RepoSync: Combined Action-Based and Data-Based Synchronization Model in Named Data Network

Weiqi Shi* and Alexander Afanasyev†
*Electronic Engineering Department, Tsinghua University
†Computer Science Department, University of California, Los Angeles

***Abstract*—Named Data Networking (NDN) is proposed to embrace the increasing demands of data-sharing and content dissemination. This paper proposes a distributed synchronization model, RepoSync, to provide accurate and efficient content synchronization. RepoSync adopts a combination of action-based and data-based synchronization, where actions are used to record the modifications of datasets, so as to instruct the sync process. Moreover, snapshot is introduced as data-based synchronization to provide the copy of dataset, handling the synchronization of large scale modifications. Simulations reveal that RepoSync can provide a promising performance of synchronization under various network conditions.**

I. Introduction

As the demand of content sharing is increasing rapidly, datasets maintained by multiple users need to be synchronized periodically. Current most wildly used synchronization applications, like Dropbox and Google Docs, maintain a client-server paradigm, where files should be first uploaded to a centralized server and then distributed to the users. This centralized pattern can easily manage the modification to share files and solves the conflicts, but may suffer from the single-point failure and other performance issues. The alternative peer-to-peer designs [1] also contribute to the content sharing, such as BitTorrent Sync service [2]. However, the peer-to-peer solutions have the drawbacks of tremendous redundant data and heavily depend on the overlay network. Besides, they also suffer from limited resilience to network failure.

Due to the development of Named Data Networking (NDN), the distributed pattern of synchronization can be realized in a simpler but more efficient way. Several distributed synchronization protocols, including ChronoSync [3] and CCNx Sync [4] have been proposed. However as network level protocols, they do not take data removal into consideration. FileSync/NDN [5] uses snapshot to make a copy of whole dataset and exchanges it directly among users for synchronization. However, the drawback of snapshot is its inefficiency of handling small number of modifications.

To solve the problems mentioned above, a distributed sync protocol RepoSync is proposed, which combines *action-based* and *data-based* synchronization. Action-based mechanism is designed to synchronize each single modification for dataset, while data-based mechanism is for large-scale modifications. Actions are introduced to record the modification, including who changes datasets and what types of changes it makes. If one user makes a modification to its dataset, an action will be generated. Action should be propagated to other users so that they can adopt the same modification. Data-based synchronization is defined as using snapshot to synchronize datasets directly. Different from existing works, where snapshot is directly used to sync the copy of datasets, our snapshot serves as the back-up of actions and only is used for synchronizing large-scale modifications. Action-based synchronization can provide efficient and accurate data-sharing for each single modification, but may generate huge overhead when large-scale modification is needed and the whole process while be delayed. Data-based synchronization can efficiently handle the large-scale modifications but need time to reconstruct the dataset. Therefore we combine these two paradigms, taking both of their advantages. RepoSync adopts the synchronization pattern of ChronoShare [6], [7], which is a distributed synchronization application based on NDN. In particular, the contributions of RepoSync can be concluded as follows:

1) RepoSync introduces action-based synchronization, which provides clear trace of dataset modification and instruct other users how to make the same modifications.
2) RepoSync employs data-based synchronization, using snapshot to efficiently sync large number of modifications and serve as the back-up of actions.
3) RepoSync combines the action-based and data-based synchronization and takes fully advantages, while avoids drawbacks of two independent modes.
4) RepoSync also supports efficient data removal during synchronization process.

The rest of paper is organized as follow. Section II introduces the NDN architecture. Section III describes the details of RepoSync design. Section IV and V demonstrate the evaluation and discussion respectively and Section VI concludes the paper.

II. Background

In this section, NDN and repo-ng are briefly introduced as the background of RepoSync, and their features allow us to easily build the distributed synchronization application.

