Distributed Random Servers with Timed Labels for Synchronization over Named Data Network

Hebi Li, Xiaobin Tan, Zijian Zhou, Zhifan Zhao
CAS Key Laboratory of Technology in Geo-spatial Information Processing and Application System
University of Science and Technology of China
Hefei, P.R. China
Email: lihebi@mail.ustc.edu.cn, xbtan@ustc.edu.cn, zjzhou62@mail.ustc.edu.cn, zzfan@mail.ustc.edu.cn

***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. In this paper, we propose a new protocol based on Named Data Network to address this problem. In this protocol, the system will randomly select multi-level servers in charge of the synchronization of time-labeled control messages, while data fetching in distributed way. This method synthesizes the powerful control ability of servers and NDN's distributed features. We implemented our design and it proved high quality performance of fast data fetching and strong control ability.**

I. Introduction

Applications dealing with multi-user communications, like group chat and video conference, is an important part of modern life. However, it is still a problem about how to synchronize data within participants. Traditionally in IP-based network, all the participants should register on the central server, send self-generated messages to the server as well as fetching data back from it. Due that IP-based network only provides support for end-to-end communication, so everyone should 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 very much in the topology, resulting in performance problems, such as high burden of links and bandwidth, extremely big overhead, etc. As one of the Future Internet Architecture(FIA) [2], Named Data Network(NDN) [1] goes over the basic downsides of IP-based network, providing the basis of this new method to handle synchronization problems. In NDN, routers will filt the duplicated interest, only send one copy to the next hop, which ensures that there is only one same interest per link. The router will record all the incoming interest's interfaces, so that it could forward the received data to every interface that comes the interest. Besides, NDN router has the ability to cache data packet thanks to Content Store. So if there comes a same interest, the router could satisfy it immediately without fetching for the same data packet again from the producer. The proposed method in this paper take fully advantage of these useful features provided by NDN. In the mean while, this method uses self-generated multi-level servers to synchronize control messages, which is a mix of merits of traditional mode and NDN.

In 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 server, and thus ready to satisfy the same interest from others. On receiving this reply, a node will set his server and request control messages from the server. In the case that the region is large, an upper level server will be generated from the lower level servers in the same way, resulting in multi-level architecture. Everyone send 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 server to get synchronization control messages in his neighborhood.

When a participant generates some data, he will send interest together with the true data name to inform his server about it. When the server received such interest, he will add his name and time to the label of this record and save it in local log. If there is an upper level server, he will inform his server again in the same way.

In order to get latest control messages, 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. 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. 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 this data packet, the client extracts the record from the packet, adds them into his own record log and sends data interest for real data. If the client is also a server in a lower level, he will reply to his client's sync interest, thus forward the synchronization control messages to the whole system.

As for real data fetching, since that the name of every piece of data is stable, 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.

Our Contribution includes: (1) proposed a new protocol handling the problem of multi-user application synchronization over NDN. (2) implemented our new thoughts on ndnSIM [4]. (3) evaluated the performance of our methods. Demonstrated the merits of our design in comparison to another related protocol *ChronoSync [3]*.

The rest of the paper is organized as follows. We describe

our protocol design in detail in section II. In section III, we talk about the implementation on ndnSIM over NS3 platform and evaluate the performance. Section IV shows related work, and we conclude the paper in section V.

II. 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, which means the consumer sends interest to fetch the data back. Every piece of data is specified by its 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. 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 throughly in the following sections for details.

There are four interest names in our design, as in Table I. *Any Server Interest* is used for server generation. *Something New Interest* and *Anything New Interest* are responsible for synchronizing control messages. *Data Interest* 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, *'/drsapp'* is a specific label for the application.[1]

[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

TABLE I: Interests' Names

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

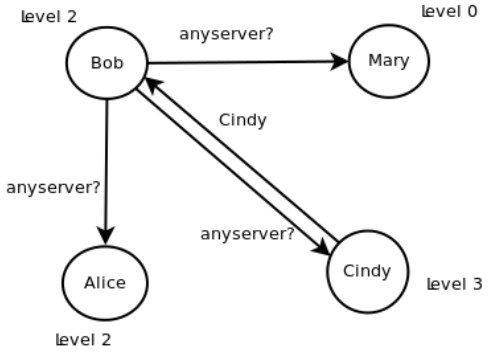

Fig. 1: Server Generation

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

C. Server 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 *Any Server Interest* 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 to this server name and request for synchronization control messages from then on. 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 *Any Server Interest* 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

[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.

