Design and Implementation of Not So Cooperative Caching System

HU Xiaoyan, GONG Jian, CHENG Guang, ZHANG Weiwei, Ahmad Jakalan

School of Computer Science and Engineering, Southeast University, Nanjing, China
Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education
Email: {xyhu, jgong, gcheng, wwzhang, ahmad}@njnet.edu.cn

Abstract: Not So Cooperative Caching (NSCC) considers a network comprised of selfish nodes; each is with caching capability and an objective of reducing its own access cost by fetching data from its local cache or from neighboring caches. These nodes would cooperate in caching and share cached content if and only if they each benefit. The challenges are to determine what objects to cache at each node and to implement the system in the context of Information Centric Networking (ICN). This work includes both a solution for the NSCC problem and a design and implementation of an NSCC system in Named Data Networking (NDN), a large effort that exemplifies ICN. Our design applies NDN synchronization protocol to facilitate the information exchange among nodes, adopts group key encryption to control data access within the NSCC group, and offers an error checker to detect error events in the system. Our approach is validated by deploying the system we developed on PlanetLab.

Keywords: named data networking; content caching; selfish caches; synchronization; group rekeying; error checker; game theory

I. INTRODUCTION

As a distinctive feature of Information Centric Networking (ICN), in-network caching plays a fundamental role on system performance. Existing works on optimizing in-network caching performance, e.g., the works in [1], [2], primarily consider a network with caches friendly to each other and coordinating to shoot for a globally beneficial object placement. However, there are scenarios where caches are selfish and aim at only their individual benefits. For example, future ICN is likely to be a network of autonomous systems (AS) with caching capability. And ASes that maintain peering agreements with each other may also engage in *content-level peering* in order to leverage each others' cached content to reduce their own data access costs [1] [3], [4], [5] (as transit traffic would be reduced). Not So Cooperative Caching (NSCC) is emerged to handle "cooperative caching" among such autonomous and selfish caches [6]. NSCC considers a network comprised of *selfish* nodes; each is with caching capability and *an objective of reducing its own access cost* by fetching data from its local cache or from neighboring caches. It assumes the access cost of retrieving data from local cache is minimal, and that of retrieving data from neighboring nodes is small as compared to that of fetching the data from its original content server. For example, fetching data from local cache or from a neighboring cache may reduce latency or reduce load on potentially expensive upstream links. As individual

1 Note that data access cost is abstract here. It can be capital expenditure, but it can also be other metrics such as data access delay.

nodes in NSCC are selfish and rational, they would choose to cooperate in making caching decisions and share cached content if and only if they each benefit from the system, which is the minimum participation requirement. The challenges are to determine what objects to cache at each node so as to satisfy the minimum participation requirement (resulting in a guaranteed global object placement) and to implement the system in an ICN way.

The previous work in [6] focuses on how to find a guaranteed global object placement in a game-theoretical way. And the work in [7] provides a basic implementation design of NSCC in NDN (Named Data Network), a large effort that exemplifies ICN. The basic implementation uses a four component design and a preliminary evaluation of the NSCC system is offered. As selfish and rational caches, NSCC nodes would like to share cached content with only the NSCC group members who contribute to the group so as to avoid the process of Interests from non-members. And thus there should be certain access control over the cached content acting as a disincentive for non-members to send Interests to the group to get rid of free-riders. Moreover, these selfish caches may unintentionally fail due to hardware or software or overload problems or unstable physical network connections, or intentionally refuse to answer some or all the requests from other members for data they commit to host to gain individual advantages at the cost of others (i.e., node cheating). And therefore, there should be some error checker mechanism to detect such error events.

For the data access control and error checker, the NSCC system introduced in [7] has not investigated them yet. In the present study, we elaborate on the design of the NSCC system and enhance the design with access control over content cached within the NSCC group and an error checker, an extra component to detect error events such as node failure, leaving, or cheating. Besides, this paper provides a more comprehensive evaluation of the enhanced NSCC system.

This paper includes the following companion supplementary work as compared to that in [7]: i) the extension of the synchronizer that adds distributed group key management (Section III-B), ii) the extension of the local cache that adds data encryption (Section III-D), iii) an extra error checker that timely detects error events (Section III-E), iv) an extra analysis and discussion of the NSCC system (Section IV), v) a more comprehensive evaluation that measures the communication overhead in synchronizing information among group members and the impact of error events on the caching performance of the enhanced NSCC system (Section V).

This work includes both a solution for the NSCC problem and a design and implementation of an NSCC system in Named Data Networking (NDN), a large effort that exemplifies ICN.

II. Problem Formulation of NSCC

We formally define the NSCC problem [6] as follows. We are given a set of n selfish caching nodes forming a "NSCC group", and a set of m unit-sized objects. The access pattern of node i is described by a vector $r_i = \{r_{i1}, \cdots, r_{ik}, \cdots, r_{im}\}$ where r_{ik} is the rate at which node i requests object k.

Each node aims to minimize its own access cost. When node i accesses an object, the cost depends on the object's location. Let $d_{i,j}$ denote the cost for node i to access an object cached at node j, $d_{i,i}$ denote the cost to fetch an object from its local cache and $d_{i,n+1}$ denote the cost for node i to fetch an object from original data sources. We assume $\forall i,j,\ d_{i,i}<d_{i,j}=d_{j,i}<d_{i,n+1}$, i.e., when a node accesses objects, local cache is preferred over other caches which are preferred over original data sources. The above definition of d is referred to as our access "price" model.

The cost of a node depends on where objects are placed and its access pattern. Due to cache space limitation, each node can cache only some objects locally and must decide which objects to place in its cache. Let S_i denote the cache size at node i ($S_i<m$) and P_i denote the set of objects cached at node i. Similarly, all other nodes decide which objects to place in their caches. The result is a global object placement $P = \{P_1, P_2, \cdots, P_n\}$. Then the cost of node i depends on the placement P. Let $C_i(P)$ denote the cost of node i under ob-

ject placement P which is computed as follow:

$$C_i(P) = \sum_{k \in P_i} r_{ik} d_{i,i} + \sum_{\substack{k \notin P_i \\ k \in Q_{-i}}} r_{ik} d_{i,l(i,k)} + \sum_{\substack{k \notin P_i \\ k \notin Q_{-i}}} r_{ik} d_{i,n+1} \quad (1)$$

The cost of node i is the total access cost to serve requests for all the objects. The cost for node i to serve requests for any object k is the product of its request rate and the cost for node i to fetch object k which depends on the location from which the object is fetched, either from local cache, or from the "cheapest" node that caches object k or from the original source. Let $Q_{-i}=P_1 \cup \cdots \cup P_{i-1} \cup P_{i+1} \cdots \cup P_n$ denote the set of objects collectively held by nodes other than node i under the global placement P and $d_{i,l(i,k)}$ denote the cost for node i to fetch object k from the "cheapest" node $l(i, k)$ that caches object k . More specifically, for each request for any object k, if object k is locally cached, it is accessed from local cache with cost $d_{i,i}$; otherwise if object k is cached at certain nodes in the NSCC group, it is accessed from the "cheapest" node $l(i, k)$ among these that store object k with cost $d_{i,l(i,k)}$; otherwise it is accessed from the original source with cost $d_{i,n+1}$.

Instead, if nodes are operate in isolation under Greedy Local policy (GL), objects at node i are sorted in descending order by their request rates and node i caches the S_i most popular objects. Then each object is accessed either from local cache or from its original source, and the cost of node i under GL is computed as follow:

$$C_i(GL) = \sum_{k \le S_i} r_{ik} d_{i,i} + \sum_{k > S_i} r_{ik} d_{i,n+1} \quad (2)$$

NSCC seeks a guaranteed object placement P such that for each node in the group, its cost would be reduced as compared to that under GL. And the objective is formulated as follow:

$$\forall i, C_i(P) < C_i(GL) \quad (3)$$

which is the participation or individual rationality constraint for each rational node.