A. NDN Architecture

Named Data Networking (NDN) [8] is a data-oriented network architecture. NDN changes the narrow waist layer, focusing on the data instead of host. Two fundamental components, interest packet and data packet, are introduced to fetch the data with a given name. An interest is sent when there is a request for a certain piece of data. The interest includes the name of target data and other necessary information, such as selectors. It will go through the network until reaches a node which can provide the data it needs and the data will be traced back to the requester. The satisfied data should be signed to prove its integrity and intactness. Hierarchical namespace is used to distinguish the unique data packet.

When an interest reaches a router, it will be forwarded to a registered interface based on the forwarding interest base (FIB) maintained by the router. To record which interface the interest comes from, each router also maintains a pending interest table (PIT). Router will add an entry in the PIT once an interest arrives. When the interest is satisfied, the expected data can be traced back to the requester with the help of PIT. Along the way back to the requester, data are cached in content store (CS)

for same request in the future. The pull based model of NDN provides the basis for the distributed sync model.

B. repo-ng

RepoSync is designed for repo-ng [9], which is a set of storage application over NDN. Compared with Content Store, where data packets in network are temporary stored in cache, repo-ng provides persistent storage for data object. The unit of storage is data object and the management scope is based on NDN name prefix. The purpose of RepoSync is to make sure every user (repo-ng) in the same network maintains exactly the same dataset. In repo-ng, data is immutable and there is no consistency problem for the content.

III. RepoSync Design

A. Overview

RepoSync combines action-based and data-based synchronization. For action-based synchronization, each operation that changes the dataset, including data insertion and deletion, can generate an independent action. Each action includes following components: creator name, sequence number, data name and action type. Creator name is used to specify the creator (user) who generated this action. It is assumed that each user maintains a unique name. Sequence number is used to distinguish the actions generated from a given user. The sequence number increases with the new generation of actions. Actions can be uniquely represented by the combination of creator name and sequence number. For simplicity, we refer *action status* to the combination of user's creator name and its own lastest sequence number. Data name is used to denote the data which has been modified and the action type reveals the type of modification (INSERTED or DELETED).

We assume that if two users apply same actions in an identical order, they will maintain same datasets. Actions are generated when operations are made to datasets(different order of insertion and deletion for same data may have different consequences). For a given user, its dataset status is based on the actions generated from all the user in the group. Therefore, it status can be represented by the root digest, which is the aggregation of every known user's *action status*, since *action status* reflects how many actions generated by a certain creator have been applied (the case of missing or out-of-ordered actions will be discussed in III-E). To detect status of other users' datasets, each user periodically sends *sync interest*, which carries its lastest root digest.

Following the idea of ChronoSync, a sync tree is used to record all the known users' action statuses. Fig. 1 provides an example of sync tree, where each node represents a digest and the leaf nodes are the combination of action status and MIN Seq[1]. Each leaf node generates a digest from its action status by using a hash function. To obtain the root digest, the leaf nodes' digests are aggregated by the same hash function in a certain order (e.g. canonical order of creator name). The sync tree should be updated whenever a user generates, receives and applies an action.

To record all the applied actions, either generated by itself or fetched from others, each user maintains an action list (AL) (Table I). Entry in AL is the pair of <action, root digest>, where the root digest is the updated one after the action is applied. AL can be used to infer whether users are synchronized. If the digest in the comming *sync interest* is same to user's own root digest, then the user and the one who sent the *sync interest* are synchronized. If the digest is same to user's previous root digest, it means the one that sent the *sync interest* maintains outdated root digest, which is probably due to action missing. If the digest cannot be recognized, in other word, the digest is not recorded in the AL, then the user itself may maintain the outdated root digest.

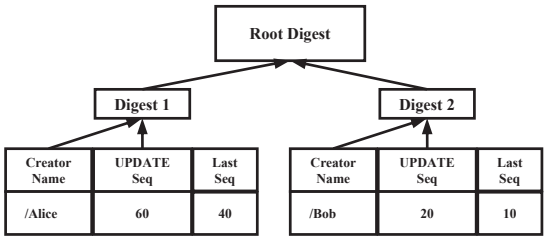

Fig. 1: An example of Sync Tree

TABLE I: An example of ActionList

Action				digest
creator name	seq	data name	type	
...	...	...	...	...
/Alice	60	/ndn/route/pic1.jpg	inserted	0010....
/Bob	20	/ndn/route/pic2.jpg	deleted	092a....
/David	50	/ndn/route/pic3.jpg	deleted	b92a....

