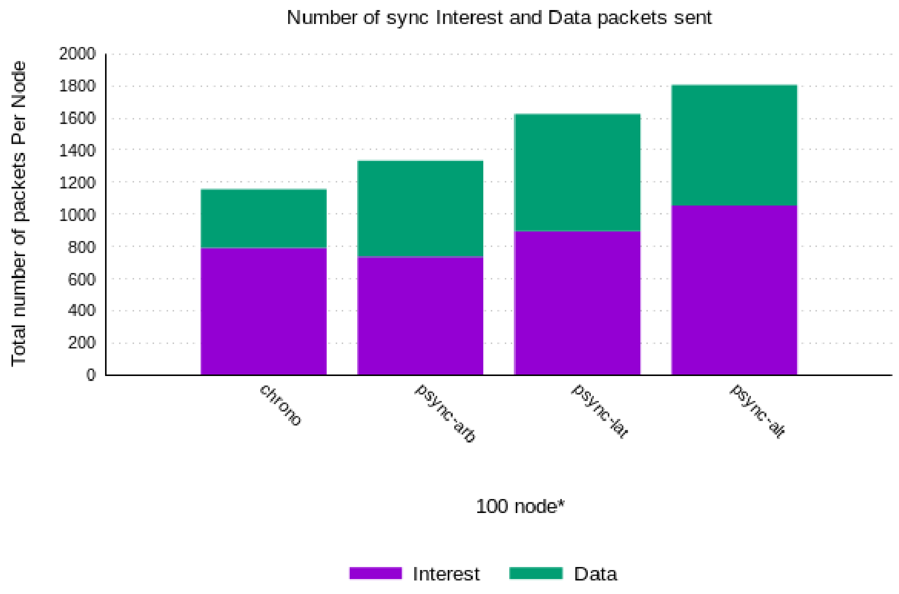


Figure 34. Total Number of Packets Per Node in MCN Experiment

Table 14. Number of Bytes for Partition Experiment Per Node

Protocol	ChronoSync		PSync-arbitrary		PSync-latest-seq		PSync-alternative	
Topology	Interest	Data	Interest	Data	Interest	Data	Interest	Data
100 node*	68675.78	248235.77	119145.19	377593.07	144776.05	449001.41	170629.76	450904.37

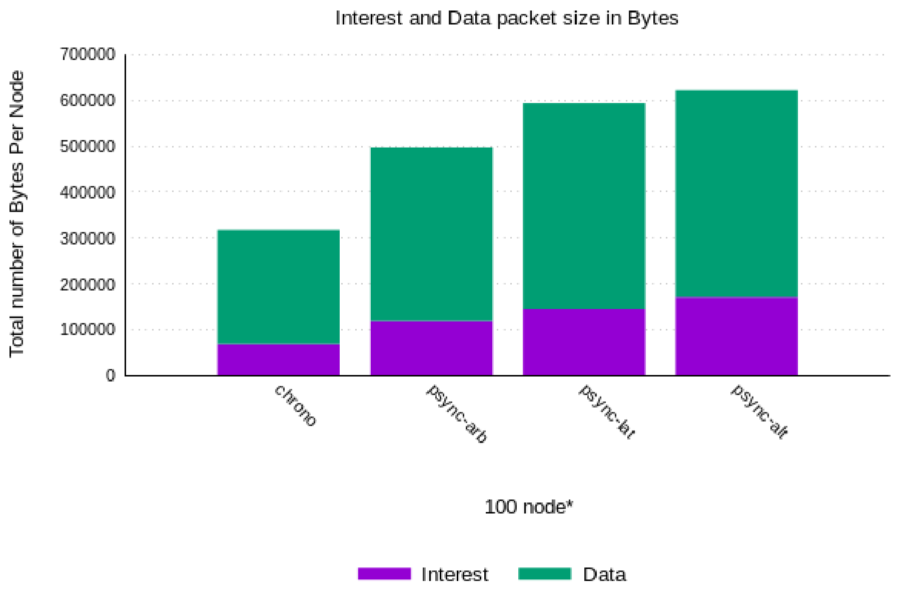


Figure 35. Total Number of Bytes Per Node in Partition Experiment

Chapter 9

Conclusion

We suggested changes to PSync to solve slow convergence and high overhead problems. We made PSync converge with a small IBF size of 9 for a wide variety of scenarios which was not possible before even for simple cases. All three solutions are able to achieve this with different trade-offs. Solution I is the most balance case which gives a better median delay than ChronoSync while being the closest to ChronoSync's overhead in all the experiments. It is especially better in MCN experiment than Solution II and Solution III due to avoidance of the future hash problem. Solution II and Solution III are able to provide better delays in all the cases other than MCN but it comes at a higher overhead cost. On the other hand, solution I also keeps track of all the names and hence will increase the state of a node. Solution II and Solution III only keep track of the latest sequence number and will have minimum state, whereas Solution I's state will continue to grow.

REFERENCES

[1] A. Afanasyev, T. Refaei, L. Wang, and L. Zhang, “A Brief Introduction to Named Data Networking,” in *IEEE MILCOM 2018*.

[2] M. Zhang, V. Lehman, and L. Wang, “Scalable name-based data synchronization for named data networking,” in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[3] M. T. Goodrich and M. Mitzenmacher, “Invertible bloom lookup tables,” in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2011, pp. 792–799.

[4] “Ndn packet format specification 0.3 documentation,” https://named-data.net/doc/NDN-packet-spec/current/index.html, (Accessed on 10/27/2019).

[5] “Nfd developers guide,” http://named-data.gitlab.io/TR-NDN-0021-NFD-dev-guide/ndn-0021-nfd-guide.pdf, (Accessed on 10/28/2019).

[6] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, “A survey of distributed dataset synchronization in named data networking,” *NDN, Technical Report NDN-0053*, 2017.

[7] V. Lehman, A. M. Hoque, Y. Yu, L. Wang, B. Zhang, and L. Zhang, “A secure link state routing protocol for ndn,” *Tech. Rep. NDN-0037*, 2016.

[8] “Ndn testbed status,” http://ndndemo.arl.wustl.edu/, (Accessed on 10/28/2019).

[9] Z. Zhu and A. Afanasyev, “Let’s chronosync: Decentralized dataset state synchronization in named data networking,” in *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2013, pp. 1–10.

[10] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, “What’s the difference?: efficient set reconciliation without prior context,” *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 218–229, 2011.

[11] NDN Project Team, “Mini-NDN GitHub,” https://github.com/named-data/mini-ndn, (Accessed on 10/29/2019).

[12] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[13] S. Mastorakis, A. Afanasyev, and L. Zhang, “On the evolution of ndnSIM: an open-source simulator for NDN experimentation,” *ACM Computer Communication Review*, July 2017.

[14] NDN Project Team, “ndn-cxx: NDN C++ library with eXperimental eXtensions,” https://github.com/named-data/ndn-cxx, (Accessed on 10/29/2019).

[15] ——, “NDN Essential Tools,” https://github.com/named-data/ndn-tools, (Accessed on 10/29/2019).