Let's ChronoSync: Decentralized Dataset State Synchronization in Named Data Networking

Zhenkai Zhu and Alexander Afanasyev
University of California, Los Angeles
{zhenkai, afanasev}@cs.ucla.edu

***Abstract*—In supporting many distributed applications, such as group text messaging, file sharing, and joint editing, a basic requirement is the efficient and robust synchronization of knowledge about the dataset such as text messages, changes to the shared folder, or document edits. We propose ChronoSync protocol, which exploits the features of the Named Data Networking architecture to efficiently synchronize the state of a dataset among a distributed group of users. Using appropriate naming rules, ChronoSync summarizes the state of a dataset in a condensed cryptographic digest form and exchange it among the distributed parties. Differences of the dataset can be inferred from the digests and disseminated efficiently to all parties. With the complete and up-to-date knowledge of the dataset changes, applications can decide whether or when to fetch which pieces of the data. We implemented ChronoSync as a C++ library and developed two distributed application prototypes based on it. We show through simulations that ChronoSync is effective and efficient in synchronization dataset state, and is robust against packet losses and network partitions.**

I. Introduction

Applications such as file sharing, group text messaging, and collaborative editing are playing increasingly important roles in our lives. Many of such applications demand efficient and robust synchronization of datasets (file revisions, text messages, edit actions, etc.) among multiple parties. The research community has been working on distributed system synchronization since the early days of the Internet [1] and has produced a rich literature of solutions. Quite a few popular applications like Dropbox and Google Docs, on the other hand, are implemented based on a centralized paradigm, which generally simplifies the application designs and brings many other advantages, but also results in single points of failure and centralized control of the data. At the same time, a number of different peer-to-peer solutions [2], including the recently announced BitTorrent Sync service [3], represent another direction in the searching of efficient dataset synchronization solutions, which requires either the maintenance of a sophisticated peer-to-peer network overlay structure or critical nodes for participants rendezvous.

The recently proposed communication paradigm, Named Data Networking (NDN) [4], where data is the first-class entity and multicast of data is naturally supported, brings new opportunities to efficiently solve the problem of dataset synchronizations in a completely distributed fashion. Thus, we propose ChronoSync, an efficient and robust protocol to synchronize dataset state among multiple parties in NDN. The core idea of ChronoSync is to compactly encode the state of a dataset into a crypto digest form (e.g., SHA256), which we call the *state digest*, or digest in short, and to exchange the state digests among all parties in a synchronization group. Each party sends out a broadcast interest with their respective state digest, calculated according to their knowledge of the dataset, to all others in the group. Such interests are directly broadcasted in small networks and are broadcasted via simple overlays in large networks.[1] If the state digest carried in the incoming interest is the same as the one locally maintained, indicating identical knowledge about the dataset, no action is required from the recipient. Otherwise, one of the two actions will be triggered:

- the differences of the dataset state can be directly inferred and sent as the response to the sync interest if the state digest is the same as one of the previous local state digests;
- a state reconciliation method is used to determine the differences of the knowledge if the state digest is unknown (for example, when recovering from a network partition).

Actions will be triggered to eliminate the differences of the dataset state until the interests from all parties carry an identical state digest.

The rest of the paper is organized as follows. We first briefly introduce the NDN architecture in Section II. The main contribution of the paper is the design of the ChronoSync protocol as presented in Section III, which exploits the features of the NDN architecture to efficiently synchronize dataset state among a distributed group of users. Section IV and Section V demonstrate the implementation of ChronoSync and the evaluation results. Discussions and related work are presented in Section VI and Section VII respectively. We conclude the paper in Section VIII.

II. NDN Architecture

In this section we briefly go over a few basic concepts of the NDN architecture [4] that are essential to describe the design of ChronoSync.

NDN architecture has two basic communication units: *interest* packet and *data* packet, both carrying hierarchically structured names. An interest packet is sent when a consumer requests data. Each data packet is cryptographically signed,

[1]Note that the cost of broadcast interests in NDN is low, as identical interests are collapsed. Hence, there is at most one interest transmitted over a link in one direction regardless of the number of parties in the synchronization group.

978-1-4799-1270-4/13/$31.00 ©2013 IEEE

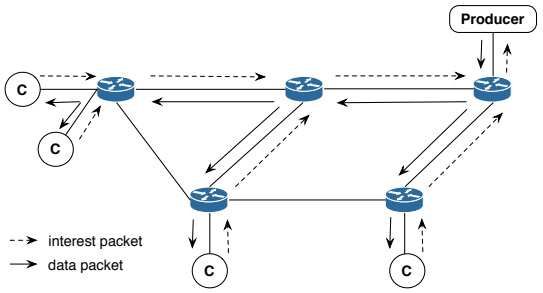


Fig. 1: Multicast of data is naturally supported in NDN

enabling the recipient to check the provenance and integrity of the data regardless of how it is obtained (from the data source, cache, or neighbor). A data packet can be used to satisfy an interest, as long as the name carried in the interest is a prefix of or identical to that of the data. An interest can also carry a selector field to specify preferences in case there are multiple data packets that can satisfy the interest.

All communication in NDN is receiver-driven, meaning that the only way for a data packet to be delivered is that a consumer explicitly sends out an interest requesting this data first. When receiving a interest, routers add an entry to the pending interest table (PIT), recording the interface from which the interest came, and use a forwarding strategy [5] to determine where to forward the interest. As a result, a returning data packet can simply trace the reverse path back to the requester. When multiple interests for the same data come from the downstreams, NDN routers create only one PIT entry, remembering all the interfaces from which the interests came, and forward out only one interest to the upstream. As shown in Fig. 1, this process essentially constructs a temporary multicast tree for each requested data item, along which the data is efficiently delivered to all requesters.

III. ChronoSync Design

In this section we describe the ChronoSync protocol design. We first present an overview of ChronoSync components, and then explain the naming rules in Section III-B. Section III-C shows how ChronoSync maintains the knowledge about the dataset and Section III-D describes how the changes to the dataset propagate to all participants. Section III-E and III-F discuss how ChronoSync handles simultaneous data generations and network partitions.