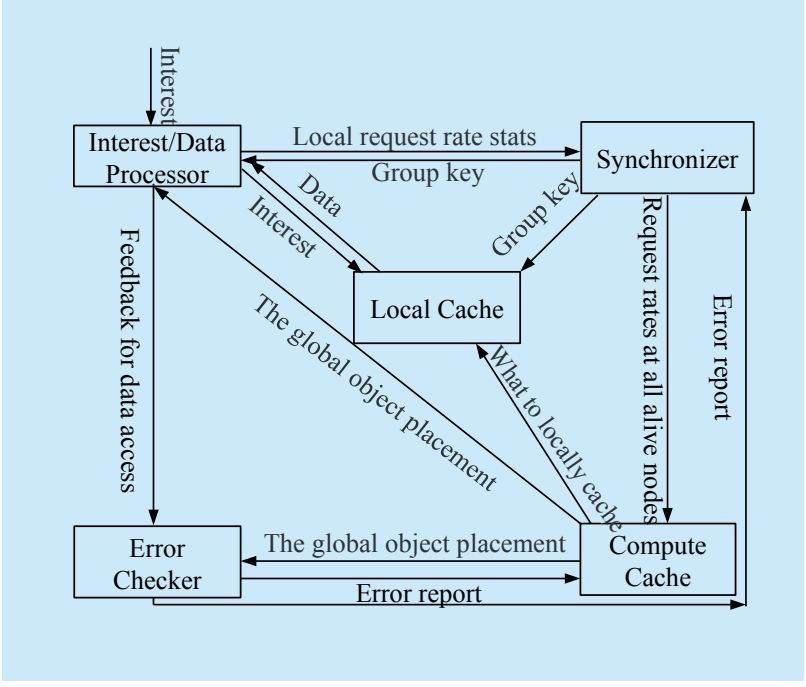


Fig.1 *The design of an NSCC node*

III. THE SYSTEM DESIGN

We designed a Not So Cooperative Caching system in NDN. Our NSCC design runs at the application level and makes use of the CCNx library [8]. An NSCC node is similar to a proxy and can be deployed by any organization. An organization simply configures routing so that the NSCC node is located on the path from its users to the Internet and its gateway is the best candidate to install the NSCC application. No changes are required to other NDN nodes or the underlying CCNx.

We offer a specific solution to the object placement problem in Section III-C. However, our design is independent of the object placement algorithm and allows one to use different algorithms for object placement, provided that the nodes agree upon the placement algorithm to be used.

In our design, each NSCC node consists of five components as illustrated in Figure 1. *Interest/Data Processor* processes users' requests for Data. *Synchronizer* periodically synchronizes request rate information at different NSCC nodes for global object placement computation and manages group key for the system. *Compute Cache* computes the global object placement at these NSCC nodes following our game-theory approach presented in [6]. *Local Cache* maintains Data objects specified by the Compute Cache and encrypts

such Data objects with the group key. *Error Checker* detects error events in the system. Each component is a "black box" to the others knowing nothing about how the other components do their jobs. These components interact with other components, if necessary, through their input and output interfaces.

3.1 Interest/data processor

We begin our discussion of NSCC with Interest/Data processor. Figure 2 illustrates the scenario under which the Interest/Data processor of an NSCC node works. The Interest/Data processor must meet the following four requirements:

- **Tracking Local Popularity**: the Interest/Data processor listens to all Interests from all local users and calculates local popularity of content.
- **Satisfying Interests From Local Users**: if an Interest is received from local users, the Interest/Data processor returns the data from local cache, or requests it from other NSCC node, or fetches it from the Internet.
- **Recording Users' Data Access Experience**: users' data access within the NSCC group that times out would be recorded so that error checker can investigate what happens.
- **Processing Interests Not From Local Users**: besides the Interests from local users, other Interests may be from the synchronizers of local node and other NSCC nodes, or from other NSCC nodes requesting data that may be cached at this node.

To track local popularity, the NSCC node is located on the default path from its users to the Internet. Its Interest/Data processor installs a route for the root name prefix pointing to the face of the NSCC application itself at this node. Then each Interest (except the Interests sent from the Interest/Data processor itself) would be forwarded to the NSCC application. A received Interest might come from local users, or other NSCC nodes, or from the synchronizers. An NSCC node only needs to track local content popularity and hence only the Interests from local users would be used for tracking local popularity. Interests from either other NSCC nodes or from the synchronizers begin with known common prefixes and thus can easily be distinguished from the Interests sent by local users. This work introduces a space efficient method with *double Counting Bloom Filter* (CCBF)[9], [10] to identify popular content that may be cached later and whose request rate information needs to be exchanged in the synchronizer. Figure 3 illustrates how to identify popular content with CCBF. The PopularData_CBF is a CBF used to test if a Data packet has already been filtered to be popular and record the access times of such Data. And Filter_CBF is another CBF used to filter popular Data. When an Interest for Data with name *ID* arrives, *k* different hash functions are used to map *ID* into *k* different counters and its following process is as follow:

- If the values of the *k* counters in PopularData_CBF are all larger than 0, the Data has already been filtered to be popular and then update the record of its corresponding access times.
- Otherwise, the requested Data is not yet popular enough and is being filtering. The values of the *k* counters in Filter_CBF are increased by 1. If the values of these *k* counters now are all larger than the preconfigured threshold *x* -- the definition of popular content, Data *ID* is filtered to be popular at this time. Then decrease the values of

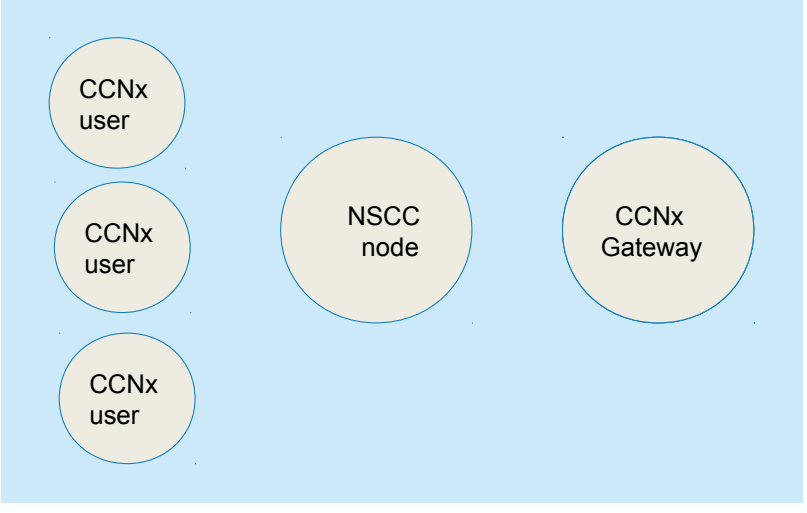


Fig.2 *The NSCC scenario*

the *k* counters for Data *ID* in Filter_CBF by the threshold *x*, increase the value of the *k* counters for Data ID in PopularData_CBF by 1, and create a popular Data record for Data *ID*.

Our application of CCBF is similar as the algorithm in [10] which applies double Counting Bloom filter to identify large flows in backbone networks, analyzes its false positive, and proves its effectiveness and space efficiency with extensive experiments. As the "cooperative caching'' is only about filtered popular content and only the request rate information of filtered popular content would be synchronized among NSCC nodes, the scalability issue of the NSCC system is mitigated.

There are the three following possibilities of satisfying Interests from local users:

1) The Interest/Data processor consults its local cache to see if the requested data is at local cache. If it is present, the Interest/Data processor returns it to the requester and must not send the Interest anywhere else. To ensure that the Interest would not be sent anywhere else, the route for this data name points to only this NSCC application.