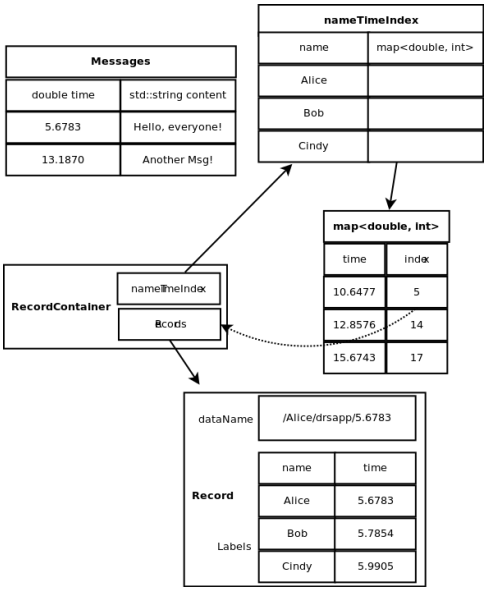

Fig. 2: Data Structure

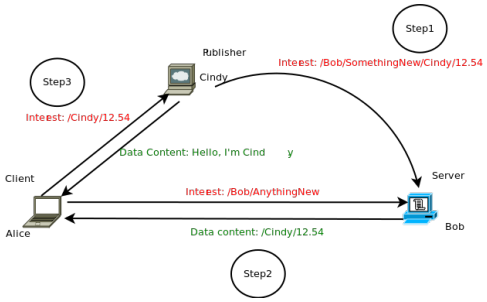

Fig. 3: Synchronization and Data Fetching

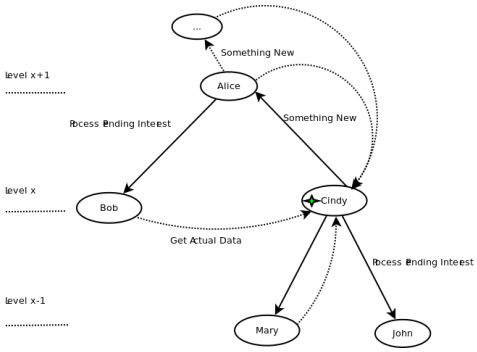

Fig. 4: Multi-Level Synchronization

to detect possible failure. We will cover failure reversion in Section II-H.

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 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 synchronization control messages between clients and servers.

D. Data Structure

In this section, we describe the data structure used for storing and processing synchronization messages.

As shown in Figure 2, there are two data structures in every node, namely Message Container and Record Container. Message Container stores the actual data generated by this participant. Keep in mind that participant doesn't need to store everyone else's messages. They only care about whether they have received all the others' messages. However, it is the publisher's responsibility to store his own published messages for the sake of replying the actual data request interest from others. Record Container holds the synchronization control records. A timed label is the pair of a user name and his time. A record holds a data name and all the labels on behalf of the dataname. A record container consists of two parts, namely *Record Vector* and *NameTimeIndex*. The *Record Vector* is an array of the records by sequence, so that their indexes can tell which is newer. The *NameTimeIndex* structure is a dual map. It is use to find the first index that is newer than a provided label contains a name and time. The index is the one that used in *Record Vector*.

Note that every label about a record contains a user name and user time, thus there's no problem when different participants use different time criteria. Also, since that the record only contains labels from the participants' upper-level servers, and the top level is a small constant number, we don't need to worry about a huge amount of labels. What's more, when a server is down, his labels will never be updated again, so we can use some algorithm like Least Recent Use(LRU) [8] to delete useless labels. One more thing, this design provide support for node failure and node mobility, which is referred in Section II-G.

E. Synchronization

We describe how to synchronize in the system in this section. As shown in Figure 3, a client sends *Something New Interest* to inform server a new change. Server then reply pending *Anything New Interest* 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 *Something New Interest* and replying *Anything New Interest* respectively.

