TreeSync: A Distributed Message Synchronization Algorithm Using Topology-Related Hierarchy over Named Data Network

***Abstract*—Key problem of multi-user real-time communication applications, such as group chat and video conference, is how to synchronize messages among all participants. Traditional IP-based way depends on central server, resulting in unbalanced link burden and robust problem. A recent solution on NDN, ChronoSync, has several crucial limitations. In this paper, we propose a new distributed algorithm based on Named Data Network to address this problem. In our design, the system generates a tree topology according to the real one. Every node takes responsibility to synchronize its children and bubble messages up to its parent. On receiving control message from upper node, participants send request to the real message directly, taking advantage of NDN's consumer-driven design. This method synthesizes the powerful control ability of servers and NDN's distributed features. We implemented TreeSync on ndnSIM and compared it with ChronoSync. It proved small overhead and fast synchronization.**

I. Introduction

Applications dealing with multi-user communications, like group chat and video conference, need efficient and robust way to synchronize among participants. Traditionally in IP-based network, all the participants should register on a central server, send self-generated messages to it as well as fetching data back from it. Due that IP-based network only provides support for point-to-point communication, everyone should have to set up a connection with the server. The server has to send a copy to every participant regardless of the fact that they may be near each other in the topology, resulting in performance problems, such as high burden of links and bandwidth, extremely big overhead, etc. What's worse, the whole system goes down when the server fails.

As one of the Future Internet Architecture(FIA) [1], Named Data Network(NDN) [2] goes over the basic downsides of IP-based network, providing the basis of this new method to handle synchronization problems.

1) Instead of point-to-point connection, resources in NDN are identified by a unique name. Consumers just send interests for the specific name rather than to a certain IP.
2) In NDN, routers will filter the duplicated requests and only send one copy to the next hop.
3) NDN routers will response to all incoming interests when receiving data from next hop.
4) NDN router has the ability to cache data packet thanks to Content Store, which enables the routers to response immediately for a same interest.

Based on these features, it's time to get messages synchronized in distributed way to gain small overhead and high robustness.

There is a distributed algorithm to deal with synchronization problem by researchers in UCLA, the *ChronoSync [3]*. ChronoSync is a pure distributed protocol, where participants in the system broadcast sync interests to the whole system. Since the newly generated message can be responded directly to the sync interests, it can be very fast for every one to receive it. However, ChronoSync also suffers from several downsides.

1) There is no server, so no control ability when conflict happens. For example, when there are simultaneous messages generated, the existing sync interests may be consumed by different messages, resulting in divide of the system.
2) Sync Interest is broadcasted, so it is not scalable.

We are going to talk about ChronoSync in Detail in Section II.

Our Contribution includes:

1) proposed a new algorithm handling the problem of message synchronization using NDN.
2) implemented our new protocol on ndnSIM [4].
3) evaluated the performance of our methods. Demonstrated the merits of our design in comparison to ChronoSync.

The rest of the paper is organized as follows. Section II shows related work and their limitation. We describe our protocol design in detail in section III. In section IV, we talk about the implementation on ndnSIM over NS3 platform and evaluate the performance. We conclude the paper in section V.

II. Related Work

Zhenkai Zhu et al recently introduced a new way to handle this problem on NDN, which is an entirely distributed and server-less protocol, named ChronoSync.

In ChronoSync, clients keep a digest of current message status to identify new messages. Everyone broadcasts a sync interest periodically to the system in order to get the latest news. Whoever receives a sync interest will compare the digest with his own. If they are the same, he keeps it as a pending interest. If the digest is in his log, he sends back data to tell what's missing. Otherwise, it is an unknown digest.

The main downside of ChronoSync is its short of control ability, resulting in problems to handle frequent simultaneous data generation. When this happens, the system will be divided into two different groups and can't recognize digest from each other. For example, as shown in Figure 1, Cindy sends out

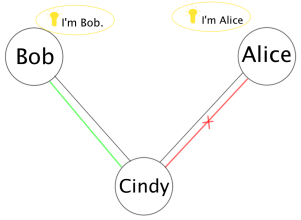

Fig. 1: Simultaneous Message Generation Problem on ChronoSync

sync interest to ask for a latest messages whoever generates. If Alice and Bob say something at a close time, that is, Alice say something before Bob's words reach Cindy. In this manner, Alice will respond to Cindy's sync interest, but it will never reach Cindy because the sync interest from Cindy can only fetch one piece of data back in NDN. At this time, Bob and Cindy are in the same state, but Alice isn't. The system is divided into two group that can't recognize each other.