2) Otherwise if the requested data is covered by another NSCC node, the Interest/Data processor must request the data from that node and send it back to the requester, but must not send the Interest anywhere else. The solution to request data from other nodes is achieved by appending an NSCC common name prefix in front of the Interest name and setting up a route for this new name pointing to the default gateway. In this fashion, the resulting Interest will only be sent to members in the NSCC group. The Interest/Data processor at each NSCC node installs a route for the NSCC common name prefix pointing to itself such that it would receive the Interests sent from other NSCC nodes requesting data that may be cached at this node. Due to error events, users' access for content covered by other NSCC nodes may time out. Interest/Data processor records the content names and the times of such failed data access.

3) If the requested data is not covered by any NSCC node, it should be fetched from the Internet. The Interest/Data processor installs a route for the original data name pointing to the default gateway. When the data returns, the Interest/Data processor simply forwards it back to the requesters. Neither local users nor gateway need to know NSCC also receives a copy of the Interest.

To process Interests not from local users, the Interest/Data processor listens for Interests from other NSCC nodes or from the synchronizers. It might receive Interests for data that it has cached. All such Interest names must start with the NSCC common name prefix. The Interest/Data processor strips off the NSCC common prefix, searches its local cache, and returns the data with name */NSCC/group/original name* if present in the local cache or ignores the Interest if it doesn't have the data or the Interest is from the synchronizers.

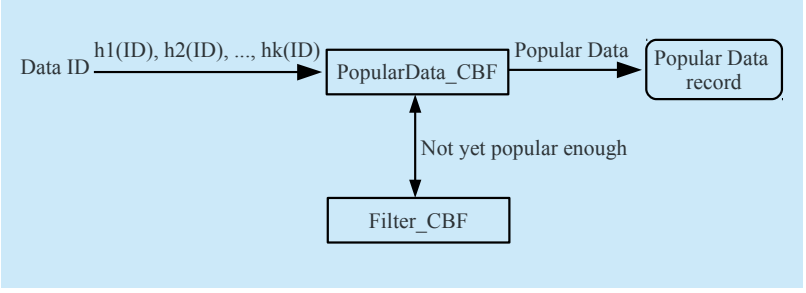


Fig.3 *Popular content identification with CCBF*

3.2 Synchronizer

Given the local popularity computed by the Interest/Data processor, the synchronizer is responsible for synchronizing request rate with other NSCC nodes. In other words, the synchronizer reports local popularity to other nodes and learns what content is popular at other nodes. There are the following four requirements for the synchronizer:

- **Fetching Local Request Rate Information**: the request rate information is obtained from the Interest/Data processor.
- **Request Rate Synchronization**: it must maintain an identical view of shared request rate data set all the time and changes in the request rate data set are reported to

the compute cache which then decides what data should be cached at each node.

- **Membership Maintenance**: it maintains a roster of participants. The events of node leaving or joining the group should be notified to all live nodes in the group so that they can make right caching decisions considering the caching in all alive group members.
- **Group Key Management**: the access of content cached within the NSCC group is protected by group key. Whenever the roster of participants changes due to node joining, leaving or failure, group rekeying should be triggered so that the access of content cached within the NSCC group can be confined to currently alive members.

To fetch local request rate information, the synchronizer simply invokes an *Application Program Interface* (API) to read the popularity records managed by the Interest/Data processor. The request rate synchronization and membership maintenance are important for the system. This information serves as the input of the object placement algorithm, discussed later in Section III-C. The synchronizer must obtain data from all other nodes and this data should be consistent with that at other nodes. If any NSCC node has a wrong roster of NSCC nodes in the group or has a wrong view of the request rate at another node, compute cache could make a wrong decision about the global object placement.

Given that any NSCC members can send Data packets to other members, they must be trusted equally and there seems no reason to ask certain member(s) to take more work such as the group key management. It is desirable that all members involve in the group key management. The group key management of the NSCC system follows the distributed group key management scheme for secure many-to-many group communication presented in [11]. Each member is assigned a binary ID when it joins the NSCC group by its neighbor who authenticates the node before its joining and both the joining node and its neighbor are responsible for generating a secret key for themselves. All secret keys are associated with their blinded versions, which are computed using a oneway function [12]. Members are represented by the leaves of a strictly binary key distribution tree and the key of each internal tree node is calculated as a mixing function [12] of the blinded keys of its two children. To compute the root key of the tree, i.e., the group key, each member holds all the unblinded keys of nodes that are on its path to the root and the blinded keys of nodes that are siblings of the nodes on its path to the root. Each blinded key is supplied by a different member of its key association group which is defined by its ID and delegates the task of key distribution evenly among all the members. While an unblinded key is computed by the node itself based on the blinded keys received from members in its key association and should not be exchanged with others. Contribution of a unique secret toward the computation of the root key gives each member partial control over the group. Please refer to [11] for the detailed group key distribution among members in a key association when there is node joining or leaving.

