Not So Cooperative Caching in Named Data Networking

Xiaoyan Hu*, Christos Papadopoulos†, Jian Gong*, Daniel Massey†
*School of Computer Science and Engineering, Southeast University, Nanjing, P. R. China
†Computer Science Department, Colorado State University, Fort Collins, USA
Email: {xyhu, jgong}@njnet.edu.cn {christos, massey}@cs.colostate.edu

***Abstract*—This work designs and implements a Not So Cooperative Caching system for Information Centric Networking (ICN). We consider 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 sharing 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 Named Data Networking (NDN), a large effort that exemplifies ICN. Our results include both a solution for the Not So Cooperative Caching problem and an NDN design and implementation. We evaluate our approach by deploying the system we developed on PlanetLab and show that it improves the content hit ratio by up to 13%.**

I. Introduction

This work proposes a scheme that enables selfish nodes to cooperate in caching and shows how the scheme can be implemented in an information centric architecture, in particular *Named Data Networking* (NDN) [1] where caching is pervasive in network nodes. We consider 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. We assume the access cost of retrieving data from local cache is minimal, and that the cost of retrieving data from neighboring nodes is small as compared to fetching the data from an 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. The challenges are to *determine what objects to cache at each node (an object placement)* and to *implement the system in an information centric network.*

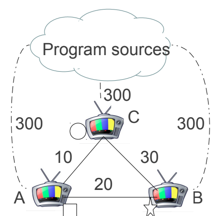


Fig. 1: Three set-boxes in a neighborhood.

Figure 1 shows how three set-top boxes, A, B, C can benefit from cooperation. The boxes record programs □, ★, ◯ separately. None of these boxes has the capacity to store all these programs, but the set-top boxes could cooperate in selecting what programs to record and then share the programs. If node A has not cached programs ★ and ◯, it is preferable for A to fetch them from B or C rather than from the original source. But an individual's set-top box is selfish and would choose to cooperate if and only if its own cost is reduced.

This work is similar to "Cooperative Caching" [2], [3], but dubbed *Not So Cooperative Caching* (NSCC) as each node aims to only maximize its own benefit rather than the common welfare. A node would join the cooperation if and only if its access cost will be reduced by at least some amounts so as to at least cancel out the overhead in cooperation, which is *the minimum participation requirement*. Nodes can freely join and leave the system based on whether their minimum participation requirement can be satisfied.

The two main problems of this work are as follows. First, we try to identify feasible object placements. We say an object placement is feasible if it satisfies the minimum participation requirements of all nodes in the current system. In other words, by cooperating in making caching decisions and sharing cached content, each node gains at least some minimum benefit. Second, we show how the scheme can be implemented in NDN, one paradigm of *Information Centric Networking* (ICN). We design a systematic model in NDN for nodes to exchange information for the object placement decisions and serve users' requests.

The rest of the paper is organized as follows. Section II formally defines the NSCC problem. Section III describes our design and implementation of a Not So Cooperative Caching application that can be deployed at an organization without requiring any modification to other nodes in the organization. Our NSCC nodes collaborate and solve the object caching as described in Section III-C. We deployed our system on PlanetLab and Section IV presents the preliminary results which verify the effectiveness of the NSCC system. We discuss the impact of the NSCC system on the network as well as the overhead of the system in Section V. Finally, Section VI concludes the work.

II. Not So Cooperative Caching

We formally define the NSCC problem here. We are given a set of n selfish nodes forming a "NSCC group", and a set of m unit-sized objects. The access pattern of node i is described by r_{ik}, i.e., the rate at which node i requests object k.

Each node aims to minimize its own access cost. Our access cost model follow the one defined in [4] and later used in several works [5], [6]. Under this model, the cost for accessing an object from a node's local cache is t_l, from another node in the group t_r and from an origin server t_s, with $t_l < t_r < t_s$. i.e., it is preferable for a node to access objects from neighboring caches rather than from original sources. These values may denote averages, and are assumed to be

978-1-4799-1353-4/13/$31.00 ©2013 IEEE

the same for all nodes in order to simplify the analysis.