Besides the action-based synchronization, RepoSync also adopts a data-based synchronization, where snapshot is introduced. Snapshot can be exchanged directly among users, helping users to find the difference of datasets. There are serveral reasons to introduce the snapshot. First, action-based synchronization might be inefficient to sync large-scale of modifications, since each action only record one single modification of the dataset and a huge amount of actions will increase the overhead of the network and delay the sync process. Snapshot is necessary when a new user joins in the group, since it needs to synchronize the whole dataset from others. Also, snapshot can easily handle the the conflicts after network partition since datasets are compared directly when necessary. Besides, after synchronization, actions are useless for synchronized users and they may take up a lot of memory. Snapshot can replace actions without losing the information of dataset.

The format of snapshot is illustrated in Table II. Snapshot maintains the basic info (generator and version number) used to distinguish from the snapshots generated by others. The main content of snapshot is the summary of dataset, including the data name, statuses (INSERTED or DELETED) and version numbers. INSERTED represents the data that currently exist in the dataset and DELETED represents the data that have already been removed. Version number represents how many times a specific piece of data has been modified (switch from INSERTED to DELETED or vice versa). For example, when a data packet is first inserted, its version is zero. The version increases when it is deleted and increases again when it is reinserted. Version number can solve the conflicts when data have different status in snapshot and local dataset, since higher version number implies the updated status. Only when data entries in snapshot have higher version numbers, the corresponding data in local datasets will be modified to be consistent with those in snapshot. Otherwise, the data will remain unchanged.

Besides, to make sure those users who apply the snapshot can share the same root digest with other users, action statuses

[1]MIN Seq is the minimum sequence number currently maintained by the user from corresponding creator. Previous actions generated from that user may be replaced by snapshot. Details will be discussed in III-D

TABLE II: An example of Snapshot

Snapshot Info		
generator		version
/Alice		2
Dataset		
/ndn/route/pic1.jpg	INSERTED	0
/ndn/route/pic2.jpg	DELETED	1
/ndn/route/pic3.jpg	INSERTED	2
...	...	
Sync Tree		
/Alice	60	
/Bob	20	
/David	50	
...	...	

should also be included in snapshot. In this way, once the snapshot is received, the sync tree can be updated using the action status in snapshot to recalculate root digest.

When the snapshot is generated, actions will be removed. If a request wants to fetch the action that has been removed, it will get the snapshot back. Otherwise, the action will be returned. In this way, actions and snapshots can work independently to serve for the synchronization.

B. Namespace

Since the name of each piece of data in NDN is globally unique and follows the hierarchical namespace structure, it is important to build proper naming conventions for data packets so that corresponding interests can find them conveniently. Interests can serve for four different functions in RepoSync: 1) notify others with own current root digest (*sync interest*); 2) fetch the actions or snapshots (*fetch interest*); 3) serve for recovery process (*recovery interest*) and 4) fetch the data packets (*normal interest*).

Different types of interests are distinguished by the data they are trying to fetch. Each type of interests can be used to fetch a corresponding type of data, for example, *sync interests* fetch the *sync data* and normal interests fetch the normal data packets. Interests should encode the data name so that they can find the data. To classify different types of data and corresponding interests, different naming conventions should be introduced.

For *sync interest* and *sync data*, a broadcast prefix (e.g. /ndn/broadcast) is included so that the *sync interest can* be broadcasted to all the users. It is necessary since a user should be able to propagate its root digest to others when the root digest is changed. To distinguish from other types of interest, there is a component "sync" following the broadcast prefix. The last component is the root digest of the sender, which is used for notification and comparison with other users' status.

For *fetch interest* and *fetch data*, its namespace follows [10]. A component "fetch" is also included for demultiplexing. The last component specifies the name of required actions, which is uniquely represented by the combination of creator name and its corresponding sequence number.