To better illustrate the basic components of the ChronoSync design, we use a group text chat application, ChronoChat, as an example throughout the paper. While a real chat application includes a number of essential components, such as roster maintenance, in our example we introduce only elements that are directly relevant to ChronoSync.

A. Overview

In the core of any ChronoSync-based application there are two interdependent components, as shown in Fig. 2: the ChronoSync module that synchronizes the state of the dataset and the application logic module that responds to the change

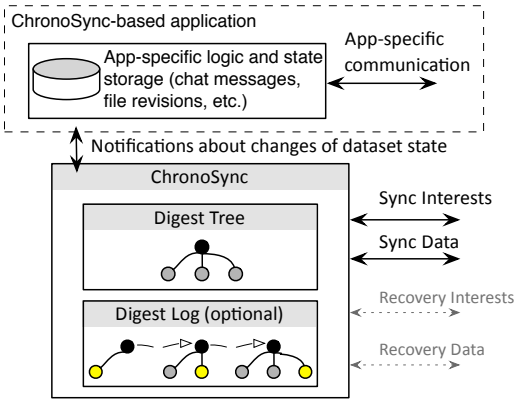


Fig. 2: ChronoSync overview

of the dataset state. In ChronoChat, ChronoSync module maintains the current user's knowledge about all the messages in the chatroom in the form of a *digest tree*, as well as history of the dataset state changes in the form of a *digest log*. After ChronoSync module discovers that there are new messages in the chatroom, it notifies ChronoChat logic module to fetch and store the messages.

To discover dataset changes, the ChronoSync module of each ChronoChat instance sends out a *sync interest*, whose name contains the state digest that is maintained at the root of the digest tree. Generally, with the help of digest tree and digest log, ChronoSync can infer dataset changes directly and reply to the sync interest with the data containing the changes, which we henceforth refer to as *sync data*. In cases of network partitioning, ChronoSync also uses *recovery interests* and *recovery data* to discover the differences in the dataset state.

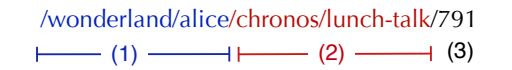
ChronoSync focuses solely on facilitating the synchronization of the knowledge about new data items in the dataset, leaving the decision on what to do after ChronoSync discovers state changes at the application's discretion. For example, the sync data in ChronoChat brings back the names of messages newly added to the chatroom, and thus a user's knowledge of the dataset is brought up to date. However, the user may decide whether to fetch all the missing messages or just the most recent ones, if the total number of missing messages is large (e.g., after recovery from a network partition).

B. Naming rules

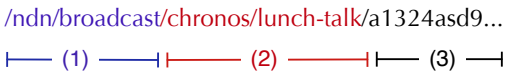
One of the most important aspects of application design in NDN is naming, as the names carry out several critical functions. The name carried in an interest packet is used by the network to figure out where to forward it and to determine which process to pass it to when it reaches the producer. Also proper naming rules can greatly simplify the design of applications.

There are two sets of naming rules in ChronoSync: one for application data names and one for sync data names.[2]

[2]The naming for recovery data will be discussed in Section III-F.



(a) An example of chat data name



(b) An example of sync data name

Fig. 3: Naming rules of ChronoSync

We design the application data names to have routable name prefixes so that the interests can be forwarded towards the producers directly. These prefixes can be constructed by appending one or more components under a prefix assigned by the Internet provider. For example, part (1) of the chat data name in Fig. 3a is such a prefix. The purpose of the part (2), which includes the application name and the chatroom name, is to demultiplex the interest once it reaches the data source: it identifies the process that is responsible for handling such interests.

The data generated by a user is named sequentially. For example, in ChronoChat, the initial message from a user to the chatroom has sequence number zero and whenever a new message is generated, be it a chat message or user presence message, the sequence number is incremented by one. As a result, the complete knowledge of the user can be compactly represented by just one name. Assume the name shown in Fig. 3a is the latest chat data name used by *Alice*. We can infer from the naming rules that *Alice* has produced 792 pieces of chat data to this chatroom, with sequence numbers ranging from 0 to 791.

Similarly, the name for sync data (Fig. 3b) also consists of three parts. Part (1) is the prefix in the broadcast namespace for a given broadcast domain. A broadcast prefix ensures that the sync interests are properly forwarded to all participants of a group, as it is often impossible to predict who will cause the next change to the dataset state. Part (2) serves the purpose of demultiplexing (similar to that of the application data name), and the last part carries the latest state digest of the interest sender.

C. Maintaining dataset state

The application dataset can be represented as the union of the subsets of data generated by all producers.

Since the knowledge of a data producer can be solely represented by its name prefix and the latest sequence number, ChronoSync tracks the latest application data name of each producer in order to maintain up-to-date knowledge of the dataset. For the sake of simplicity in writing, we refer to the latest application data name of a producer as its *producer status*.

Inspired by the idea of Merkle trees [6], ChronoSync uses a digest tree to quickly and deterministically compress knowledge about the dataset into a crypto digest, as illustrated in Fig. 4. [3] Each child node of the tree root holds a cryptographic digest calculated by applying, for example, SHA-256 hash function over a user's producer status. Recursively applying the same hash function to all child nodes of the root results in the digest that represents state of the whole dataset, which we refer to as the *state digest*. To ensure that every participant calculates the same state digest when observing the same set of producer statuses, the child nodes are kept in the lexicographic order according to their application data name prefixes.

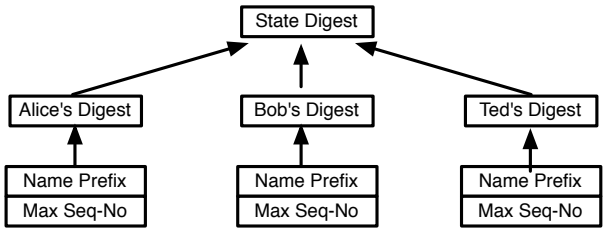


Fig. 4: An example of digest tree used in ChronoChat

State Digest	Changes
0000...	Null
9w35...	[Alice's prefix, 1]
...	...
23ab...	[Bob's prefix, 31], [Alice's prefix, 19]
05t1...	[Bob's prefix, 32]

TABLE I: An example of digest log

The digest tree is always kept up-to-date to accurately reflect the current state of the dataset. Whenever a ChronoChat user sends a new chat message or learns about the name of a new message from another participant, the corresponding branch of the digest tree is updated and the state digest is re-calculated.