The overall cost of a node depends on where objects are placed. Each node can cache some objects locally. Node i must decide which objects to place in its cache. Let S_i and P_i denote the cache size and the set of cached objects at node i respectively. Similarly, all other nodes decide which objects to place in their caches. The result is an object placement $P = \{P_1, P_2, \cdots, P_n\}$. The cost of node i depends on the placement P. Let $C_i(P)$ denote the cost of node i under object placement P and is computed as follows:

$$C_i(P) = \sum_{k \in P_i} r_{ik} t_l + \sum_{\substack{k \notin P_i \\ k \in P_{-i}}} r_{ik} t_r + \sum_{\substack{k \notin P_i \\ k \notin P_{-i}}} r_{ik} t_s \qquad (1)$$

The first term is the sum of costs for serving requests for objects locally cached at node i and the cost for serving such an object $k \in P_i$ is the product of its request rate r_{ik} and the local access cost t_l. The second term is the sum of cost for serving requests for objects cached in neighboring nodes in the group. Our terminology follows the one established in [5]. Let $P_{-i} = P_1 \cup \cdots \cup P_{i-1} \cup P_{i+1} \cdots \cup P_n$ denote the set of objects collectively held by nodes other than node i under the global placement P. The cost of serving requests for an object $k \notin P_i, k \in P_{-i}$ which is cached in neighboring nodes, is the product of its request rate r_{ik} and the access cost from this node i to the neighboring node that caches object k, t_r. The third term is the sum of costs for serving requests for objects neither cached locally nor cached in neighboring nodes. For serving such an object $k \notin P_i$ and $k \notin P_{-i}$, the cost is the product of its request rate r_{ik} and the access cost from the original source, t_s.

Given this definition, our objective is to 1) find a feasible object placement that reduces the individual node access costs by at least some minimum amount and 2) show how the scheme can be implemented in NDN.

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 [7]. An NSCC node is similar to a proxy and can be deployed by any organization. The organization simply configures routing so that the NSCC node is located on the path from users to the Internet. 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.

Each NSCC node consists of four components: *Interest/Data Processor* which processes users' requests for Data, *request rate synchronizer* which synchronizes the request rate information at NSCC nodes, *Compute Cache* which computes the global object placement at NSCC nodes, and *Local Cache* which is responsible for caching the objects specified by the Compute Cache. 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.

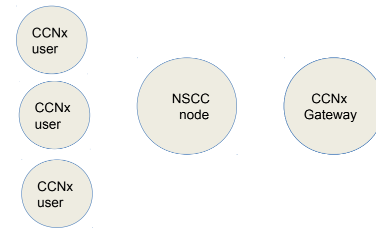


Fig. 2: The NSCC scenario.

A. Interest/Data Processor

We begin our discussion of NSCC with Interest/Data processor. As the name suggests, this component is responsible for processing Interest packets and tracking the local popularity of content. Figure 2 illustrates the scenario under which the Interest/Data processor of an NSCC node works. The Interest/Data processor must meet the following three 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.
- **Processing Interests Not From Local Users**: besides the Interests from local users, other Interests may be from request rate synchronizer of local node or 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 users to the Internet and its Interest/Data processor installs a route for the root name prefix pointing to the face of the NSCC application itself at this node such that 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 request rate synchronizer. 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 request rate synchronizer begin with known common prefixes and thus can easily be distinguished from the Interests sent by local users. And for tracking local popularity, this work introduces a space efficient method with *double Counting Bloom Filter* (CCBF) to identify popular content that may be cached later and whose request rate information needs to be exchanged in the request rate synchronizer. Please refer to [8] for the introduction of *Counting Bloom Filter* (CBF). Figure 3 illustrates how to identify popular content with CCBF. The PopularData_CBF is a CBF that is used to test if a Data has already been filtered to be popular and record the access times of such Data. And Filter_CBF is another CBF that is 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 its corresponding access times by increasing the values of the k counters in PopularData_CBF by 1.

- Otherwise, the requested Data is not yet popular enough and is being filtering for the final decision. This is done by increasing the values of the k counters in Filter_CBF by 1. If the values of these k counters in Filter_CBF are now 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 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.

CCBF is similar as the algorithm in [9] which applies double Counting Bloom filter to identify large flows in ackbone network, analyzes its false positive, and proves its effectiveness and space efficiency with extensive experiments.

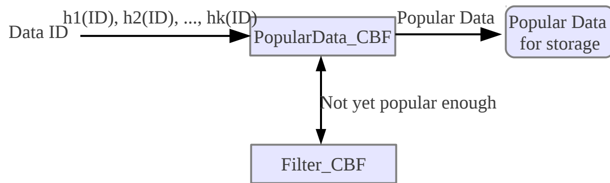


Fig. 3: Popular content identification with CCBF.

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 at local gateway 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 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.

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 that 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 request rate synchronizer. 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, returns the data with name $/NSCC_prefix/original_name$ if found in local cache or ignores the Interest if it doesn't have the data or the Interest is from request rate synchronizer.