For *recovery interest* and *recovery data*, the name is similar to the *sync interest*, but the middle component is replaced by "recovery" for discrimination. Besides, the unrecognized digest should be included in the name. Users will only send *recovery interests* when it receives a *sync interest* with unrecognized digest, and only those users who recognize this digest can handle this *recovery interest* (details will be introduced in next subsection). Therefore, the digest in the last component of the name is the received unrecognized digest.

TABLE III: Example of different types of name

Type	Example name
sync	/ndn/broadcast/sync/b92a...
fetch	/ndn/route/fetch/David/50
recovery	/ndn/broadcast/recovery/912s...
normal	/ndn/route/pic3.jpg

For normal interest and data, the specific name of the data have already been settled based on the content, and only a routable prefix should be added in order that interests can find the users who maintain the data.

C. Synchronization Process

To detect others' root digest, *sync interest* with current root digest is issued and broadcasted to the whole network periodically. The *sync interest* will not bring back the *sync data* if root digest is the same to other users' root digest, since no actions are generated and synchronization is done.

Once new action is generated, the *sync interest* will be satisfied since root digest is recalculated and newer than the root digest in *sync interest*. *Sync data*, which records the creator name and sequence number of all the missing actions, will be returned back. Once the *sync data* is received, *fetch interest* will be issued to fetch the missing actions using the information in *sync data*. After the actions are fetched back, they can be applied to the datasets. For example, if the action type is INSERTED with the data name /ndn/route/pic3.jpg, then normal interest with this name should be issued to fetch the content. If action type is DELETED with name /ndn/route/pic1.jpg, the data with the same name in dataset should be removed.

The relationship between the root digest in *sync interest* (sync root digest) sent by the sender and that maintained by the receiver (local root digest) reveals whether the sender and receiver are synchronized. There are three scenarios:

If the two digests are same, synchronization process is done. In other word, if all the nodes are synchronized in the network, there should be no different root digest.

If the sync root digest is outdated (same to a previous local root digest), it means at least the sender misses actions. *Sync data* will be generated and returned back to the sender, providing the information of the missing actions.

If the digest is unrecognized (not same to any previous local root digests), then it is likely that the receiver misses some actions. This case often happens in network partition, where different groups modify datasets separately and generate their own actions without communicating with other groups. To obtain the missing actions, *recovery interest* is issued by the receiver. Carrying the unrecognized digest, the *recovery interest* is broadcasted to whole network. It should be processed by the users who can recognize this digest, since they are likely to maintain the most updated datasets. They will return the *recovery data* with action statuses of all their known users. Once receiving the *recovery data*, the receiver can infer what actions are missing and then start to fetch them.

It is worth discussing why *sync data* and *recovery data* do not encode all the missing actions directly. Encoding each action into a unique data packet keeps the integrity of an action and makes it easy to fetch separately. It is necessary especially when data packet is mutable. Although current repo-ng only support immutable data, RepoSync can still be applied in other shared folder where data packet is mutable. If data

are mutable, the modification of content should be paid more attention. RepoSync can easily solve this problem by encoding the information of content modification in the *fetch data*. When the *fetch data* is received by others, it can instruct them how to modify the content.

```
switch Interest Type do
    case Sync Interest
        if sync root digest = local root digest then
            store the sync interest
        else
            if sync root digest is recognizable then
                return sync data (clue of missing actions)
            else
                send recovery interest
            end
        end
    case Fetch Interest
        if have the requested action then
            return the action
        end
    case Recovery Interest
        if sync root digest is recognizable then
            return recovery data(status of all users)
        end
    case Normal Interest
        if have the requested content then
            return the requested content
        end
    endsw
endsw
```

Algorithm 1: Process Interest Packet

D. Snapshot

Snapshot is introduced to maintain the summary of the datasets and replace the actions. It can also speed up synchronization of large-scale modifications since the content of dataset can be compared directly.