The solution of ChronoSync is to send recovery interest with the unknown digest. The one who can recognize this digest will send all his current statuses of the whole system back so that the receiver could compare the status with his own to find out what's missing. It can work, but results in much overhead especially simultaneous data generation is frequent. Unfortunately this seems to happen quite often during a group chat. Things are worse in case of more frequent conditions, like video conference.

Another downside of ChronoSync is that it broadcast sync interest to the whole system. Obviously it works fine in small groups, but not a large one.

We will talk about the design and how TreeSync handles these limitations in next section.

III. Protocol Design

A. NDN Background

This section briefly goes through the necessary Named Data Network architecture background in support for our protocol. In NDN, communication is consumer driven. Consumer sends out interest to fetch the data back. Every piece of data is specified by its unique name. When a router receives an interest, it will first check whether the desired data is in its *Content Store(CS)*. If the data packet is already cached in Content Store, the router will drop the interest and return the data packet. If not, the router checks the *Pending Interest Table(PIT)*. In the case the record for this interest is in a router's PIT, the router simply drops the interest and adds an interface to this PIT record. Otherwise the router will finally check the *Forwarding Interest Table(FIB)* by means of *Longest Prefix Match* to determine which interface to forward the interest to. When it still can't find the right way to forward the interest, it drops the interest.

When the publisher receives the interest for his data, he will send the data packet with the same name. On receiving a data packet, the router will check its PIT to see which interface comes the interest for this data. When found, the router forwards the data packet to all the interfaces in the PIT corresponding to the data name and then deletes this PIT record. After that, the router will cache the data packet in its Content Store so that it can satisfy the interest for the same data in the future. In the case that there is no record in the PIT for this data, it means that no one sends the interest for this data or the interest timed out, so it can cache the data packet in its Content Store or just simply drop it. In this way, data packets will be forwarded back to all the consumers in the reverse path of interest packets.

B. Design Goals

We kept several goals when designing the protocol.

1) No central servers. We should make sure that it is a distributed robust protocol.
2) Has ability to handle simultaneous data generation.
3) Scalable.

To achieve these, we design a tree topology. Each node is the controller of its children nodes. Multi tiers make sure that it is scalable. Each controller is selected by the system according to topology, and the root of the tree is just someone who is selected multi times, nothing else special. If a controller fails, one of his children takes his place. Finally, since every controller is in charge of the control message synchronization in its group, the simultaneous data generation will not become a problem.

C. Overview

In this section, we are going to talk about the whole synchronization process briefly. We will discuss every part in detail in the following sections.

1) Tree Topology: When someone join a system, the first thing for him is to find his position in the topology. At the very beginning, everyone waits for a specific random time before sending *Any Server Interest* inquiry. If he doesn't receive a reply, he will assume himself the controller, and thus ready to satisfy the same interest from others. On receiving this reply, a node will set its parent and request control messages from it. The *Any Server Interest* is labeled by a specific level to enable multi-level controllers.

Everyone sends a heart beat interest to his server to check whether the server is still available. Once the server failure is detected, a new server that substitutes him will be selected from the clients in the same group. A new participant can find a controller to be its parent in his neighborhood,

2) Control Message: Controllers control the sync process by what we call *Control Message*(CM for short). CMs are not real data, but the packet to tell when and where the data is. Besides the actual data's NDN name, the CM keeps a label. When CM propagates down the tree, each controller insert his name and the time he receive it to the label. As a result, a node will receive a CM containing the name and time of all its parents.