1) Something New Interest: Once a node generates a new message, or he receives a *Something New Interest* from his client, he will send 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 *Something New Interest* with his label and the actual data name to his server (4) processes pending interest. On receiving a *Something New Interest*, a server will do the following things: (1) removes the labels in this record [3], add his own label and store it into local Record Container. (2) sends Something New Interest with his label and the actual data name to his server, if has.

[3]because every node only keeps the labels from his upper level servers.

(3) processes pending interest (4) sends interest with the data name to fetch actual data.

Note that *Something New Interest* 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 synchronization control records, every node sends periodically *Anything New Interest* 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 Anything New Interest, 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. So 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 *Anything New Interest*s 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 II-G. Useless out of date labels will be deleted by some algorithm such as LRU.

F. Distributed Actual Data Fetching

The servers are only in charge of synchronizing control records, 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 *Something New Interest* or *Anything New Data*, a participant updates his *Record Container* and send *Data Interest* to fetch actual data. As described above, everyone stores 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.

G. Robustness and Mobility Support

Our proposed model 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 the new one, 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) he 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) he moves to another place, but connects to a different server. Chances are that he and the new server share a same upper level server [4]. Although he could not use his former 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.

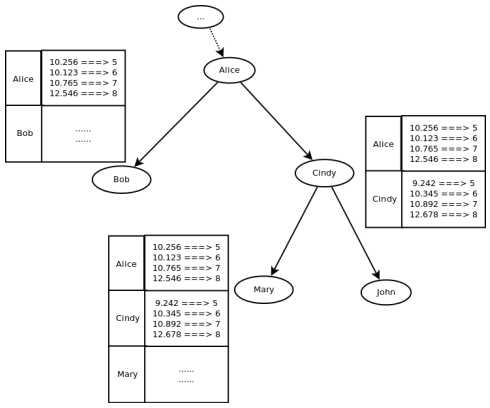

Fig. 5: Robustness and Mobility Support

H. 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 is down and the client is 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 *Any Server Interest* with a bigger level, reverting the system to stable architecture. In the second manner, 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,

[4]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.

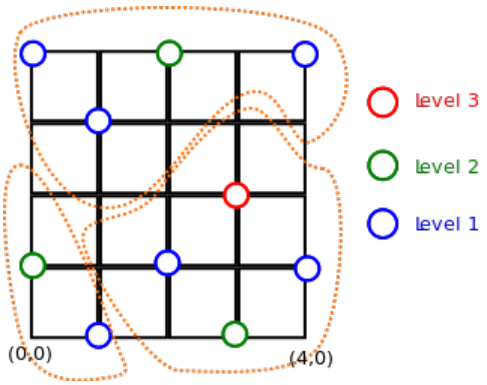

Fig. 6: 5x5 Grid Topology and Generated Servers

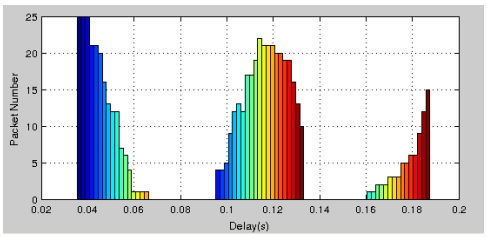

Fig. 7: Delay for every node to receive data

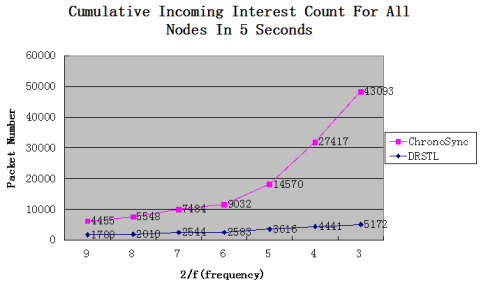

Fig. 8: Incoming Interest Count for All the Nodes

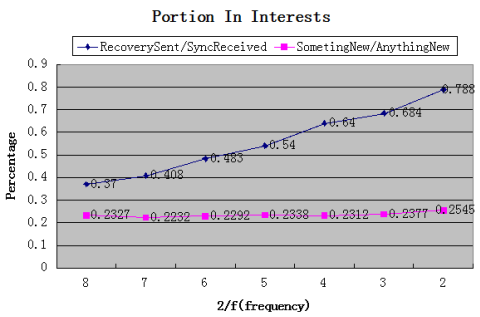

Fig. 9: Portion in Interests

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, 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.

III. Implementation and Evaluation

We implemented our protocol 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.