Snapshot should be generated periodically after the whole network has been synchronized for a certain time (denote as T). The generation of snapshot means that no new actions are generated and previous actions have not been fetched within time T. Therefore, these actions should be removed to reduce the burden of memory. When a user generates a snapshot and removes all its actions, current action status of all its known users should be recorded (since creator names are not changed, only sequence number should be recorded by MIN Seq). MIN Seq represents the minimum sequence number currently maintained by the user from corresponding creator. When a snapshot is generated, MIN Seq is updated by the lastest sequence number. MIN Seq is necessary, since it can distinguish those actions that have been removed from the new generated actions.

Snapshot is not propagated when it is generated. Instead, it is only used when *fetch interests* are trying to fetch the removed actions. Once snapshot is received, the user should reconstruct its own dataset based on the snapshot. If the data in snapshot share the same status with corresponding data in dataset, then nothing should be done. Otherwise, version number should be compared to avoid conflicts. For a certain piece of data, lager version number implies more updated status. Therefore, for a give piece of data, if its version number in dataset is larger than that in snapshot, then the dataset should remain unchanged. Otherwise, the data status should be changed to synchronize with snapshot. Besides, the version number can also be used to handle the conflict when network partition happens and different snapshots are generated. When network partition heals, multiple groups may generate their own snapshots. Those different snapshots can be exchanged and datasets can be compared directly.

E. Exceptions and Conflicts

It can be noticed that the chronological order of applying actions is the basic assumption for RepoSync. Applying actions in different orders may lead to different datasets. There are two types of out-of-order actions:

1) Actions generated from different users are out of order
2) Actions generated from same user are out of order.

The scenario 1) may happen when actions are generated simultaneously from different users and are fetched in different order, and scenario 2) may exist due to the packet loss or different link delay. In the following, we will prove that 1) can be rescued by recovery process and 2) can be prevented by the sliding window and fast retransmission mechanism.

For 1), applying actions in different order may generate different root digests, which makes root digest in *sync interest* unrecognizable even if two users maintain same actions. In this case, the recovery process will be triggered. Once *recovery data* is received, the user will find that the action statuses recorded in the *recovery data* are same to those maintained in its own sync tree. Then it can assume this unrecognizable sync root digest is caused by out-of-order actions. It will be recorded to avoid recovery process being triggered again. For datasets, only when the out-of-order actions related to same data, datasets are not synchronized. Therefore, to solve the difference, the action with the largest creator name (canonical order) will be adopted.

For 2), the sliding window and fast retransmission mechanism are introduced to make sure the actions generated from a given creator can be fetched in order. Each user maintains several independent sliding windows, each of which is used to fetch the action stream generated by one known user. The window size S determines how many *fetch interests* can be issued at the same time. If the action is received in order, a new *fetch interest* will be issued. If not, then the out-of-order actions will wait until the missing actions arrive. If certain actions are lost and cannot be returned within a certain time (e.g. the average round trip time), the corresponding *fetch interests* will be reissued immediately instead of waiting for interest timeout (normally is much larger than round trip time). Therefore, this fast retransmission can reduce the delay caused by packet loss significantly.

IV. Evaluation

This section evaluates the performances of RepoSync by conducting simulations using NS3[2] with ndnSim [11] model, which is used to simulate multiple NDN scenarios architecture.

A. Synchronization Delay

Efficiency is important for a synchronization application especially when it is used in real-time system and communication. Therefore, in this section, we evaluate the synchronization delay of RepoSync under various network conditions. The delay is

[2]ns-3: a discrete-event network simulator for Internet systems:http://www.nsnam.org