TABLE I: Interests' Names

Any Server Interest	/broadcast/treesync/anyserver/*level*
Something New Interest	*/serverName*/treesync/somethingnew/*mylabel/dataName*
Anything New Interest	*/serverName*/treesync/anythingnew/*label1:label2:label3*
Data Interest	*/publisherName*/treesync/*time*

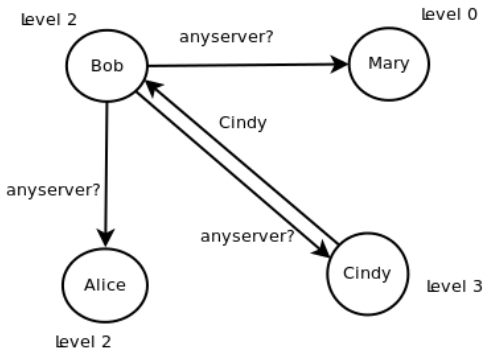

Fig. 2: Server Generation

3) Synchronization: When a participant knows something new, he will propagate it down to his clients and bubble it to his parents.

4) Bubble Up: In the case a node has something new to tell, it sends interest together with the true data name to inform his parent about it. When the controller receives such interest, he adds his name and time to the label of this record and save it in local log. If he also has a parent, he bubbles this up in the same way.

5) Propagation Down: In order to get latest CM, every participant sends sync interest, which we call *Anything New Interest*, to its server, with the latest labels he knows. When the sync interest times out, a same interest is resent. On receiving the sync interest, the server compares the label with his own.

1) If the label's time is the same as his, it is saved as pending interest for the sake of immediate reply when status updates.
2) If the label is older than his latest label, he will find all the new records after the specific time and send them back in data packet.

On receiving response of sync interest, a client propagates it down by satisfying the sync interest from his clients, thus forwards the CM to the whole system.

6) Actual Data Fetching: On receiving CM, everyone sends out interest for the specific data name directly. Since that the name of every piece of data is stable in NDN, the interest from every participant for the same data will be aggregated by routers ensuring no duplicated interests in one link. When the data packet is returned from the publisher, it travels the reverse path of the interest to all the receiver, resulting in small overhead. Besides, the data packet can be cached in router's Content Store, so when there is another participant requesting for the same data, he will get it in the router instead of the publisher.

D. Naming Rules

Naming is one of the most important parts of NDN applications. In this section, we briefly go through all the interest names that we use in this design, and they will be discussed thoroughly in the following sections for details.

There are four interest names in our design, as in Table I. *Any Server Interest*(ASI) is used for server generation. *Something New Interest*(SNI) and *Anything New Interest*(ANI) are responsible for synchronizing CMs. *Data Interest*(DI) serves for actual data fetching. The first part is used for router to forward the interest packet correctly. *'/broadcast'* interest is routed to all the corresponding interfaces. *'/serverName'* and *'/publisher'* are routed to specific nodes. The second component, *'/treesync'* is a specific label for the application. [1]

Other components of the names are specified by the procedures, and we will cover them in the following sections in detail.

E. Tree Topology Generation

To generate servers, the *Any Server Interest* is used. The first two components of the name is used for router forwarding and interest handling. The third one is the interest type. The last element *'level'* is a number indicating in which level the sender currently is.

In the very beginning, everyone doesn't have a server. They will individually wait for a random time and broadcast ASI to inquire whether there is a server nearby. Then they wait a time T_s for the reply. On receiving the data containing the server name, the participant will set his server and request for CM from it later. Otherwise, if he doesn't receive a data after Ts, he assumes himself as the server and thus is ready to reply the such interests from others. When the region is large, there should be many groups, each has a selected server. These servers will obey the same rules, wait for a random time, and broadcast an ASI with their current level. After sending this interest, the sender waits a time related to his current level [2] for the data. If he doesn't receive it after the time expire, he increase his level. The one who receives this interest will compare this level with his own. If this one is lower, he will send data packet to satisfy this interest. We can set a small constant number as the Top Level. Whenever a participant has not a server and he is not a top level server, he will act as above in server generation period. If a participant already has a server, he sends heart beat interest to his server to detect possible failure. We will cover failure reversion in Section III-I.

Noting that the server generation procedure is closely related to the reply delay, so the randomly selected multi-level servers are closely related to the topology as well. The

[1]To inserting FIBs into routers, the client publishes the interest name, declaring that they can reply to such interest. The router protocols, such as OSPF [6] [7], are responsible to inform the other routers this information. ndnSIM provides a function to set the global scale routing FIBs for every node, so in our simulation, we use this to simulate this action

[2]This time is a function of current level, because the higher the level is, the larger its region should be, the longer wait time for Any Server Inquiry will be. So the wait time can control the region's area for a particular level. In implementation, interest sender ignores data packet received after this time. However, it is better to support to set the interest life time(hop count) in the low level so that there will not be extra interest sending to unreachable areas.