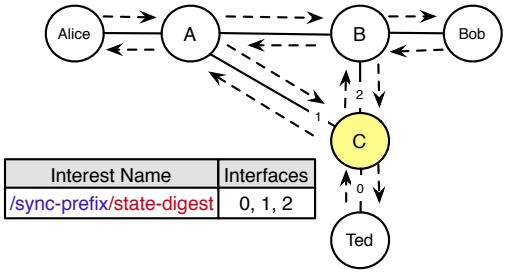
As an optimization, each party keeps a *digest log* along with the digest tree. This log is a list of key-value pairs arranged in chronological order, where the key is the state digest and the value field contains the producer statuses that caused the state change. An example of digest log is illustrated in Table I. The log is useful in recognizing outdated state digests. For example, when a user resumes from a temporary disconnection and sends out a sync interest with an outdated state digest, other parties, if recognizing the old digest, can quickly infer the differences between the dataset states and promptly reply the sender with missing data names.

Although the digest log facilitates the process of state difference discovery in many cases, it is not essential ensure the correctness of the ChronoSync design. Depending on the available resources, applications can set an upper bound on the size of the digest log, purging old items when necessary.

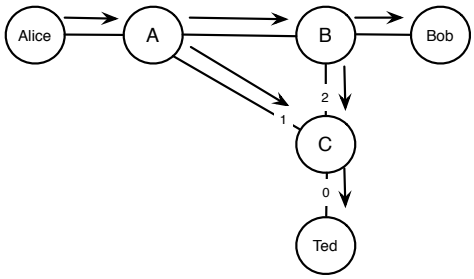
D. Propagating dataset changes

To detect dataset changes as soon as possible, every party keeps an outstanding sync interest with the current state digest. When all parties have the same knowledge about the dataset,

[3] While we use an one-level hash tree here, a more canonical form of hash trees can be used if the applications demand different naming rules.



(a) For each chatroom, at most one sync interest is transmitted over a link in one direction. There is one pending sync interest at each party in the stable state. Due to space limit, only the PIT of router C is depicted.



(b) The state change caused by *Alice*'s new message is multicasted to other two users following the PIT entries set up in routers by sync interests

Fig. 5: State change propagation in ChronoSync

the system is in a *stable state*, and sync interest from each party carries an identical state digest, resulting in efficient interest collapsing in NDN routers [4]. Fig. 5a shows an example of a system in stable state, where there is no ongoing conversation in a chatroom.

As soon as some party generates new data, the state digest changes, and the outstanding interest gets satisfied. For example in Fig. 5b, when *Alice* sends a text to the chatroom, ChronoSync module on her machine immediately notices that its state digest is newer and hence proceeds to satisfy the sync interest with sync data that contains the name of text message. Because of the communication properties of NDN, the sync data is efficiently multicasted back to each party in the chatroom. Whoever receives the sync data updates the digest tree to reflect the new change to the dataset state, and sends out a new sync interest with the updated state digest, reverting the system to a stable state. Meanwhile, the users may send interests to request for *Alice*'s text message using the data name directly. In other more complex applications, the sync data may prompt the applications to perform more sophisticated actions, such as fetching a new version of a file and applying changes to the local file system.

Normally, the state digest carried in the sync interest is recognized by the interest recipients: it is ether the same as the recipient's current state digest or the previous one if the recipient just generated new data. However, even in a loss-free environments, out-of-order packet delivery can result in receiving sync interests with digests that cannot be recognized. For instance, in Fig. 5b, *Ted*'s sync interest with the new state digest (after incorporating *Alice*'s sync data into digest tree,

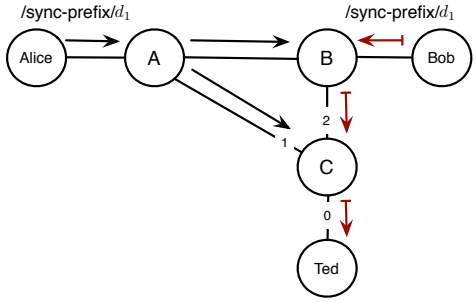


Fig. 6: An example of simultaneous data generation

not shown on the figure) may reach *Bob* before he receives *Alice*'s sync data, due to the possible out-of-order delivery in the transmission.

To cope with this problem, ChronoSync employs a randomized *wait timer* T_w, with value being set approximately on the order of the propagation delay. More specifically, a recipient sets up the wait timer T_w when an unknown digest is received and postpones the processing of the corresponding sync interest until the timer expires. In the example mentioned above, *Bob*'s state digest would become the same as the new digest after *Alice*'s reply reaches him, before T_w expires.

E. Handling simultaneous data generations

In simultaneous data generation cases, more than one data producer reply to the outstanding sync interests. As one interest can only bring back one piece of data in NDN, simultaneous data generations would partition the system into two or more groups, with each group maintaining a different state digest, depending on whose sync data they have received. At the same time, users in different group will not be able to recognize each other's state digest. This is illustrated in Fig. 6, where *Alice* and *Bob* reply to the sync interests at the same time and only *Bob*'s sync data reaches *Ted*. Thus, the new state digest of *Alice* is different from that of the other two.

This problem can be solved with the *exclude filter* [7], which is one of the selectors that can be sent along with the interest to exclude data that the requester no longer needs. When the wait time T_w times out, *Ted* proceeds to send a sync interest with the previous state digest again, but this time with an exclude filter that contains the hash of *Bob*'s sync data. Routers understand that Bob's sync data, although has the same name as the one carried in the sync interest, cannot be used as the reply to the interest. As a result, this sync interest brings back *Alice*'s sync data from router C's cache. Similarly, *Alice* and *Bob* also retrieve each other's sync data with the help of the exclude filter. At this point, all three users have obtained the knowledge about the simultaneously generated data and compute an identical state digest.