For the membership maintenance and blinded key synchronization within a key as-

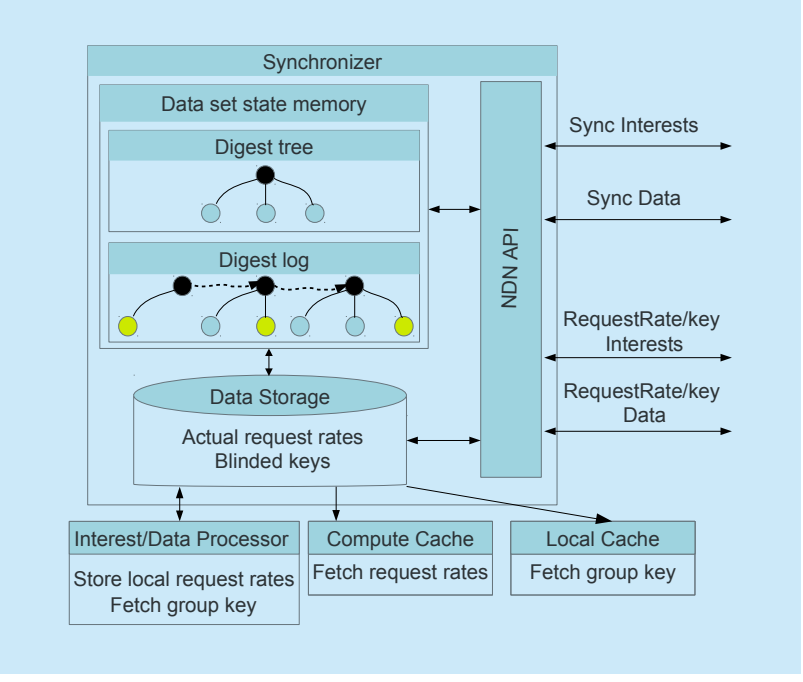


Fig.4 *The design of the synchronizer*

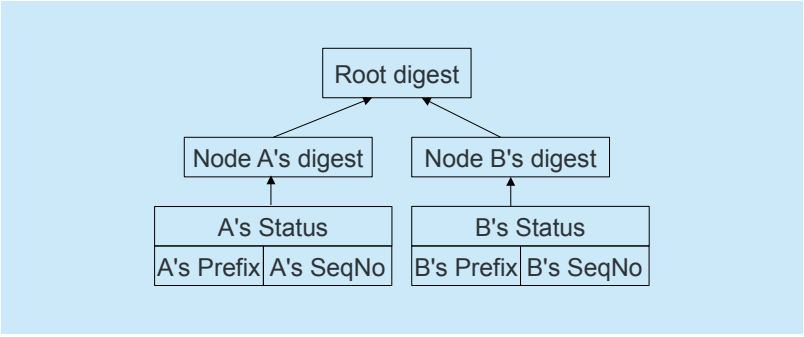


Fig.5 *An example of digest tree*

sociation group for group key management, the synchronizer is designed with a distributed data synchronization idea based on NDN SYNC protocol [13], [14]. The SYNC protocol offers reliable data synchronization and takes full advantage of the self-identifying nature of content and NDN's natural support of multicast. Figure 4 illustrates the design of the synchronizer with two main components: data set state memory and data storage (SYNC slice). The data set state memory maintains the current knowledge of the set of request rate information and blinded keys in the form of digest tree, as well as maintains history of the data set changes in the form of digest log. In the synchronizer, actual data, either request rate data or blinded keys data, is named as */ndn/nodeName/NSCC/group/seq*. There is a data type field in the Data packets that specifies the data as request rate information or blinded keys. Inspired by the idea of Merkle trees [15], digest tree is used to organize the participant statuses for quick and deterministic digest generation as illustrated in Figure 5. The digest tree of the synchronizer at each NSCC node is always kept up-to-date to accurately reflect the current state of the data set. Whenever an NSCC node sends a new Data packet (either about request rate information or about blinded keys) or learns about the name of a new Data packet from another NSCC node, the corresponding branch of the digest tree is updated and the state digest is re-calculated. The synchronizers interact using two types of Interest/Data message exchanges: synchronization (sync) and actual request rate or key data. A sync Interest represents the sender's knowledge of the current data set in the form of cryptographic digest, obtained using digest tree. To detect data set changes as soon as possible, every participant keeps an outstanding sync interest with the current state digest to the broadcast namespace of these synchronizers, */ndn/broadcast/NSCC/group*. The synchronizers at other NSCC nodes would receive the sync Interest. When all participants have the same knowledge about the data set, the system is in a stable state, and sync interest from each member carries an identical state digest. As soon as some NSCC node generates new data, the state digest changes, the outstanding interests get satisfied and then the data sets at these NSCC nodes are synchronized again. Common state and knowledge difference discovery is performed using the digest log. The digest log is a list of key-value pairs, where the key is the root digest and the value field contains the new participant statuses that have been updated since the previous state. As soon as a sync Interest discovers new knowledge about the data set state, the sender of the sync Interest sends out request rate or key Interests to pull actual request rate or blind key information. Any synchronizer that has the new data can satisfy the Interests, which offers reliable synchronization of request rates information and blinded keys.

For the management of the roster, an NSCC node is added to the roster when its presence message to the group is received. The participants periodically send "heartbeat" messages if they are in the group. If nothing is heard from an NSCC node for a certain amount of time, the NSCC node is no longer considered as a current participant of the group and the group key requires rekeying.

For the group key rekeying, whenever a new secret key occurs due to node joining or leaving, its corresponding blinded version would be stored in local SYNC slice, which triggers the rekeying process. Then subsequent sync Interests would discover the Data for the blinded key and the blinded key would be delivered to other members. Since any information exchanged

in the synchronizer is synchronized among all members, the blinded key exchange within a key association is protected by its corresponding unblinded key known only to the subgroup formed by the recipients. And the blinded key exchange between neighbors is protected by their public keys. Then only the direct neighbor can get the actual blinded version of the secret key since it is protected by the public key of the neighbor. After the two neighbors involved in the node joining or leaving exchange their blinded versions of their secret keys, they generate keys for internal nodes in the key tree and further exchange with other members in their key associations.

3.3 Compute Cache

The synchronizer provides the compute cache with a view of object popularity at all the participants. Then the compute cache starts a new round to determine a guaranteed global object placement. The resulting global object placement tells the local cache what it should contain and the Interest/Data processor what data is covered by other NSCC nodes. The computation of a guaranteed global object placement in the compute cache follows our game theory approach presented in [6]. In this approach, these NSCC nodes are sorted in ascending order by their names. With the global view of request rates information at all NSCC nodes, the compute cache first assumes that each node caches the most top popular objects following their cache size constraints as the initial global object placement. Then simulate that these nodes iteratively play the following game to seek a guaranteed global object placement. During each iteration, NSCC nodes compute their object placements (i.e., best responses) one by one according to their order. The best response is defined as follow:

Definition 1: (Best Response) Given a residual placement $P_{-i}=\{P_1, P_2, ..., P_{i-1}, P_{i+1}, ..., P_n\}$, the best response for node i is the placement $P_i \in A_i$ such that $C_i(P_{-i} + \{P_i\}) \leq C_i(P_{-i} + \{P_i'\})$, $\forall P_i' \in A_i$, $P_i' \neq P_i$ where A_i is the set of all the possible object placements at node i.

For the computation of a best response at a node, the excess gain of the node caching an object is computed based on whether the object is cached at other nodes from the current intermediate global object placement. $g_{ik}(P_{-i})$ denotes the excess gain incurred by node i from caching object k under the residual placement P_{-i} and is defined as follow:

$$g_{ik}(P_{-i}) = \begin{cases} r_{ik}(d_{i,n+1} - d_{i,i}) & \text{for } k \notin Q_{-i}, \\ r_{ik}(d_{i,l(i,k)} - d_{i,i}) & \text{for } k \in Q_{-i}. \end{cases} \quad (4)$$

The best response at node i under $P-i$ is computed as follow: objects are sorted in descending order by $g_{ik}(P_{-i})$ and the top S_i objects are selected to cache.

The iteration stops when no node wants to change its object placement. In this way, a guaranteed global object placement is found such that each node benefits. For the detailed description of the algorithm, please refer to the work in [6]. Note that different algorithms for the NSCC problem can be configured in the compute cache in the future without interfering with other components.

3.4 Local cache

The compute cache tells the local cache what data the cache should contain. The local cache fetches the data from the publishers and stores them in the local cache. Whenever the local cache needs to fetch data, a route for the corresponding Interest name pointing to the gateway should be installed and then uninstalled when the data is returned. As designed in the Interest/Data processor, the access to data covered by another NSCC node is obtained by prepending a common prefix specific to the NSCC group. For example, to fetch */seu/cs/hu/note.txt*, the Interest/Data processor will send a new Interest */NSCC/group/seu/cs/hu/note.txt*. To answer this Interest, the local cache should create a new Data packet with the name */NSCC/group/seu/cs/hu/note.txt*. In addition, to limit the access of data cached within the NSCC group to only group members, the data should be encrypted with the group key before being used to response its requests from the group. The local cache is responsible for fetch-

ing the group key from the synchronizer and encrypting the data to be locally cached. As a result, the original Data packet is first encrypted with the group key. Then the group key name and the encrypted data together serve as the content field of the new Data packet whose name starts with the common prefix. And the new Data packet is signed by the node that caches it. Figure 6 illustrates the format of a Data packet with encrypted payload.

Moreover, to reduce response latency, the local cache generates corresponding new Data packet for each Data that it should host. Then it can reply the requests for the Data from other members directly without invoking data generation process (including time consuming signing and encryption) repeatedly. Note that the original Data packet should be extracted and decrypted with the group key specified by the encryption key information in the Data packet before it is sent back to its requesters, which is performed by the Interest/Data processor.