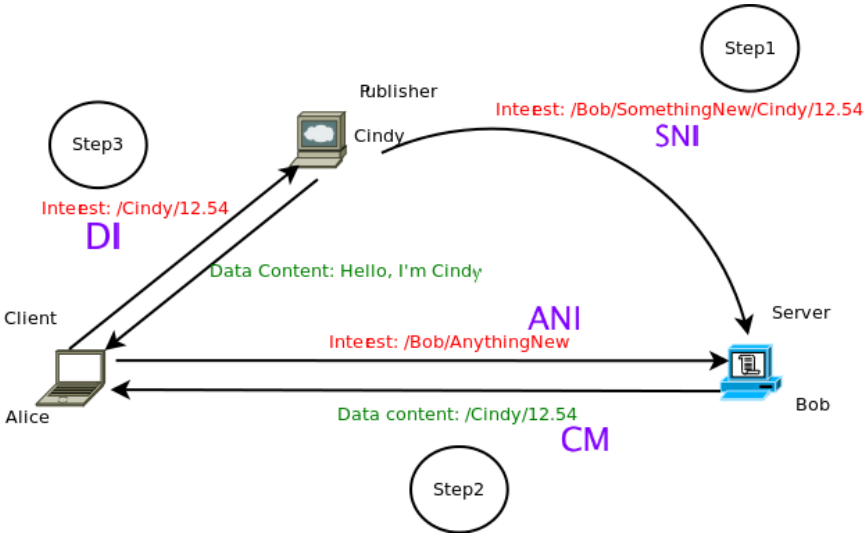

Fig. 3: Synchronization and Data Fetching

direction of transmission is related to the topology, thus it is smart enough not to forward the messages in round. This provides high performance for transferring CM between clients and servers.

F. Synchronization

We describe how to synchronize in the system in this section. As shown in Figure 3, a client sends SNI to inform server a new change. The server then reply pending ANI with the new change. Participants fetch actual data in distributed way. Figure 4 shows multi-level synchronization. A node will transfer the new status to upper and down level, through sending SNI and replying ANI respectively.

1) Something New Interest: Once a node generates a new message, or he receives a SNI from his client, he will bubble this up by sending this interest to his server with the actual data name.

When a participant generates a new data, he will do the following things:

1) adds the data to his Message Container.
2) adds his own label to the record and stores it in his local Record Container.
3) sends SNI with his label and the actual data name to his server
4) processes pending interest.

Note that SNI contains the actual data name in the end, so once receiving this interest, a node can immediately send this status change to upper and lower level without extra delay.

2) Anything New Interest: To get CM, every node sends periodically ANI to his server with the newest label on behalf of his status. When a server receives this interest, he compares the carried label with his own latest label. There are three cases here:

1) The received label is the same as his own. In such case, he saves this interest as pending interest, so that he can reply immediately when his status changes.
2) The received label is older than the his own. In this manner, he compares the record's time and find the first index in the Record Container that is newer than the given label. Then he sends back the new records.
3) There is not any record in his Record Container that shares the same user name as the received label. In this situation, the server has to send back all the records for synchronization.

When the client receives data for ANI, he will do the following things:

1) extracts records from the data.
2) updates his own Record Container.
3) processes pending interest.
4) sends data interest to fetch real data.

In general, a server is in charge of inner group synchronization, so every record in the group contains a label of the server. Thus in regular case, the clients only send his server's latest label. When the system is stable and all nodes share the same status, all ANIs are the same and thus they can be aggregated by NDN routers. Data back can transmit efficiently along the reverse path of interest. When a node moves to another place or the server fails, the node will connect to a new server. In this situation, the last server's label is of no use, so he sends all the latest labels to the new server representing his status. This model can provide natural support for robustness and mobility, which will be covered in section III-H.

G. Distributed Actual Data Fetching

The servers are only in charge of synchronizing CMs, in other words the servers tell the participants in the group when and where to fetch a data. Actual data fetching is acted in distributed way. This design minimizes the messages that travel between server and client, and takes fully advantages of NDN's distributed features such as interest aggregation and caches.