If there are more producers involved in a simultaneous data generation event, multiple rounds of sync interests with exclude filter have to be sent. Each of such interest has to exclude all the sync data of a particular state digest known to the requester so far.

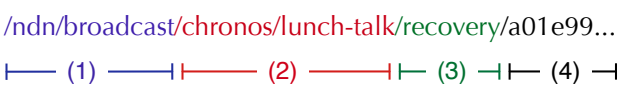


Fig. 7: An example of recovery interest

F. Handling network partitions

When network partitions happens, the users become physically divided (as opposed to the logical division in the simultaneous data generation case) into multiple groups. Although within each group users may continue to communicate due to ChronoSync's decentralized design, there is a challenging synchronization problem when the network partition heals: parties in different groups accumulated different subsets of data and it is impossible for them to recognize each other's state digests. Different from what happens in simultaneous data generations, where multiple users reply to the same sync interest with different sync data, during network partitioning an unknown number of sync data with different state digests may have been generated by multiple parties, rendering the exclude filter ineffective in determining the differences of dataset. Hence, when the interest with exclude filter times out (such interests should have very short lifetime, as the sync data, if there is any, should already be cached in the routers), ChronoSync infers that the network partitions have happened and measures have to be taken to resolve the differences. Depending on specific application requirements, various set reconciliation algorithms [8]–[10] can be used to solve this problem.

For applications such as ChronoChat, for example, ChronoSync resorts to a simple but effective recovery procedure, outlined as follows. The recipient of the unknown digest sends out a recovery interest, as shown in Fig. 7. It is similar to a normal sync interest, but has a "recovery" component before the digest and includes the unknown state digest, instead of the one in the root of the local digest tree. The purpose of such an interest is to request missing information about the dataset from those who produced or recognized the unknown state digest. Those who recognize the digest (e.g., having it in their digest log) reply the recovery interest with the most recent producer status of all users, and others simply ignore the recovery interest. Upon receiving the recovery reply, the recipient compares the producer statuses included in the reply with those stored in the local digest tree and updates the tree whenever the one in the reply is more recent. This recovery procedure guarantees that the system will revert to the steady state within few recovery rounds (e.g. one round for two groups that have different state digests).

IV. Implementation

We implemented the ChronoSync protocol as a C++ library, and also built a proof-of-concept ChronoChat application (Fig. 8 shows screenshot of a demo chat session) and tested it on both Linux and Mac OS X platforms with up to 20 participants. The chat messages were disseminated correctly

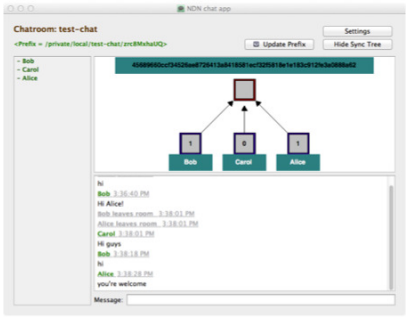


Fig. 8: ChronoChat: a distributed multi-user text chat app

and promptly to all participants. Furthermore, to demonstrate the effectiveness of ChronoSync in supporting more complex applications, we also developed and tested ChronoShare, a distributed file sharing application based on ChronoSync that provides similar user experience as what provided by the commercial counterparts (such as Dropbox), but takes full advantage of NDN's caching and multicast capabilities.

V. Evaluation

To understand characteristics and tradeoffs of the ChronoSync protocol, we conducted a number of simulation-based experiments of the group text chat service (ChronoChat) using NS-3 [11] with ndnSIM module [12], which fully implements the NDN communication model. In particular, we are interested in confirming that ChronoSync propagates state information quickly and efficiently, even in face of network failures and packet losses. To get a baseline for the comparison, we also implemented a simple TCP/IP-based approximation of the centralized Internet Relay Chat (IRC) service, where the server reflects messages from a user to all others. For simplicity of the simulation, we did not implement heartbeat messages for either ChronoChat or IRC service simulations. Also, chat messages in the simulated ChronoChat application are piggybacked alongside with the sync data. That is, when, for example, Alice sends a new message, her ChronoChat app not only notifies others about the existence of a new message, but also includes the actual message data in the same packet.

In our evaluations we used the Sprint point-of-presence topology [13], containing 52 nodes and 84 links (Fig. 9). Each link was assigned measurement-inferred delay, 100 Mbps bandwidth, and drop-tail queue with the capacity of 2000 packets. As the size of the text message is usually small, there is no congestion in the network. All nodes in the topology act as the participants of a single chatroom. The traffic pattern in the room was determined based on the multi-party chat traffic analysis by Dewes et al. [14] as a stream of messages of sizes from 20 to 200 bytes with inter-message gap following the exponential distribution with the mean of 5 seconds.

A. State synchronization delay

ChronoSync-based applications are fast in synchronizing dataset state. To evaluate this property quantitatively, we define *state synchronization delay* to be the the time interval between

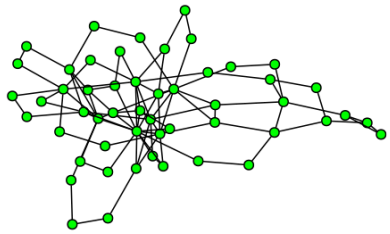


Fig. 9: Sprint point-of-presence topology

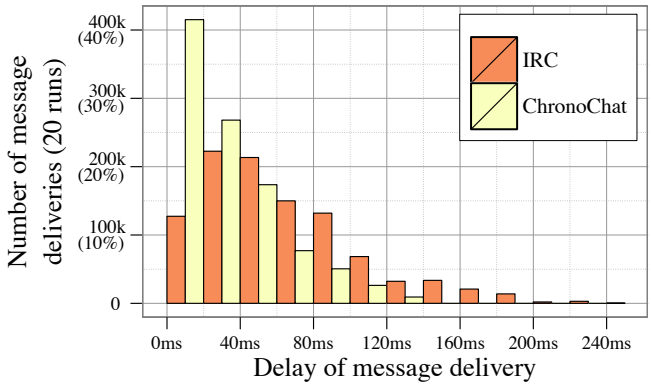


Fig. 10: Distribution of message delays

the message generation and discovery of this message by all of the chatroom participants. We performed 20 runs of the chatroom simulation with 52 participants that produced together a total of 1000 messages under various network condition. Each individual simulation run featured different sets of messages injected to the chatroom, with different inter-message delays, different message sizes, and different order of participants talking. In the IRC case, we randomly chose one of the nodes in the topology as the position of the central server for each run.