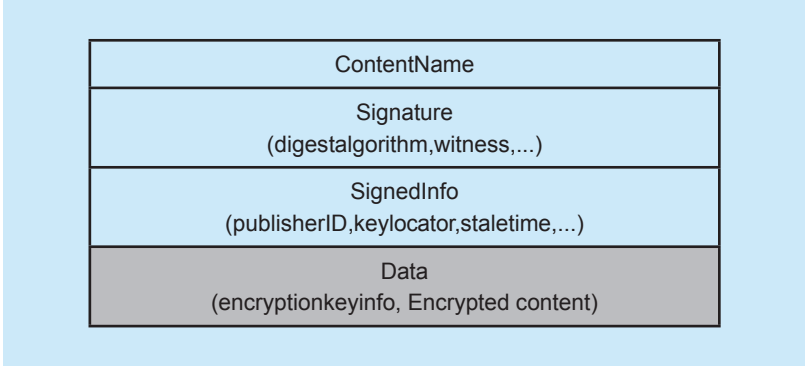


Fig.6 *A Data packet with encrypted content*

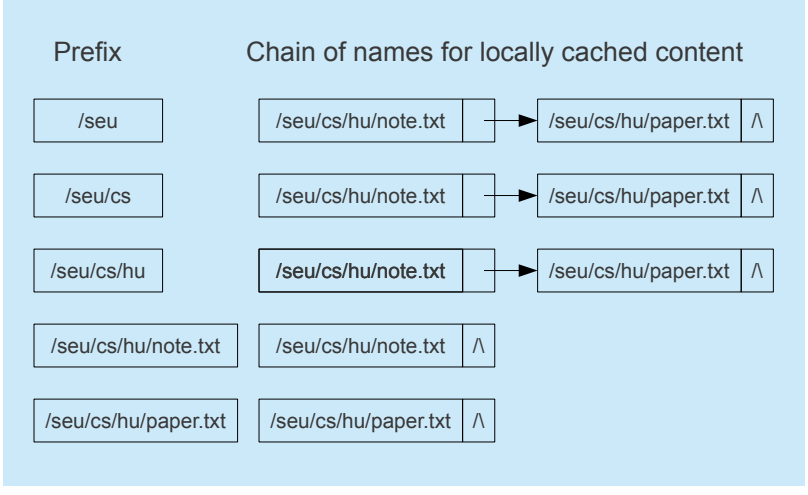


Fig.7 *The organization of indexing of cached content*

Besides, there is another design about how to record the cached data to facilitate the data lookup process. In NDN, an Interest can be satisfied by a Data with name equal to or more specific than the name specified in the Interest. For example, the Data with name */seu/cs/hu/note.txt* can satisfy an Interest with name */seu/cs/hu* or */seu/cs/hu/note.txt*. So the local cache organizes the names of locally cached content under a common prefix as a chain as illustrated in Figure 7. Upon the arrival of an Interest, its name is used as key to search corresponding chain. If the chain is present, cached content is found to satisfy the Interest.

3.5 Error checker

In practice, an NSCC node may unintentionally fail due to its hardware or software or overload problem or the physical network connection. Or a node may leave the group without explicit notification. Or an NSCC node may intentionally refuse to answer some or all the requests from other members for data it commits to host. Such refusal reduces overhead in replying others' requests at the expense of other members, and its sharing of data cached at other group members gains individual advantages, i.e., a cheating and so selfish behavior. We term such three types of events as error events in NSCC. The eventual outcomes of such error events are that other members may suffer from Interest timing out and have to fetch the requested data from content publishers.

The mission of the error checker is to discover error events in a timely manner and notify the synchronizer of evicting the initiators of detected error events from the NSCC group and group rekeying. The error checker keeps the 95% confidence interval (which is configurable) of the frequency of Interest timing out due to normal network instability (obtained from data access history or set by local operator). It periodically accesses the data access failure experience from the Interest/

Data processor. When the actual frequency of Interest timing out is not within the confidence interval, the error checker starts to investigate. From the global object placement obtained from the compute cache and the Interest timing out record from the Interest/Data processor, suspects are discovered. For each suspect, the error checker first consults the synchronizer to see if the suspect is still in the roster of the group. If the answer is no, the suspect has already left the NSCC group. Otherwise, the error checker sends background requests for data cached only by this suspectable node and sees how often such requests would time out. If this frequency of Interest timing out is still not within the confidence interval, the suspect is considered to be abnormal and its abnormity is notified to other nodes in the group. Then the abnormal node is evicted from the NSCC group. The eviction of an abnormal node is realized by triggering group rekeying so that the abnormal node cannot access data cached in the group any more.

IV. Analysis and Discussion

When new request rates information is reported to the compute cache, it starts a new round of a decision process. Then each NSCC node adapts to the current access patterns and gains benefits from other nodes in terms of cache hit ratio improvement, i.e., access cost reduction. But NSCC also incurs overhead resulting from the request rate synchronization, the object placement decisions, data encryption, and serving requests for data locally cached from other NSCC nodes. The request rate update frequency determines the overhead in the information synchronization, object placement decisions, and data encryption. A balance should be made between the gain in adapting caching to users' access patterns and the overhead in computation and communication by adjusting the request rate update frequency.

Due to space limitation, we concisely analyze the time and space complexity of the NSCC system. In the request rate synchronization, an NSCC node sends its request rates information to other nodes and fetches that at other nodes. Both the time and space that the synchronization takes are $O(nm)$. In the object placement decisions, the compute cache first computes the initial global object placement which takes time $O(nmlog(m))$. Then during each iteration, for each node, the computation of the excess gains and best response takes time $O(m)$ and $O(mlog(m))$ separately. So the object placement decisions making takes time $O(nmlog(m)+(m+mlog(m))nN) = O(nmNlog(m))$ where N is the number of iterations that the decisions making takes. The space consumption of the object placement decisions consists of three parts – the access price model, the global object placement and the excess gains which take space $O(n^2)$, $O(nm)$ and m respectively. So the object placement decisions making takes space $O(n^2+nm+m)$. In the data encryption, the local cache encrypts locally cached data, both the time and space which takes are $O(m)$. The error checker normally consults the Interest/Data processor for the frequency of Interest timing out, which takes time and space $O(1)$. If an error event occurs, the time for sending background request traffic would be $O((n-1)M)$ where M is the number of background requests for each suspect. And the space consumption would be the record of the frequency of background Interest timing out for all suspects, which is $O(n-1)$. In the group key management, the group key computation at an NSCC node requires its secret key and $log(n)$ blinded keys on its path to the root node of the key distribution tree. So for group rekeying, the synchronizer takes time and space $O(1+nlog(n)) = O(nlog(n))$.

V. Experimental Evaluation

This section evaluates the communication overhead in synchronizing information among group members and the impact of error events on the caching performance of the designed NSCC system.

5.1 Experimental setup

We conducted a number of experiments by

deploying the NSCC system on PlanetLab [16]. Figure 8 illustrates the experiment setup for the NSCC system. We installed CCNx library on seven PlanetLab nodes. Three nodes run our NSCC application, and also run NDN traffic generator application that simulates Interests sent by local users. An additional three gateway nodes are responsible for forwarding Interests to other gateways or the content server when necessary. At each gateway, routes pointing to other gateways or the content server are set up with the ccndc [17] tool. The seventh node is the content server node representing the rest of the network. If an NSCC node is unable to fetch data from any NSCC node, the content server provides the data.