When receiving SNI or data from ANI, a participant sends DI to fetch actual data. As described above, everyone stores

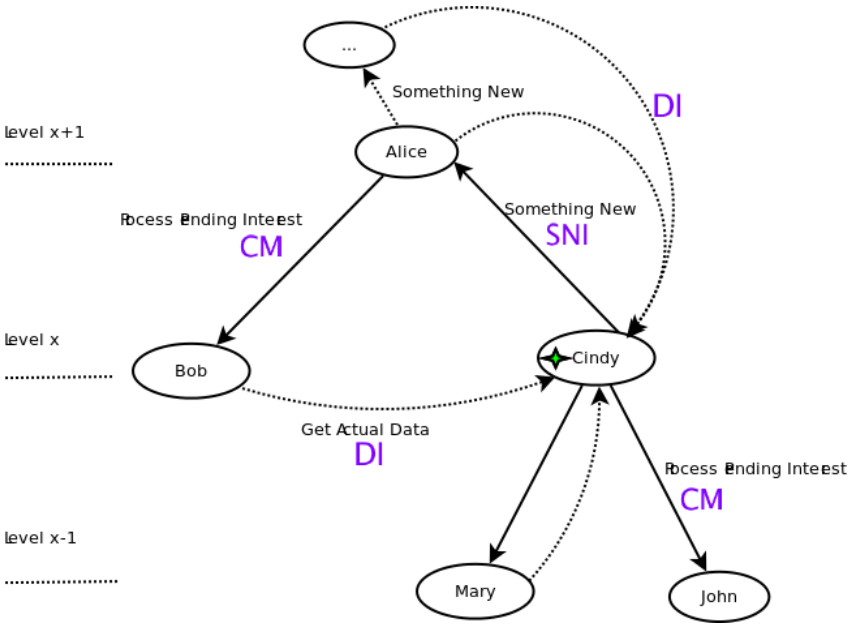

Fig. 4: Multi-Level Synchronization

his self-generated data for the sake of replying others' data fetching interest. They don't need to store the fetched data from others, because they only care about whether they have received it or not.

As a real example of how TreeSync works, as in Figure 4, let's say at some time, Cindy generates a message. She will send SNI to her upper node Alice to inform her this new data. In the meanwhile, Cindy immediately satisfies the pending sync interest from Mary and John by sending them CM. When Alice receives SNI from Cindy, she bubbles the SNI up to the upper tier and propagates down by satisfying Bob's ANI. Finally, everyone send DI to Cindy for the message itself no soon than it receives CM or SNI.

H. Robustness and Mobility Support

Our proposed algorithm can provide natural robustness and mobility support.

When a node goes off-line and returns after some time and connects to the same server, he will send his latest status, which can be recognized easily by the server, since that the server can just compare the time in the label with his own record numerically, find all new messages, and reply to the client. When a node move, whether it is caused by link failure, server failure or the actual move in topology, there are three situations.

1) It moves to another place, but connects to the same server. In this manner, he can communicate with the server and fetch new things without any affection.
2) It moves to another place, but connects to a different server. Chances are that he and the new server share a same upper level server [3]. Although he could not use his former direct server's labels to communicate with the new server, he can use the shared upper level server's labels with the same performance. For instance, as shown in Figure 5, assume that Mary moves close to Bob and connects to him. She can not use her former server Cindy's label now. However, she and Bob hold Alice's label together, thus Bob can still recognize her.
3) When the node is moved to a completely new environment that even the Top Level Server is not the same. This may be caused by contemporary partition or permanent one, which means the two groups never connect to each other. This situation is extreme, and the node has to exchange all the messages. However, in deed it is not reasonable to handle this condition.

[3]actually the Top Level Server is always shared by the nodes, but chances are high that a shared upper level server not far away is available because server selection is closely related to topology.

I. Trouble Shooting

Clients send heat beat interest to detect connection failure. When detected, the client will reset his server and enter server generation period. There are two situations here, the server goes down and the client goes down.

In the first manner, the server is off-line, so the clients in the group will soon select another one to substitute him after the detection. If there are higher level servers, the newly selected server will surely connect to the upper server by sending another ASI with a bigger level label, reverting the system to stable architecture.