1) **Performance under normal network conditions:** As an initial step, we evaluated ChronoChat under normal network conditions without network failures or packet losses, which allowed us to understand the baseline performance of ChronoSync protocol.

Since in ChronoChat sync data always follows optimal paths built by outstanding sync interests, the synchronization delay is significantly lower, compared to that of the client-server based IRC implementation, as shown in Fig. 10: for ChronoChat, more than 40% of all messages sent in 20 runs experienced delay less than 20 ms, compared to ≈13% of messages in IRC case for the same delay range.

2) **Performance in lossy environments:** We evaluated ChronoChat in lossy network environment, with varying level of per-link random packet losses, ranging from 1% to 10%. Fig. 11 summarizes the simulation results in form of cumulative distribution function graphs for ChronoChat and IRC services (for better visual presentation, x-axis is presented in the exponential scale and y-axis is in the quadratic scale). A conclusion can be made from these results that the performance of ChronoChat stays practically unaffected if the network experiences moderate levels of random losses (≤1%). Moreover, even if network conditions deteriorate and random losses increase to abnormally high values (5%–10%), ChronoChat continues to show significantly shorter state synchronization delay, compared to IRC-like systems.

Overall, regardless of the random loss rate value, more messages in ChronoChat experienced smaller delay, compared to those in IRC. This trend is more clear as the loss rate grows: the percentage of messages with small delay drops rapidly in IRC and in ChronoChat it drops more gracefully. However, a careful reader may note that there is a small fraction of messages in ChronoChat that experienced longer delay, compared to IRC. This is because ChronoChat uses NDN's pull based model: a receiver needs to discover a new state first in order to request for it, as opposed to TCP/IP where the source keeps (re-)sending the data packets until it is acknowledged by the receivers. In cases where the sync interests or sync data are dropped so heavily that some participants are not aware of the state change, it has to wait until these participants re-express sync interests or another state change occurs before the message can be disseminated to all users. We believe that adaptive adjustment of sync interest re-expression interval, depending on application requirements and network conditions, should be able to keep synchronization delay within reasonable ranges.

B. Synchronization resiliency to network failures

Another key feature of ChronoSync is its serverless design, which means that users can communicate with each other as long as they are connected. Even in the case of network partitioning, the group of participants in each partition should still be able to communicate with each other, and when the partition heals, different groups should synchronize the chatroom data automatically.

1) **Basic verification of link failure resiliency:** To verify this property we conducted a small-scale 4-node simulation with link failures and network partitioning (Fig. 12). The total simulation time of 20 minutes was divided into 5 regions: 0–200 seconds with no link failures (Fig. 12a), 200–400 seconds with one failed link between nodes 0 and 1 (Fig. 12b), 400–800 seconds with two failed links between nodes 0, 1 and 2, 3 (partitioned network, Fig. 12c), 800–1000 seconds with one failed link between nodes 2 and 3, and finally 1000–1200 seconds period with no link failures.

The results are depicted in Fig. 13, visualizing node 0's knowledge about the current states of all other participants as a function of time. This figure not only confirms that the parties within a connected network continue to communication during the partitioning event, but also the fact that when the network recovers from partitioning, the state is getting synchronized as soon as interests start flowing through formerly failed links.

2) **Impact of link failures:** To quantify the effect of network failures on ability of text chat participants to communicate with each other, we again used our 52-node topology that is now subjected to varying level of link failures. In each individual run of the simulation we failed from 10 to

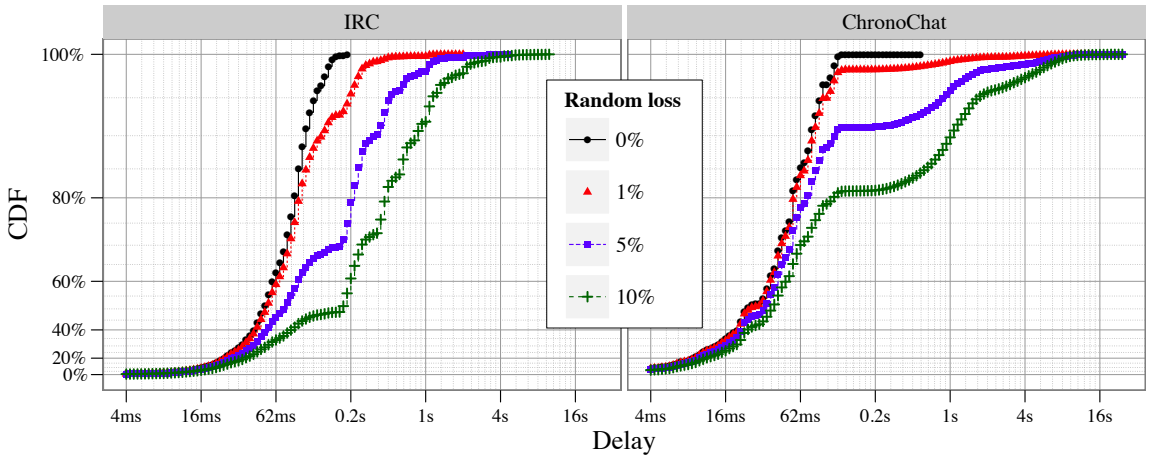


Fig. 11: Packet delivery delay in face of packet losses

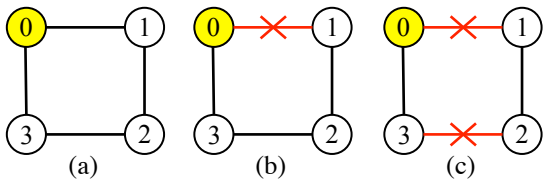


Fig. 12: Simple 4-node topology with link failures (link delays were chosen uniformly at random in the interval 1–2 ms)