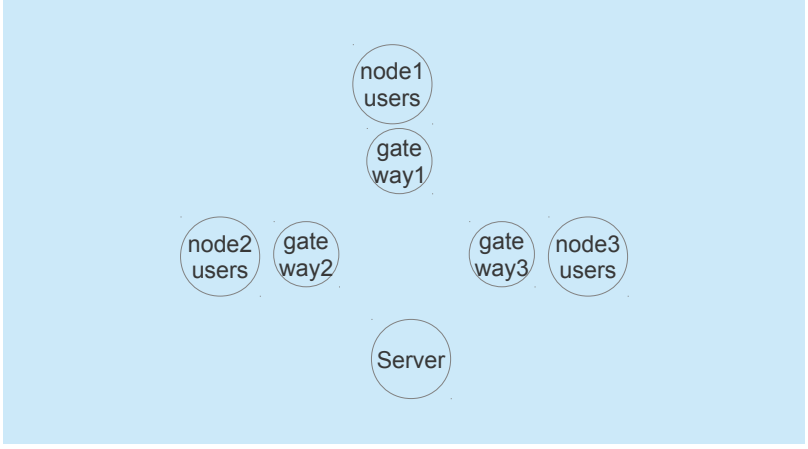


Fig.8 *The deployment of experiments*

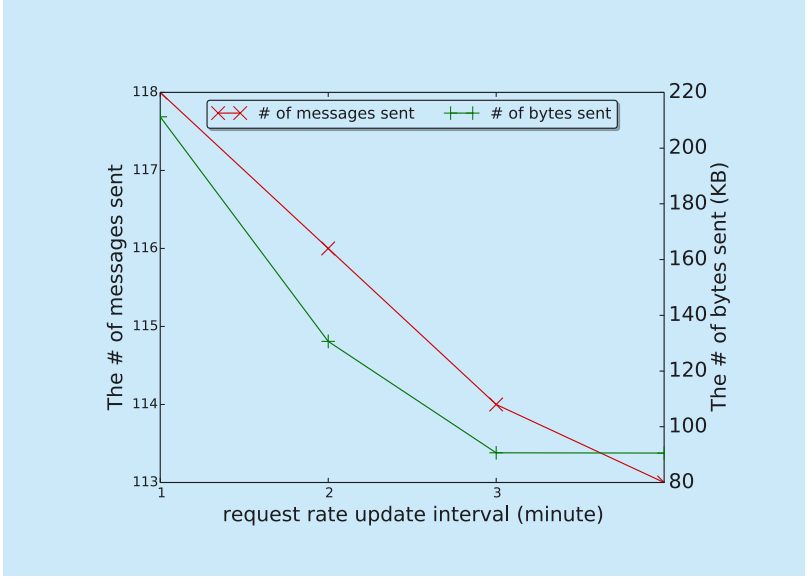


Fig.9 *The overhead in the synchronizer versus the request rate update frequency*

We assume there are 1000 unit-sized objects in the system and the access pattern at any node i follows a Zipf distribution with exponent s (Zipf preference). The Zipf distribution has been shown to be a good model for the popularity of web objects [18]. The three NSCC nodes are set with a caching capacity of 100 objects.

5.2 The impact of request rate update frequency

Since users' access patterns are dynamically changed, the synchronizers at NSCC group members periodically exchange request rates at which their users access content items. Such exchange triggers the compute cache process to adapt the caching to users' dynamic access patterns. In this subsection, we evaluate how request rate update frequency impacts the overhead spent in the synchronizer, and how the request rate update frequency and the dynamics of users' access patterns impact users' average cache hit ratio.

Group rekeying occurs only on demand. Without group rekeying happening, the overhead in the synchronizer comprises of the messages exchanged for the membership maintenance ("heartbeat" messages) and the messages exchanged for the request rate update. The overhead of the former is relative stable, while that of the latter relates to the request rate update frequency. To evaluate the impact of the request rate update frequency, we set different request rate update frequency (from every minute to every four minutes) for a five-minute experiment and then measure the number of messages and the number of bytes sent during the experiment.

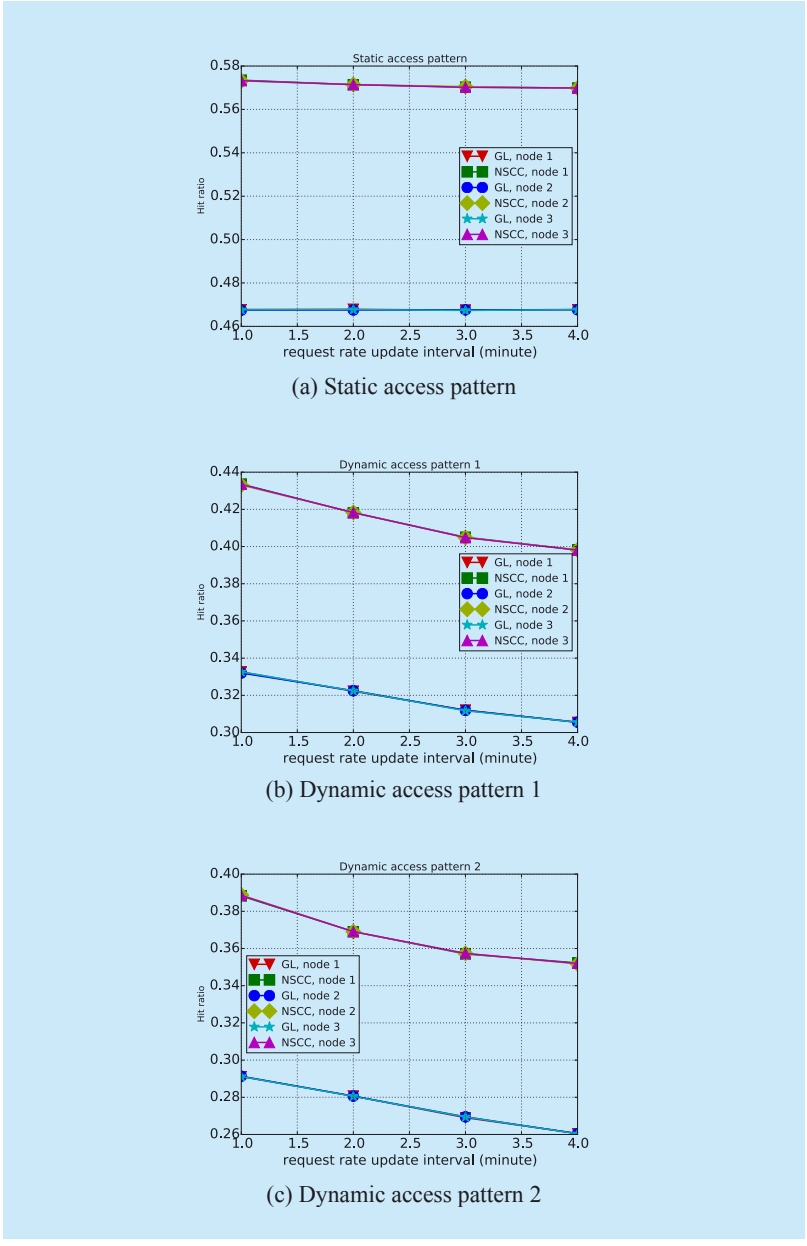
Figure 9 illustrates the corresponding results and each result is averaged from five runs of the experiments. It can be seen that as the request rate update interval increases (the request rates are updated less frequently), the average number of messages or the average number of bytes sent by each node decreases. But when the request rate update interval increases from three minutes to four minutes, the

difference in the number of bytes sent by each node is negligible. This is because that during the five-minute experiments, with three or four minutes as the request rate update interval, the request rates can be updated by only once, and the "heartbeat" messages for the two five-minute experiments should be almost the same.

Then we evaluate how the request rate update frequency and users' access patterns together impact the cache hit ratio of users requests. For the access patterns, we evaluate the following three cases:

- **Static access pattern**: during the five-minute experiments, the access pattern of users behind each NSCC node always follows the Zipf distribution with exponent 0.73 and the popularity ranking of objects does not changes.
- **Dynamic access pattern 1**: during the five-minute experiments, the access pattern of users behind each NSCC node always follows the Zipf distribution with exponent 0.73, but for each minute, the popularity ranking of objects cycle moves by 10, e.g., ranking changes from 0, 1, 2, ..., n-1 to n-10, n-9, ..., n-2, n-1, 0, 1,..., n-11.
- **Dynamic access pattern 2**: during the five-minute experiments, the access pattern of users behind each NSCC node always follows the Zipf distribution with exponent 0.73, but for each minute, the popularity ranking of objects cycle moves by 20.

Figure 10 illustrates the average cache hit ratio of requests at each node under the three users' access patterns when NSCC nodes are with different request rate update intervals (each result is averaged from five runs). As displayed, under our considered scenarios, NSCC achieves about a 0:10 improvement in cache hit ratio at each node against that if the three nodes operate in isolation (denoted as GL in Figure 10) and update their caching decisions with the same intervals. That said, NSCC offers global object placements that satisfy the minimum participation requirement, and treats these nodes fairly. From Figure 10(a), it can be seen that under the static access pattern, the request rate update frequency makes almost no difference on the average cache hit ratio. This is because the request rate information from statistic almost does not changes during the whole experiment and the exchange of the request rate information makes little difference in the caching decisions. So if operators of NSCC nodes are pretty sure that their users' access patterns are almost static, they can set the request rate update interval to be a large value, and even no request rate information update. Note that under the static access pattern, the cache hit ratio of each NSCC node does decrease a little bit as the request rate up-



(a) Static access pattern

(b) Dynamic access pattern 1

(c) Dynamic access pattern 2

Fig.10 *The average cache hit ratio versus the request rate update frequency under different user access patterns*

date interval increases. The reason for this is as follow. The initial caching is populated according to the ideal Zipf distribution with exponent 0.73. But the following caching decisions are based on the access patterns obtained from statistics which may be a little different from the ideal Zipf distribution due to the Pseudo-random number generator.