In the second one, we consider a more general condition that a small group disconnected with the outside world for a while. The participants inside the group can still communicate with each other in this case, and the highest level server in this group will grow to be a *Top Level Server*. When the connection heals, the group should be able to connect to the outside world again and exchange messages instead of remaining a close system with its own *Top Level Server*. To solve this problem,

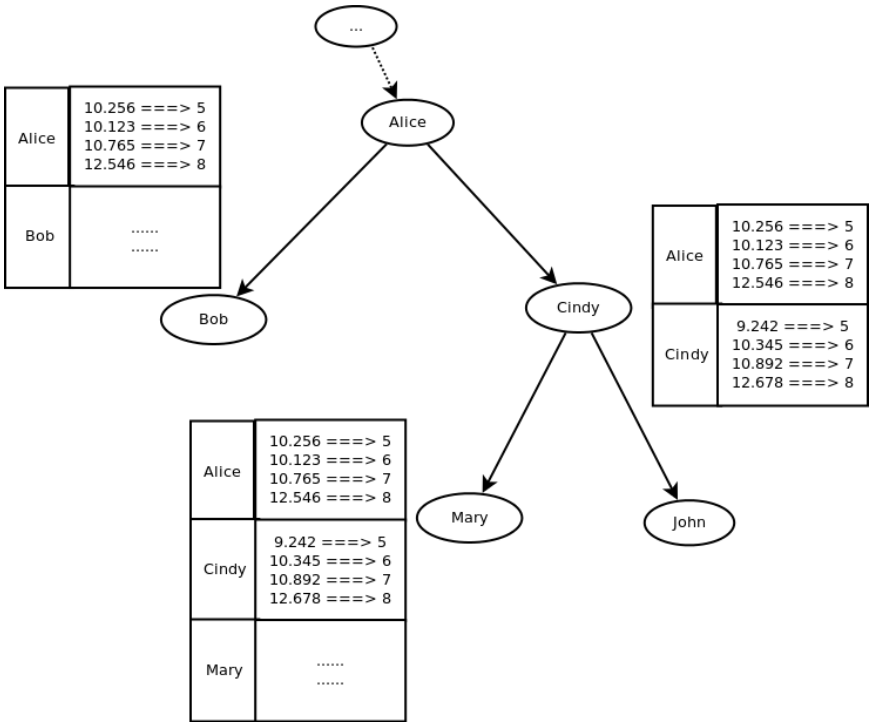

Fig. 5: Robustness and Mobility Support

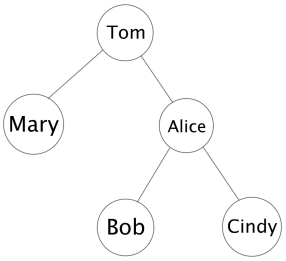

Fig. 6: Handling Simultaneous Message Generation

a server can broadcast an interest inquiring if there is a same level server nearby. If so, the server will decrease his level, set his server to the newly detected server and tell his client to connect to the new server too.

J. How TreeSync Handles ChronoSync's Downsides

As the discussion in Section II, ChronoSync has two major limitations, trouble of handling simultaneous data as well as no scalability support. In this section, we will talk about how TreeSync solves these problems using the design presented above.

1) Simultaneous Message Generation: TreeSync has powerful control ability to handle simultaneous message generation. Consider the topology in Figure 6, take Alice for example, we consider three different situation: two simultaneous data from upper tier, one from upper and one from lower, and two from her children region.

When Tom and Mary say something together, Tom's message will go directly to Alice by answering her pending ANI. When Mary's message reaches Tom, he will answer another ANI from Alice. If Mary's message reaches Tom when Tom hasn't sent his CM, he can send both messages within one CM, making the overhead even smaller.

In the second condition, Tom and Bob type words at the same time. No matter Tom's CM reaches Alice first or Bob's SNI reaches earlier, it is Alice's responsibility to sort the messages and propagate down to Cindy.

Finally, if Bob and Cindy talks simultaneously, messages will reach Alice and be sorted by her. Alice will than propagate the sorted messages to the outside world.

2) Scalability: TreeSync uses a tree topology, every controller takes responsibility of synchronizing its children. The level number enjoys a exponent decrease against the particpants' number. What's more, the tree topology in TreeSync is generated according to the real topology, so the hierarchy won't add much to message delay.