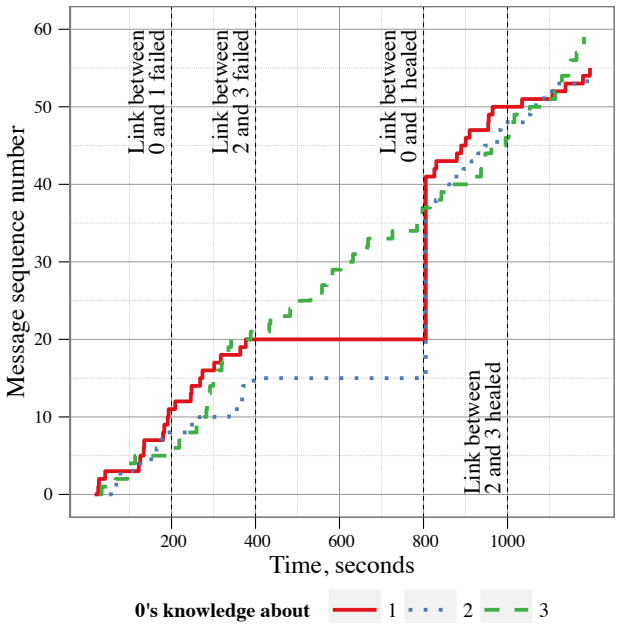


Fig. 13: ChronoChat performance in face of link failures (sequence number progress)

50 links (different set of failed links in different runs), which corresponded to ≈10% and ≈50% of the overall link count in the topology. We performed 20 runs of the simulation for each level of link failures, counting the number of pairs that are still able to communicate. As shown in Fig. 14, we use

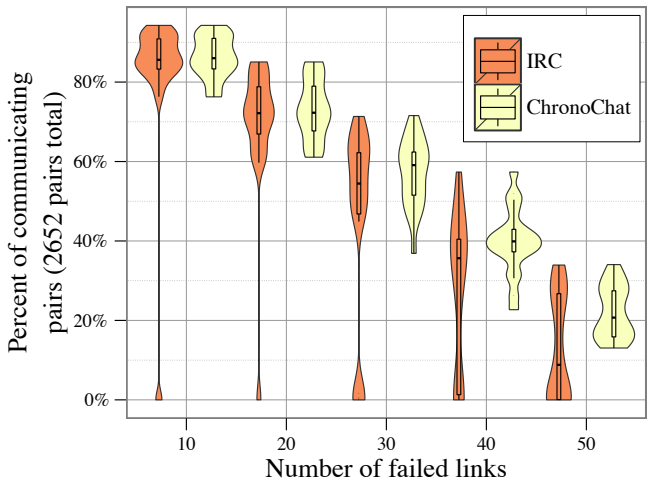


Fig. 14: Distribution of the number of communicating pairs versus number of failed links (violin plot)

a violin plot[4] for this graph to highlight a bimodal nature of the distribution for the percent of communicating pairs in the centralized IRC service: with significantly high probability the users were almost not able to communicate at all (notice the portion of the violin plots near the bottom of the y-axis for IRC). ChronoChat, being completely distribute, always allows a substantial number of pairs able to communicate. For any centralized implementation, like IRC, there is always a single of point of failure and the communication can get completely disrupted even with a small level of link failures.

C. Network utilization pattern

To understand how the fast state synchronization and robustness to links failures in ChronoSync relates to the network utilization, for the same sets of experiments we collected statistics about the number of packets transferred over each

[4]The violin plot is a combination of a box plot and a kernel density estimation plot. Wider regions represent higher probability for samples to fall within this region.

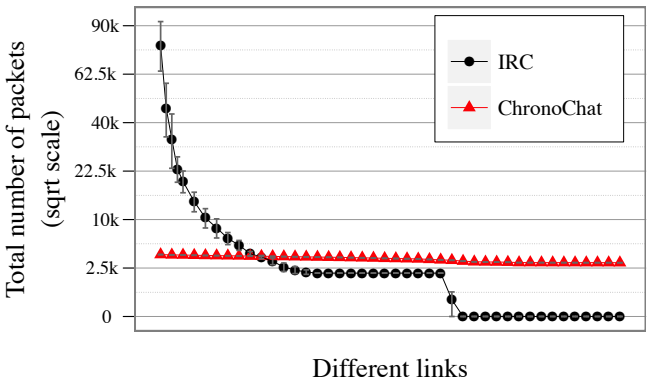


Fig. 15: Number of packets in links (packet concentration)

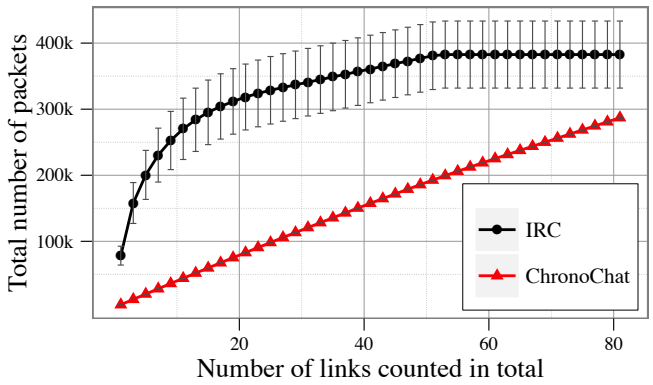


Fig. 16: Cumulative sum of per-link packet concentrations

link in the topology (we call it *packet concentration* for a link). When counting the packets, we included both interest and data packets in ChronoChat, and both TCP DATA and ACK packets in IRC. The obtained data for our 52-node topology experiment is summarized in Fig. 15,[5] where the links were ordered and visualized by the packet concentration value (with 97.5% confidence interval).