Figures 10(b) and 10(c) illustrate that under the two dynamic access patterns, as expected, the cache hit ratios at the three nodes are improved more if the NSCC nodes synchronize their request rate information more frequently (i.e., with smaller update interval). This is because the content caching adapts to the present access pattern more quickly. Moreover, comparing the cache hit ratios in Figure 10(b) with that in Figure 10(c), it can be seen that even with the same request rate update intervals, the cache hit ratio decreases when the access patterns are more dynamic. The reason for this is that the more dynamic access pattern requires more frequent request rate exchange to capture the dynamics and then being reflected in the content caching. But as shown in Figure 9, more frequent request rate information exchange generally implies more overhead in terms of messages and bytes sent by each node in the synchronizer. So a tradeoff should be made between the improvement in cache hit ratio and the overhead in synchronizing request rate information. For example, in our considered five-minute experiments, combining Figure 9 and Figures 10(b) and 10(c), an update interval of three minutes would be the tradeoff as if increasing the request rate update interval, the bytes sent by each node does not decrease, but the cache hit ratio does decrease.

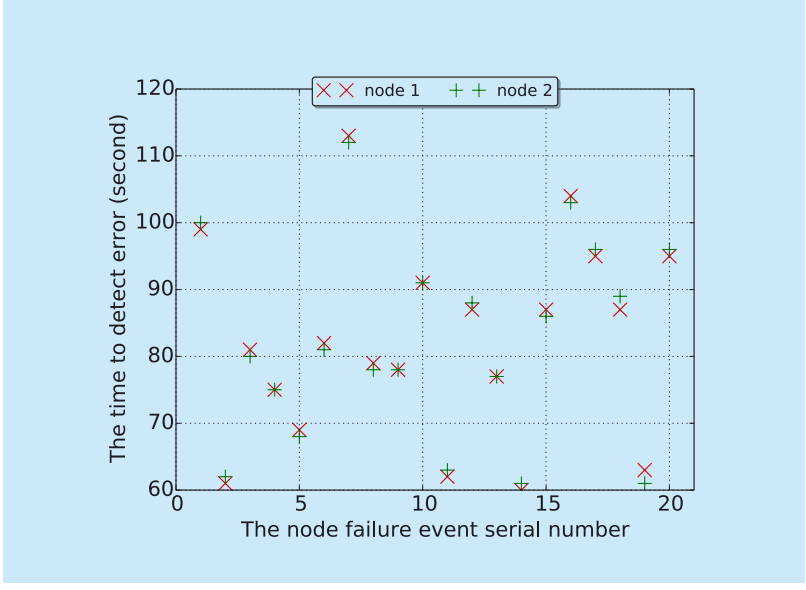


Fig.11 *The time of detecting node failure*

5.3 The impact of error events

As mentioned in Section III-E, the eventual outcomes of node failure, node leaving or node cheating are that the requests for content cached only at the node failed or leaving or cheating time out and are resent to the content server for the matching Data packets. With this in mind, to simplify the evaluation, we choose to evaluate the NSCC performance under node failure event representing that under three types of error events. To be specific, we measure how long it takes an alive NSCC node to detect the failure of another node and how a node failure impacts the content caching and the NSCC performance in terms of cache hit ratio and data access delay.

We emulate 20 trials of the node failure in the following way: during a five-minute experiment in which users' access patterns are static and follow Zipf distribution with exponent 0.73, we randomly choose a node to fail and leave the NSCC group at a time between the second and the third minute without explicit notification to the others. The left two alive nodes in the group may still send Interests for Data that they think are held only by the failed node until their error checkers detect the node failure. We measure how long it takes the two alive nodes to detect the node failure in these experiments and illustrate the results in Figure 11. When there is node failure, the caching belonging to the failed node does not contribute to the NSCC group any more. And thus the alive nodes need to re-distribute the content in their caches to adapt to the changes. To be specific, after the node failure is discovered, the synchronizers of the alive nodes distribute a new group key and synchronize their request rates. Then the compute cache starts a new

round to make the caching decisions for alive nodes. So the alive nodes have a new view of what are locally cached and what are cached at the other alive node, and access data following that view. We measure the changes of content caching (inserting or deleting content in the caches) at the two alive nodes due to the node failure and illustrate them in Figure 12. We also compare the average cache hit ratio as well as the average access delay of data access at the two alive nodes before the node failure with that during the node failure and not detected yet and with that after detecting the node failure, and illustrate them in Figure 13.

Seen from Figure 11, both nodes detect the node failures mostly within 60 to 100 seconds, less than 120 seconds, and the difference in the time spent by the two nodes detecting the node failures is negligible. For an NSCC node, its error checker consults the data access failure experience from its Interest/Data processor every once for a while within (45, 60) seconds (configurable) and the background traffic for testing a suspect lasts for 60 seconds (configurable). And thus the node failures should be detected within 120 seconds. When an NSCC node fails, a large number of Interests timing out would be detected by the error checkers of the two alive nodes. And such abnormity is further corroborated by background data access traffic. Once the error checker of an alive node identifies the node failure, it notifies the other alive node as soon as possible. The notification results in the small difference in the time that the two alive nodes spend on detecting the node failure. From Figure 12, it can be seen that the changes of content caching at the two alive nodes are within 24 to 36. Namely, as their caches are with a size of 100 objects, about 12 to 18 cache units need to be replaced by content newly downloaded from the content server. As shown in Figure 13(a), as expected, the cache hit ratio before the node failure is the largest (larger than 0.54), and that during the node failure and not detected yet is the smallest (smaller than 0.525). The reason

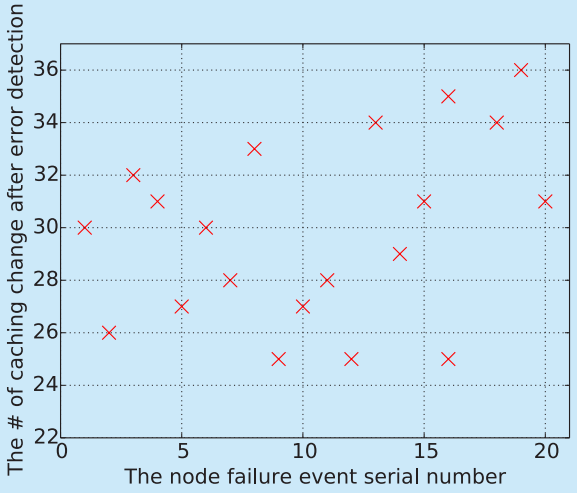
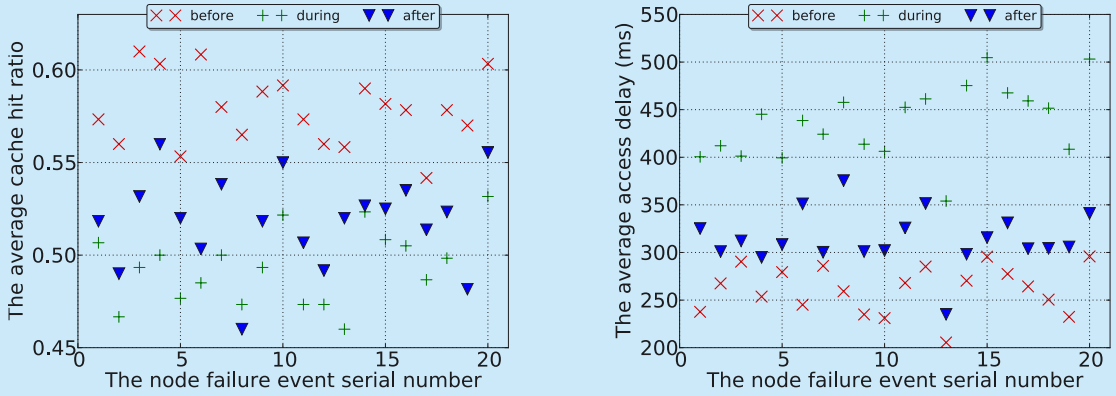