IV. Implementation and Evaluation

We implemented TreeSync on ndnSIM, an overlay of Network Simulation 3(NS3) [5]. As a comparison, we implemented another new protocol handling status synchronization problem, the ChronoSync. This section, we will first confirm the rightness of the logic. Then we will evaluate the performance on both packet overhead and delay.

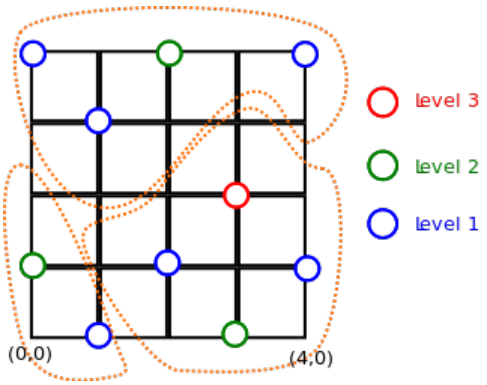

Fig. 7: 5x5 Grid Topology and Generated Servers

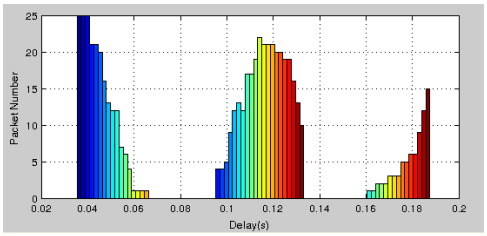

Fig. 8: Delay for every node to receive data

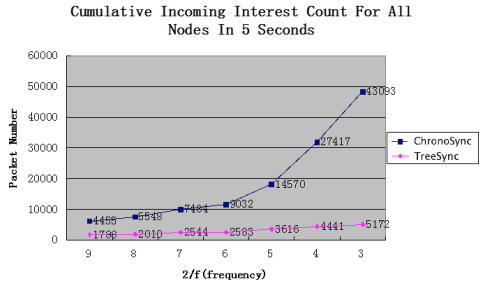

Fig. 9: Incoming Interest Count for All the Nodes

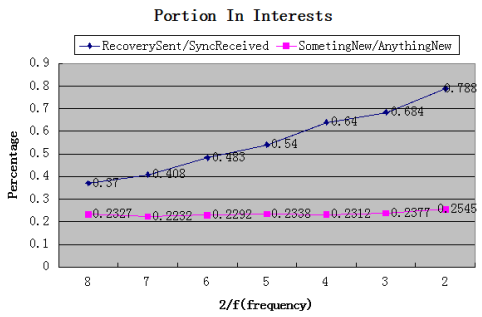

Fig. 10: Portion in Interests

To be more general, We use the 5x5 grid topology in our simulation, as shown in Figure 7. Every node in the grid represents a participant in the synchronization system, thus there are totally 25 nodes. The figure may seem simple, but it actually contains nodes with 6 different environments and a max hop count of 8. We think this topology can represent a general topology well. The tree topology is randomly selected, as described before. All links are symmetrical with link delay of 10ms and 1Mbps bandwidth. We let participants randomly generate data, and we change the frequency of the generation to see the trend. Everyone's message generation period is independent.

A. Functional Correctness

1) Server Generation: In the grid topology, the server generation works well. There forms a four-level architecture, with the top level server on node(3,2).

2) Message Synchronization: All messages are successfully transferred to every participants. Figure 8 shows the delay of the packages at the frequency of $\frac{2}{7}$Hz. From the graph, we can see that delay is divided into three parts, one very quickly, one average, and the other is a little slow. That is related to the tree architecture. If the receiver is close to the publisher in tree topology, he should get the message quickly and sends out real data interest. However, in the case that these two participants are far away in the hierarchy tree (chances are high that they are far from each other in real topology too), the synchronization control messages should go through several servers to reach to the other, resulting in the delay.