The results presented in Fig. 15 show that ChronoChat more or less equally utilizes all of the available network links between participants. [6] Results for network utilization in IRC case show a completely different pattern. A few links close to the server have high packet concentrations, with value as large as ≈90,000 packets (≈90 times of the total number of messages in the chatroom) in the link directly adjacent to the server. Many links that are close to clients have a low packet concentration, while some links, which are not on the shortest path between clients and the server, are not utilized at all.

D. Overall overhead

The difference between network utilization patterns in ChronoChat and IRC highlights an important design trade-off of ChronoSync protocol. As the primary objective of ChronoSync is to synchronize the state in a complete distribute manner as fast as possible, and with ability to mitigate network failures, it utilized more links in the topology compared to IRC. At the same time, as ChronoSync does not have triangular data distribution paths and NDN architecture ensures that each piece of data travels over a link no more than once, the overall overhead in ChronoChat can be even lower than that of the centralized solutions which are generally considered to be efficient in network utilization. For example, the cumulative sum of packet concentrations presented in Fig. 16 shows that in our experiments, where sync interests are distributed by broadcast, ChronoChat still has considerably lower overall overhead compared to that of the IRC service.

[5]the figure summarizes data about experiments under ideal network conditions, but results in lossy environments show similar trends

[6]When not all nodes participate in chat sessions, the interest forwarding strategy would ensure that links that are not on the path between participants, will not be unnecessarily utilized. The specific implementation of a such strategy is one of our future research directions.

Note that ChronoSync also features application-specific trade-offs, which can be directly related to the overall overhead. In particular, when an application seldom generates new data and can tolerate certain synchronization delay, it is not necessary to always keep an outstanding sync interest. Instead, the sync interests can be expressed with longer intervals to reduce the overall overhead.

VI. Discussions

In this section we briefly discuss application domain where ChronoSync fits the best, as well as the scalability and security issues of the ChronoSync protocol.

A. Target application domain

ChronoSync strives to be an efficient general-purpose state synchronization protocol. At the same time, it is most suited for applications with the following features and properties:

1) state synchronization needs to be done in a distributed way, without relying on a mandatory central node;
2) parties contribute to the dataset in a non-deterministic fashion, i.e., it is hard or impossible to predict either who will be generating the new data or when the data generation will happen.
3) all parties wish to have the same knowledge about a dataset;
4) the probability of simultaneous data generations by a large number of parties is low.

Decentralized synchronization keeps the local communication local. For example, synchronizing photos between a laptop and a nearby smartphone does not need to go through the cloud and should be able to happen even without a wireless access point. However, note that ChronoSync can easily work together with infrastructure based storage services without any special configurations. For example, if a chatroom in ChronoChat utilizes a backup service provided by a cloud storage server, the server can simply participate in the chatroom as a regular user.

The non-deterministic way of data generation and the desire to synchronize knowledge of a dataset are often natural for the applications in this domain. For example, in collaborative editing, every user wishes to have the up-to-date version of the

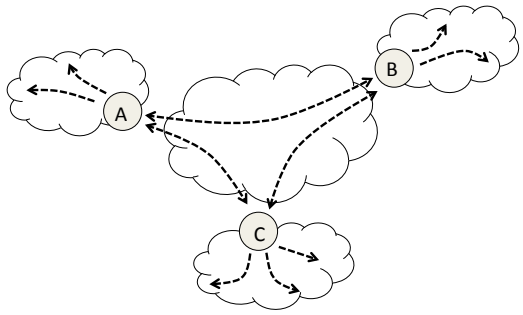


Fig. 17: Overlay broadcast network for ChronoSync

document, but it is unrealistic to predict users' editing behavior. Furthermore, if the data generation pattern is predictable, one can simply request the data directly from the producers at the expected data generation time.

The last requirement ensures that ChronoSync can deduce the differences of dataset states efficiently using recent history of state digests for most of time, without resorting to relative costly state reconciliation methods.

B. ChronoSync in large networks

ChronoSync uses the broadcast sync interests to exchange state digests. However, broadcasting sync interests to the parties scattered in large networks, such as the Internet, could be costly. A possible solution is to build an overlay broadcast network over NDN. As shown in Figure 17, each network with users of ChronoSync-based applications sets up a gateway node, which knows how to forward sync interests to other overlay gateways. A gateway node relays the sync interests received from its local network to gateways in other networks. Vice versa, the sync interests received from other gateway nodes would also be broadcasted in the local network. As a result, the broadcast of sync interests is confined to networks where such interests are desired to be received. Gateways can learn each other's presence through configurations or some other means, but this is out of scope for this paper.

Furthermore, various different distributed applications based on ChronoSync can share the same gateway overlay.

C. Security considerations

From the security point of view, ChronoSync needs to address two problems. First, outsiders and non-cooperative users should not be able to disrupt the state synchronization process by injecting false information about the statuses of other users. Second, non-authorized users should not have access to the data published by eligible users.

In solving both problems, we assume that users can obtain each other's public keys through a trustworthy key management system, such as manually configured key files, certificates signed by a trusted entity, or some other means.

The first problem can be solved by requiring parties to include a signature of the producer status when replying the sync interest. That is, whenever a party needs to reply the sync interest with the producer statuses of other parties (e.g., to a recovery interest), it also needs to include the corresponding original signatures. Figure 18 shows an example of sync reply to a newcomer. The recipient can easily verify the validity of each producer status and update the digest tree only if the signature is valid.

/ndn/broadcast/chronos/lunch-talk/000...		
Alice's name prefix	37	Alice's signature
Bob's name prefix	21	Bob's signature
Ted's name prefix	96	Ted's signature
NDN packet signature		

Fig. 18: Example of a sync reply to a newcomer

Restricting access to the private data can be implemented through dataset sharing moderation. That is, each sharing group can have one or several moderators, who can grant or reject access to the data. The protected data should be encrypted using a shared secret key, effectively preventing outsiders from eavesdropping. To request access privilege to the protected data, a party would need to publish its encryption public key and to ask for permission from the moderator(s). As each data packet in NDN is signed, the moderator(s) can easily fetch the key and verify if the party is authorized to access the data based on the signing public key for the NDN packet signature (which is different from the encryption public key). If the access should be granted based on the policy, the moderator publishes a shared secret key, encrypted with the encryption key provided by the requester.