To be more general, We use the 5x5 grid topology in our simulation, as shown in Figure 6. Every node in the grid represents a participant in the synchronization system, thus there are totally 25 nodes. The server 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 frequence 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). The formed architecture should be better if the topology is random, not symmetrical.

2) Message Synchronization: All messages are successfully transfered to every participants. Figure 7 shows the delay of the packages at the frequence 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 level architecture. If the receiver is close to the publisher in level 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 the topology too), the synchronization control messages should go through several servers to reach to the other, resulting the delay.

B. Evaluation

1) Overhead: Our model holds advantage for control, thus the overhead is small. In this experiment, we change the data generation speed and see the overhead change. As shown in Figure 8, 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 9, the percentage that the receiver will eventually send recovery interest when receiving a sync interest is much higher when data generation frequency increase. However, Something New Interest remains the same ratio with Anything New in DRSTL.

2) Delay: Since control messages should go through multi-level servers in our design, in spite that the level is related to

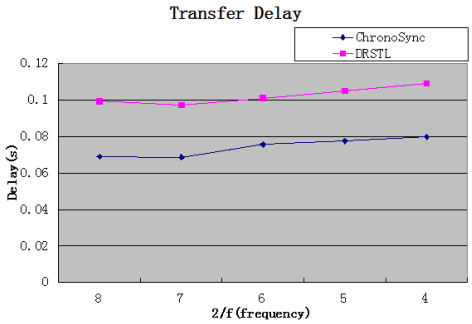

Fig. 10: Delay Compare

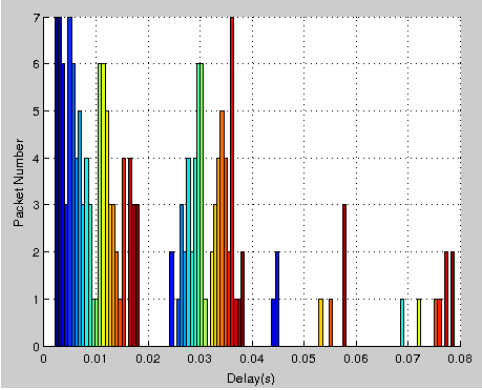

Fig. 11: Data Fetching Delay

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 10 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 11 shows the delay from a participant receives synchronization control message to when he gets the desired data. It is clear that this procedure is very fast, and it contributes to small delay and overhead.

IV. Related Work

Zhenkai Zhu et al recently introduced a new way to handle this problem on NDN, which is an entirely distributed and serverless protocol, named ChronoSync. In ChronoSync, status is represented using the digest of all participants' current status. Every participant keeps a digest tree and generates the root digest. If all the root digests are the same in the system, then obviously the system status is stable. To recovery from exceptions, every node keeps a digest log that contains the old digest and the trigger events. Everyone broadcasts a sync interest periodically to the system with its current root digest. Whoever receives a sync interest will compare the digest with his own root digest. If it is the same, he keeps it as a pending digest. If it is in his digest log, he sends back data containing the trigger events. On receiving the data, the receiver knows what data is missing, thus send interest for that piece of data.

Since ChronoSync is entirely distributed, and messages go along the optimized way, it almost has the smallest delay. The main downside of ChronoSync is short of control ability, resulting in problems to handle frequent simultaneous data generation. When this happens, the system will be divided into two different group and can't recognize digest from 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 should work fine, but in the case that the system is large, or data generation is frequent, video conference for example, there will be lots of simultaneous data generation.

V. Conclusion

In this paper, we proposed a new protocol to handle the synchronization problem in multi-user applications. Our design, Distributed Random Servers with Timed Labels, take advantages of both traditional server-based model and NDN's neat and efficient distributed features. (1) Multi-Level servers can provide enough control ability to handle complex conditions; (2) The essential distributed features allow it to fetch data efficiently with little overhead; (3) The hierarchy structure allows the server to control the synchronization by his local time label, which is easy and natural to process compared to using digest. We have implemented our protocol over ndnSIM and evaluated the performance. Data fetching is fast by a distributed way in NDN. 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. We will continue focusing on them in the future.

References

[1] 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.

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

[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.

[8] Johnson, T., & Shasha, D. (1994). X3: A Low Overhead High Performance Buffer Management Replacement Algorithm.