B. Evaluation

1) Overhead: Our model holds advantage for control, thus the overhead is much smaller. In this experiment, we change the data generation speed and see the overhead change. As shown in Figure 9, as the frequency enhances, our design holds a linear increase in packet delivery count, while ChronoSync suffers a big increase as the frequency goes up, because when the participants generate message more quickly, chance of simultaneous data generation will increase, resulting in ChronoSync applications send out recovery interests to solve the problem. Note that we use all *Incoming Interest*s, because it can represent to the status. The aggregation of NDN router will decline *Incoming Interest* count too. From the view of proportion that consists the interests, as you can see in Figure 10, the percentage that the receiver will eventually send recovery interest when receiving a sync interest is much higher when data generation frequency increase. However, SNI remains the same ratio with ANI in TreeSync.

2) Delay: Since CMs should go through multi-level servers in our design, in spite that the level is related to topology, the performance is still lower than ChronoSync, in which data always returns along the best way. However, as a sacrifice, when the data is generated frequently, ChronoSync applications should have to send out recovery interests, resulting in a bigger delay. Figure 11 shows the delay as a function of frequency. In the graph, we can see that the delay is a little bigger than ChronoSync, but is still pretty good. Take the gained high control ability into consideration, the sacrifice is worthy.

Another point, the data fetching procedure is distributed, thus can take fully advantages of NDN architecture. Figure 12 shows the delay from a participant receives CM to when he gets the desired data. It is clear that this procedure is very fast,

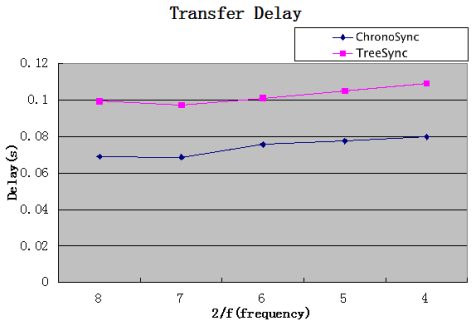

Fig. 11: Delay Compare

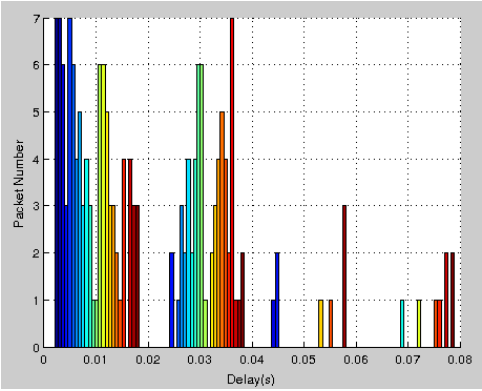

Fig. 12: Data Fetching Delay

and it contributes to small delay and overhead.

V. Conclusion

In this paper, we proposed TreeSync, a new algorithm to handle the synchronization problem in multi-user applications. TreeSync takes advantages of both traditional server-based model and NDN's neat and efficient distributed features. First of all, Multi-level controllers can provide enough control ability to handle complex conditions; Secondly, The essential distributed features allow it to fetch data efficiently and robustly with little overhead; Besides, The hierarchy structure makes the algorithm scalable. We have implemented TreeSync over ndnSIM and evaluated the performance. Message is correctly and fast synchronized in a distributed way. With a reasonable delay of sync control message transmission, the overhead is much lower due to powerful control ability.

Some further topics worth researching include: (1) minimizing within-level delays, (2) generating sever by some better algorithms to generate a hierarchy structure that best fits each specific topology.

References

[1] "NSF Future Internet Architecture Project", http://www.nets-fia.net

[2] Zhang, L., Estrin, D., Burke, J., Jacobson, V., Thornton, J. D., Smetters, D. K., ... & Yeh, E. (2010). Named data networking (ndn) project. Relatrio Tcnico NDN-0001, Xerox Palo Alto Research Center-PARC.

[3] Zhu, Z., & Afanasyev, A. (2013). Lets ChronoSync: Decentralized Dataset State Synchronization in Named Data Networking. In Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013).

[4] Afanasyev, A., Moiseenko, I., & Zhang, L. (2012). ndnSIM: NDN simulator for NS-3. University of California, Los Angeles, Tech. Rep.

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

[6] Moy, J. (1998). Open shortest path first (ospf) version 2. IETF: The Internet Engineering Taskforce RFC, 2328.

[7] Wang, L., Hoque, A. K. M. M., Yi, C., Alyyan, A., & Zhang, B. (2012). OSPFN: An OSPF based routing protocol for Named Data Networking. University of Memphis and University of Arizona, Tech. Rep.