After a requester fetches and decrypts the shared key, it gains the ability to participate in the synchronization process with others who are already in the group.

VII. Related Work

There is an extensive amount of research to bring multicast functionality, and in particular reliable multicast functionality [15]–[17] to the Internet. NDN architecture, based on which ChronoSync is designed, natively solves the multicasting problem, but requires applications to be implemented using a pull-based data delivery model, i.e., users need to explicitly request for data. ChronoSync protocol gives an opportunity for the classes of applications listed in Section VI-A to efficiently discover names for dynamically generated data.

To some extent, the design of ChronoSync protocol was inspired by the CCNx Synchronization protocol (*ccnx-sync*): the protocol to facilitate automatic synchronization of data collections in CCNx repositories [18]. However, ChronoSync and ccnx-sync are completely different protocols with different objectives—synchronizing knowledge about the data collections versus synchronizing the data collection itself.

The key building block of ChronoSync design—a compact representation of the knowledge about the whole dataset as a hash value—is based on concept of Merkle trees (hash tree) [6]. The Merkle tree is widely used in many different areas, including file systems to verify/maintain integrity of

the data on disk [19], anti-entropy mechanism in distributed key/value stores [20], and many others.

Another component of ChronoSync design (reconciliation of knowledge about the date collection) is closely related to numerous research efforts that aim to efficiently discover differences in files, folders, and databases: RSYNC [21], [22], CDC in LBFS [23], TAPER [24] to name a few. However, in most cases, the nature of applications for which ChronoSync was designed (see Section VI-A) allows efficient difference discovery without resorting to any complex state reconciliation procedures. For other types applications, ChronoSync supports the use of any of the existing or new promising algorithms, for example the algorithm crafted by Eppstein et al. [8], in addition to the simple state reconciliation approach described in Section III-F.

There is also a rich literature of peer-to-peer solutions [2]. In general these solutions are designed to run over today's TCP/IP network and build an application level overlay to interconnect peers. Such an overlay can be subject to frequent changes as users join and leave, and are unaware of the underlying network topological connectivity. Even though some solutions offer application level multicast data delivery, the resulting data distributions tend to be inefficient due to the mismatch between the overlay and the underlay network topology.

VIII. Conclusion

In this paper we presented ChronoSync, a dataset synchronization protocol for distributed applications running in NDN networks. Leveraging on NDN's interest-data packet exchanges for fetching named data, ChronoSync effectively names the state of a dataset by its digest at a given time. Carrying the name of the dataset state, each sync interest is broadcasted to all participants in a synchronization group to solicit "data" that reports changes in the dataset. The design takes a completely distributed approach, and the resulting ChronoSync protocol removes both single point of failure and traffic concentration problems commonly associated with centralized implementations. Our simulation results also show that ChronoSync is highly robust in faces of packet losses, link failures, and network partitions.

We hope that ChronoSync represents a step towards a new direction of providing useful building blocks in supporting distributed applications development. The initial ChronoSync design emerged during our effort of developing a chatroom application to run over NDN. Since then we have also used ChronoSync in developing a rather different application, ChronoShare (a.k.a. NDN-Dropbox) that provides distributed file sharing among a set of users. To verify whether ChronoSync can support a wide range of distributed applications, or what needs to be changed to enable it, as part of our ongoing efforts we plan to apply ChronoSync to support more types of applications running over NDN. For example multi-party audio conferencing over NDN [25] can leverage ChronoSync to propagate the speaker information and instruct the listeners to fetch the audio streams from active speakers; resource discovery applications such as zero configuration networking [26] could be another candidate to apply ChronoSync protocol.

We also hope that this work can help stimulate more discussions on the design space of distributed applications over NDN and identify and implement useful building blocks to lower the hurdle of application development.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, 1978.

[2] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, 2004.

[3] "BitTorrent Sync," http://labs.bittorrent.com/experiments/sync.html.

[4] L. Zhang et al., "Named data networking (NDN) project," PARC, Tech. Rep. NDN-0001, 2010.

[5] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A case for stateful forwarding plane," *Computer Communications*, 2013.

[6] R. C. Merkle, "A certified digital signature," in *Proc. of Advances in Cryptology*, 1989.

[7] "Ccnx techincal documentation: Ccnx interest message," http://www.ccnx.org/releases/latest/doc/technical/InterestMessage.html.

[8] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," *Proc. of SIGCOMM*, 2011.

[9] Y. Minsky et al., "Set reconciliation with nearly optimal communication complexity," *IEEE Trans. Info. Theory*, 2003.

[10] J. Feigenbaum et al., "l^1-different algorithm for massive data streams," *SIAM Journal on Computing*, 2002.

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

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

[13] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with Rocketfuel," *IEEE/ACM Transactions on Networking*, vol. 12, no. 1, 2004.

[14] C. Dewes, A. Wichmann, and A. Feldmann, "An analysis of Internet chat systems," *IMC'03*.

[15] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," in *Proc. of SIGCOMM*, 1995.

[16] L.-W. H. Lehman, S. J. Garland, and D. L. Tennenhouse, "Active reliable multicast," in *Proc. of INFOCOM*, 1998.

[17] S. Paul, K. K. Sabnani, J.-H. Lin, and S. Bhattacharyya, "Reliable multicast transport protocol (RMTP)," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 3, pp. 407–421, 1997.

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

[19] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: a ZFS case study," in *Proc. USENIX conference on File and storage technologies*, 2010.

[20] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, 2007.

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

[22] S. Agarwal, D. Starobinski, and A. Trachtenberg, "On the scalability of data synchronization protocols for PDAs and mobile devices," *IEEE Network*, vol. 16, no. 4, 2002.

[23] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, 2001.

[24] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proc. USENIX Conference on File and Storage Technologies*, 2005.

[25] Z. Zhu, S. Wang, X. Yang, V. Jacobson, and L. Zhang, "ACT: An audio conference tool over named data networking," in *Proc. of SIGCOMM ICN Workshop*, 2011.

[26] "ZeroConf," http://www.zeroconf.org/.