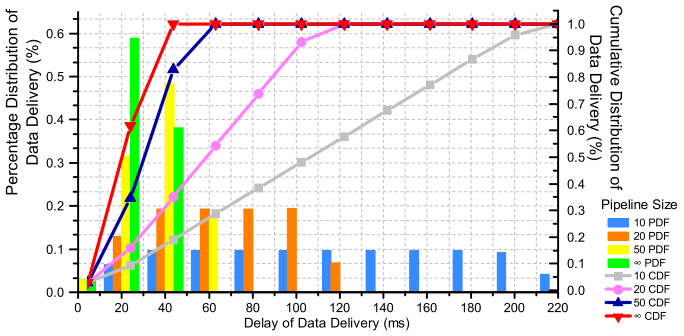

Fig. 2: PDF and CDF of data delivery delay in face of congestion conditions

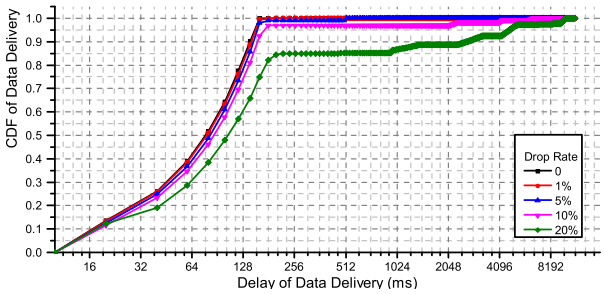

Fig. 3: CDF of data delivery delay in face of drop rates

measured from data generation to the data propagation to all users.

We use a typical topology [12], containing 34 users and 78 links. Each link was assigned measurement-inferred delay and 100 Mbps bandwidth. We run each scenario 20 times with 1000 data inserted. In each run, the data to be synchronized are distributed in different users with different distributions.

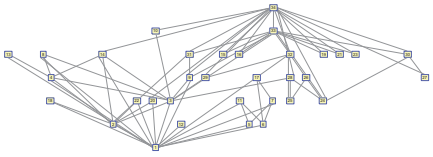

Fig. 4: Simulation topology

1) Performance with various congestion condition: Firstly, we evaluate synchronization delay under different congestion condition. It is important since network conditions vary heavily in reality. We can use different window sizes to represent different congestion conditions, since in the situation with less congestion, more data packets can be transmitted successfully at the same time. In Fig. 2, PDF reveals the percent of 1000 original data that has been synced and CDF shows the trend of whole synchronization process. It shows that the congestion control works well since the percentage of delivery is distributed uniformly. Under the low congestions, the delay of whole synchronization is pretty small (around 80ms). However, with congestion increasing, the total time of synchronization process increases as well.

2) Performance with packet drop: Next, we conduct the simulation in the network environment with various per-link drop rate since in packet loss could cause huge trouble for synchronization. The cumulated distribution is used to reveal the percentage of data that have been synchronized. Fig. 3 shows the simulation result. In the case with low drop rate (1%-5%), curves almost overlap with the curve with no drop, which suggests that the synchronization process is almost

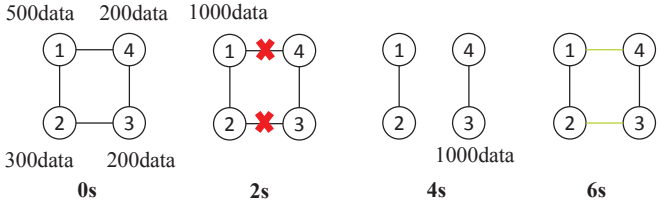

Fig. 5: Simple Topology with Link Failure

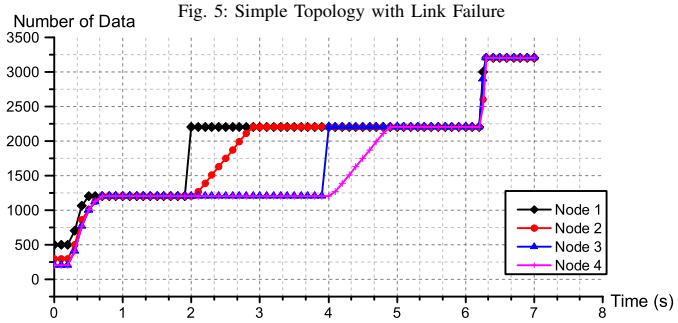

Fig. 6: Recovery Process of Simple Topology

unaffected by the packet drop. The performance of RepoSync stays unaffected because fast retransmission mechanism (III-E) can detect the packet loss in a short time and reissue the interest. In the case with high drop rate (over 10%), the impact of packet drop cannot be avoided since it may happen in retransmission. However, the sync process can still finish within tolerable time.