Fig.12 *The changes of content caching due to the node failure*



(a) Hit ratio

(b) Access delay

Fig.13 *The average cache hit ratio and access delay of alive nodes before the failure, during and after detecting the node failure*

for this is as follow. Before the node failure, the caching of the three nodes contributes to the group. While during the node failure and not detected yet and after detecting the node failure, only the caching of the two alive nodes contributes to the group. And the content caching after detecting the node failure adapts to users' access patterns better. From Figure 13(b), it can be seen that the access delay before the node failure is the smallest, and that during the node failure and not detected yet is the largest. The explanation for this is similar to that for the cache hit ratio, but also that during the node failure and not detected yet, some Interests are sent to the failed node, then time out and are resent to the content server, which greatly increases the average access delay. The above experimental results suggest that the performance degradation in terms of average cache hit ratio and access delay during the node failure and not detected yet is non-ignorable. And thus the error checker that can timely detects such node failures plays a fundamental role in the performance of the NSCC system.

VI. Conclusion

Our enhanced Not So Cooperative Caching (NSCC) system enables selfish and autonomous caching nodes in Named Data Networking (NDN) to cooperate in sharing cached data and making caching decisions in an Information Centric Networking (ICN) way. It considers a network comprised of selfish nodes requesting data in an ICN way; each is with caching capability and an objective of reducing its *own* access cost by fetching data from its local cache or from neighboring caches; and these selfish nodes may unintentionally fail or intentionally refuse to answer other members' requests to gain individual advantage. These nodes would cooperate in caching and share content if and only if they each gain benefits as compared to that when they operate in isolation.

This work includes both a solution for the NSCC problem and a design and implementation of an NSCC system. The previous work [7] provides a basic design of the NSCC system using a four-component design. This paper extends the basic design. Access control over data cached within the NSCC group is built into the system to free NSCC group members from receiving and replying Interests from non-members. And an extra error checker is offered so that the system can timely detect and recover from node failure, node leaving and node cheating. The data access control is implemented by distributing group key using NDN SYNC protocol and protecting data with the group key. Our system can be deployed at an organization without requiring any changes to other nodes at the organization or to the underlying CCNx library. Through deployment on PlanetLab, we explored the potential benefits of the enhanced NSCC system, the communication overhead of the system as well as the time that the error checker takes to discover error events and the extent of the system performance degradation due to such errors.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was sponsored by the National Grand Fundamental Research 973 program of China under Grant No.2009CB320505, the National Nature Science Foundation of China under Grant No. 60973123, the Technology Support Program (Industry) of Jiangsu under Grant No.BE2011173, and Prospective Research Project on Future Networks of Jiangsu Future Networks Innovation Institute under Grant No.BY2013095-5-03. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of those sponsors.

References

[1] J. Ren, W. Qi, C. Westphal, J. Wang, K. Lu, S. Liu, and S. Wang, "Magic: A distributed max-gain in-network caching strategy in information-cen-

tric networks," in *Proc. INFOCOM Workshops*, 2014, pp. 470–475.
[2] I. Psaras, W. K. Chai, and G. Pavlou, "Probabilistic in-network caching for information-centric networks," in *Proc. ICN'12*, 2012, pp. 55–60.
[3] J. Rajahalme, M. Sa¨rela¨, P. Nikander, and S. Tarkoma, "Incentive-compatible caching and peering in data-oriented networks," in *CoNEXT*, 2008, p. 62.
[4] S. DiBenedetto, C. Papadopoulos, and D. Massey, "Routing policies in named data networking," in *Proc. ICN'11*. New York, NY, USA: ACM, 2011, pp. 38–43.
[5] V. Pacifici and G. Dn, "Content-peering dynamics of autonomous caches in a content-centric network." in *Proc. INFOCOM*. IEEE, 2013, pp. 1079–1087.
[6] X. Hu and J. Gong, "Study on the theoretical framework of not so cooperative caching," *Journal of Internet Technology*, vol. 15, no. 3, pp. 351–362, 2014.
[7] X. Hu, C. Papadopoulos, J. Gong, and D. Massey, "Not so cooperative caching in named data networking," in *Globecom 2013 - Next Generation Networking Symposium (GC13 NGN)*, Atlanta, USA, Dec 2013.
[8] Ccnx project. Http://www.ccnx.org/.
[9] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.
[10] H. Wu, J. Gong, and W. Yang, "Algorithm based on double counter bloom filter for large flows identification," *Journal of Software*, vol. 21, no. 5, pp. 1115–1126, 2010.
[11] L. R. Dondeti, S. Mukherjee, and A. Samal, "A distributed group key management scheme for secure many-to-many communication," Department of Computer Science, University of Maryland, Tech. Rep. PINTL-TR-207-99, 1999.
[12] D. Balenson, D. McGrew, and A. Sherman, "Key management for large dynamic groups: One-way function trees and amortized initialization," *IETF Draft: draft-balenson-groupkeymgmt-oft-00.txt*, 1999.
[13] Sync protocol. Http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtoc [14] Z. Zhu and A. Afanasyev, "Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking," in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, Goettingen, Germany, October 2013.
[15] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238.
[16] Planetlab. Http://www.planet-lab.org/.
[17] Ccndc. Http://www.ccnx.org/releases/latest/doc/manpages/ccndc.1.html. [18] D. N. Serpanos, G. Karakostas, and W. H. Wolf, "Effective caching of web objects using zipf's law," in *IEEE International Conference on Multimedia and Expo (II)*, 2000, pp. 727–730.

Biographies

GONG Jian, is a professor in School of Computer Science and Engineering, Southeast University. His research interests are network architecture, network intrusion detection, and network management. He received his BS in computer software from Nanjing University, and his PhD in computer science and technology from Southeast University. Email: jgong@njnet.edu.cn

CHENG Guang, a professor of computer science at Southeast University, received his BS in Transportation Engineering from Southeast University in 1994, MS in Computer Science from Hefei University of Technology in 2000, and PhD in Computer Science from Southeast University in 2003. From 2006 to 2007, he was a post-doctor at School of Electrical and Computer Engineering in Georgia Institute of Technology. His current research interests include active measurement and traffic sampling in computer network. Email: gcheng@njnet.edu.cn

HU Xiaoyan, the corresponding author, email: xyhu@njnet.edu.cn. She focuses her research interests on information centric networking, in-network caching and scalable name-based routing. She received her BS in software engineering from Nanjing University of Science and Technology in 2007 and PhD in computer architecture from Southeast University in 2015. She visited netsec lab in Colorado State University, a research group working on NDN, from Sep. 2010 to Aug. 2012. Email: xyhu@njnet.edu.cn

ZHANG Weiwei, a PhD candidate in School of Computer Science and Engineering, Southeast University, focuses his research interests on computer network, network security and network management. He received his BS in software engineering from Southeast University, and MS in computer architecture from Southeast University. Email: wwzhang@njnet.edu.cn

Ahmad Jakalan, was born in Aleppo, Syria. Now he is a PhD Candidate in the School of Computer Science and Engineering, Southeast University, China. His research interests are network security, network intrusion detection, IP relationship discovery, and network traffic and host profiling. He received his BS in Informatics Engineering from Aleppo University, Aleppo, Syria in 2005 and MS in Computer Science and Technology from Southeast University. Email: ahmad@njnet.edu.cn