B. Recovery

Recovery process is a key feature of RepoSync, since it is used to solve issues related to various network failure. Recovery process makes sure that even in network partition happens, the synchronization process can still work correctly. Therefore in this section, two simulation scenarios are conducted to show the performance of recovery process.

1) Simple Scenario: We first use a small topology with 4 users to reveal how recovery process works. Fig. 4 shows the whole simulation process, where data are injected in to different users at different time.

The result is shown in Fig.6. It can be seen that after links fail, two groups can work separately and get synchronized. After the links recover, the two groups are connected and the data inserted separately during the link failure are synchronized between two groups. The performance proves that RepoSync can work well in face of network failure.

2) Complex topology and Snapshot: The previous topology with 34 users is used to show the recovery performance in complex topology. It is assumed that all the 34 users are synchronized, maintaining 1000 same data in initial state. At 20ms, 20 new users join in the network, each of which has 100 different data. They are randomly connected to the original 34 users.

In the Fig.7 the CDF reveals the recovery process when 20 users join in the synced group. We compare the recovery process with and without generating snapshot (simply using actions). It is obvious that snapshot can speed up the process significantly. By using the snapshot, each user only needs to exchange snapshots directly instead of sending multiple *fetch interests* to fetch the actions, which also reduces the overhead of whole network.

C. Deletion

Another important feature of RepoSync is its action design, which can provide the clear trace of dataset modifications and

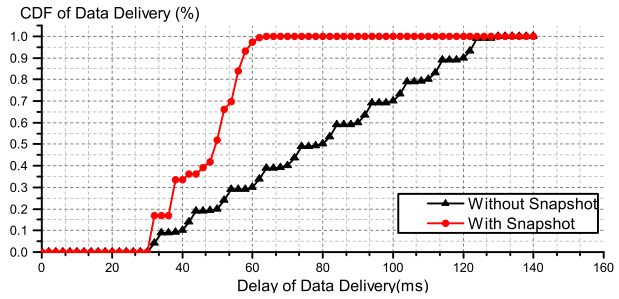

Fig. 7: Recovery Process in Complex Topology

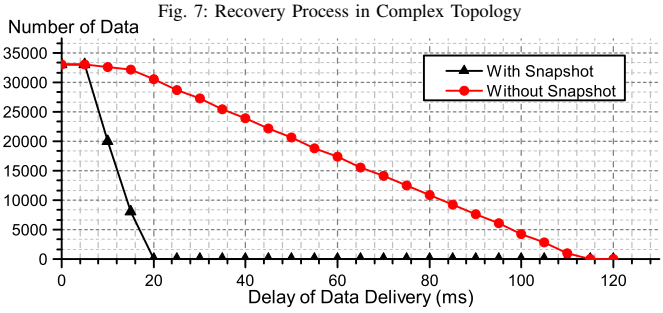

Fig. 8: Total Number of Data maintained in 34 users

support the deletion during synchronization. In this section, we evaluate the deletion performance with the same 34 users topology. All the users are synchronized with 1000 same data at the initial point. At 2s a user removes all the data and these deletion actions will be propagated to the whole system, which makes all the other users apply the same actions.

The Fig.8 shows the result, which compares the performance of deletion processed by snapshot and actions. It is obvious that both of the two methods can achieve the goal of synchronizing deletion actions, however, with the help of snapshot, the process is significantly faster.

V. Related Work

Several synchronization applications have been developed. CCNx Sync [4] and ChronoSync [3] are developed over NDN to detect and sync the differences among datasets. However as network level protocols, they do not take data removal into consideration. Because there is no mechanism to record the modifications of dataset, the removed data will be falsely regarded as the missing data and brought back. But for synchronization applications, any kinds of dataset modifications should be supported during sync process. The custodian-based information sharing [13] is proposed, where an addition content object is introduced to record the deleted data. However, it is limited in handling network partition and conflicts. FileSync/NDN [5] synchronize the copy of whole dataset directly and exchanged it among multiple users. It is efficient in the sync modifications of large scale datasets but is tedious for small editions, since snapshot should be compared with whole dataset to locate the modification.

Apart from the research based on NDN, many other sync applications were also developed, such as LBFS [14], TAPER [15] and RSYNC [16], which focus on detecting the difference of datasets. Compared with these solutions, RepoSync outperforms in its efficiency, where it only needs to use the actions or snapshot to instruct the reconciliation, instead of complex procedures. Besides, the distributed pattern of RepoSync gets rid of the single-point failure issues. Current peer-to-peer solutions [1] often suffer from the limitation of mismatch between overlay and underlay network, where frequent changes of topological connectivity (caused by users join and leave) would lead to inefficiency of sync processes.

VI. Conclusion

In this paper, we propose RepoSync, a distributed synchronization model that combines action-based and data-based synchronization. Actions are designed to record the modifications applied to the datasets and can be fetched by other users, who can follow the actions to update their own datasets. If every user shares the same knowledge about all users' action statuses and apply same actions in order, then synchronization is done. Snapshot is introduced as data-based synchronization to provide a copy of dataset, which is used to efficiently synchronize large scale of modifications. The motivation of combining actions and snapshot is to take both of their advantages: small changes of dataset can be easily fetched in actions and large scale modification can be synchronized efficiently by snapshot. Security and Privacy issues are the future work of RepoSync and will be further discussed. However, we believe the concept of distributed synchronization in RepoSync will encourage the appearance of other data-sharing applications.

References

[1] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, 2004.

[2] "Bittorrent," http://labs.bittorrent.com/experiments/sync.html.

[3] 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 2013, Göttingen, Germany, October 7-10, 2013*, 2013.

[4] "Ccnx synchronization protocol," http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html.

[5] J. B. Jared Lindblöm, Ming-Chun Huang and L. Zhang, "Filesync/ndn: Peer-to-peer file sync over named data networking," *NDN Technical Report NDN-0012*, 2013.

[6] A. Afanasyev, Z. Zhu, and L. Zhang, "The Story of ChronoShare, or How NDN Brought Distributed Secure File Sharing Back," NDN, NDN NDN-0029, February 2015.

[7] Z. Zhu, A. Afanasyev, and L. Zhang, "ChronoShare: a new perspective on effective collaborations in the future Internet," Poster, UCLA Tech Forum 2013, May 2013. [Online]. Available: http://www.engineer.ucla.edu/techforum/index.html

[8] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos *et al.*, "Named data networking (ndn) project," *Technical Report NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.

[9] "Ndn repo-ng." [Online]. Available: https://github.com/named-data/repo-ng

[10] A. Afanasyev, C. Yi, L. Wang, B. Zhang, and L. Zhang, "SNAMP: Secure namespace mapping to scale ndn forwarding," in *Proceedings of 18th IEEE Global Internet Symposium (GI 2015)*, April 2015.

[11] A. Afanasyev, I. Moiseenko, and L. Zhang, "ndnSIM: NDN simulator for NS-3," NDN, Technical Report NDN-0005, October 2012.

[12] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking Data Centers Randomly," in *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[13] V. Jacobson, R. Braynard, T. Diebert, P. Mahadevan, M. Mosko, N. H. Briggs, S. Barber, M. F. Plass, I. Solis, E. Uzun, B. Lee, M. Jang, D. Byun, D. K. Smetters, and J. D. Thornton, "Custodian-based information sharing," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 38–43, 2012.

[14] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *SOSP*, 2001, pp. 174–187.

[15] N. Jain, M. Dahlin, and R. Tewari, "TAPER: tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*, 2005.

[16] A. Tridgell and P. Mackerras, "The rsync algorithm," *TR-CS-96-05*, 1996.