B. Request Rate Synchronizer

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

- **Fetching Local Request Rate Information**: the request rate information is obtained from the Interest/Data processor component.
- **Request Rate Synchronization**: it must maintain an identical view of shared request rate data set all the time and changes in the request rate are reported to the compute cache component which then decides what data should be cached at each node.
- **Membership Maintenance**: it maintains a roster of participants. The events of nodes 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 group members.

To fetch local request rate information, request rate 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 to the object placement algorithm, discussed later in Section III-C. The request rate 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.

The requirements of request rate synchronization and membership maintenance are similar as that in traditional multi-person conference or multi-user chat. Many existing applications use a central-server based implementation, where every participant synchronizes its data with a centralized server. However, such designs lead to traffic concentrations at the server and make the applications vulnerable to the single point of failure. Zhu *et al.* evolves a distributed data synchronization idea to design a serverless multi-user chat over NDN [10] (based on SYNC protocol[1] which offers reliable data synchronization) taking full advantage of the self-identifying nature of content and NDN's nature support of multicast.

Figure 4 illustrates the design of the request rate synchronizer. The design of request rate synchronizer has two main components: data set state memory and data storage (SYNC slice) as illustrated in Figure 5. The data set state memory maintains the current knowledge of the request rate information set in the form of digest tree, as well as maintains history of the data set changes in the form of digest log. Request rate synchronizers interact using two types of Interest/Data message exchanges: synchronization (sync) and request rate data. A sync Interest represents the sender's knowledge of the current request rate data set in the form of cryptographic digest, obtained using digest tree, and is delivered to every other participant by periodically sending the locally generated sync Interest to the broadcast namespace

[1] http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html

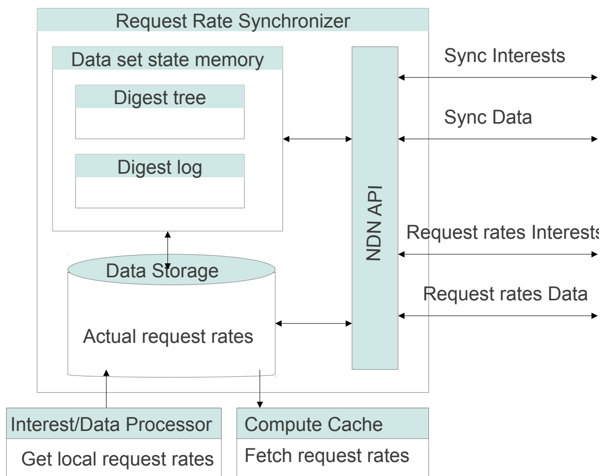


Fig. 4: Request rate synchronizer.

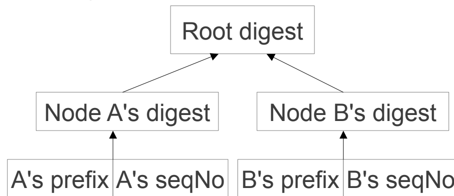


Fig. 5: Digest tree.

of these synchronizers, $/ndn/broadcast/NSCC/group$, such that the request rate synchronizers at other NSCC nodes would receive it. Any recipient of the sync Interest who detects that it has more information, satisfies this Interest with a Data packet that includes the names of the missing part of the data set and the actual data of request rate information is named under the namespace of $/ndn/nodename/NSCC/group/$ appended with corresponding sequence number. Common state and knowledge difference discovery is performed using the digest log. The log is a list of key-value pairs, where the key is the root digest and the value field contains the new producer statuses that have been updated since the previous state. As soon as any participant discovers new knowledge about the request rate information state, it sends out request rate Interests to pull actual request rate information and multicast helps the delivery of request rate information.

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.

C. Compute Cache

The request rate synchronizer provides compute cache with a view of object popularity at all the participants. The objective now is to start a new round to determine what objects should be cached locally telling the local cache what it should contain and infer what other nodes would cache so that the Interest/Data processor can determine if the requested data is covered by other NSCC nodes. For this specific component, we do not invent a new algorithm to find a feasible object placement, but borrow the game theory approach from Laoutaris *et al.* [5]. 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 component first computes the intermediate caching decisions at each node assuming these nodes do not cooperate at all. Namely, each node caches the most top popular objects following their cache size constraints. Then based on the intermediate caching decisions, the object placements at nodes (i.e., best responses) are decided one by one according to their order. At this step, the excess gain of a node caching an object is computed based on whether the object is cached at other nodes, and then each node greedily caches the most valuable objects as its best response which is defined as follow:

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

$g_{ik}(Q_{-i})$ denotes the excess gain incurred by node i from caching object k under the residual placement Q_{-i} and is defined as follow:

$$g_{ik}(Q_{-i}) = \begin{cases} r_{ik}(t_s - t_l) & \text{for } k \notin P_{-i}, \\ r_{ik}(t_r - t_l) & \text{for } k \in P_{-i}. \end{cases} \tag{2}$$

The best response at node i under Q_{-i} is computed as follow: objects are sorted in descending order by $g_{ik}(Q_{-i})$ and the S_i most valuable objects are selected to cache. In this way, Laoutaris *et al.* proves that a feasible global object placement is found such that each node benefits. For the detailed description of the algorithm, please refer to [5]. Note that different algorithms for the NSCC problem can be configured in the compute cache by the operators of the NSCC nodes in the future without interfering with other components.

D. Local Cache

The final component implements the local cache. The compute cache tells the local cache what the cache should contain and it 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 pointing to the gateway can be installed and then uninstalled when the data is returned.

As described in the Interest/Data processor component, access to data covered by another NSCC node is obtained by prepending a prefix specific to the NSCC group. For example, to fetch $/CSU/cs/hu/note.txt$, the Interest/Data processor will send a new Interest $/NSCC/group/CSU/cs/hu/note.txt$. To answer this Interest, the local cache should create a new Data packet with the name $/NSCC/group/CSU/cs/hu/note.txt$. The original $/CSU/cs/hu/note.txt$ Data packet becomes the content field and is signed by the NSCC node. To reduce response latency, the local cache component generates the corresponding new Data for each Data that it should hold so that it can reply the requests for the Data directly without invoking the data generation process repeatedly. Note that the actual Data should be extracted before it is sent back to the users and this is performed by the Interest/Data processor.

And 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 $/CSU/cs/hu/note.txt$ can satisfy an Interest with

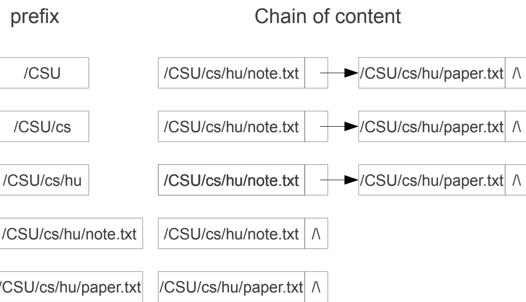


Fig. 6: The record of cached Data. A node in chain of content is a pointer to content with the specified name.

name $/CSU/cs/hu$ or $/CSU/cs/hu/note.txt$. So the local cache organizes the pointers to these objects under a common prefix as a chain as illustrated in Figure 6.

IV. Evaluation

We have implemented the NSCC system proposed in section III. In this section, we present the preliminary results of the empirical evaluation of the system. Our experimental results verify the effectiveness of the NSCC system.

A. Experiment Setup

We conducted a number of experiments by deploying the NSCC system on PlanetLab [11]. Figure 7 illustrates the experiment setup for not so cooperative caching. 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. The three nodes serve as the users who send request traffic as well as the NSCC nodes. 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 tool ccndc [12]). The seventh node is our content server node representing the rest of the network. If nodes are unable to fetch data from any NSCC node, the content server provides the data.

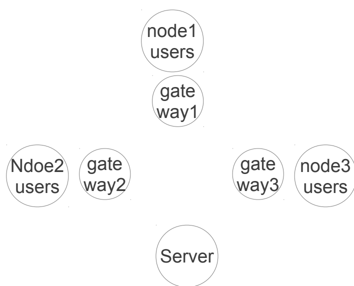


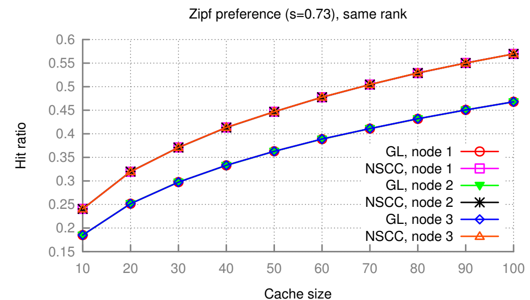
Fig. 7: The deployment of experiments.

We assume there are 1000 unit-sized objects in the system and the access pattern at any node follows a Zipf distribution with exponent s (Zipf preference). This has been shown to be a good model for the popularity of web objects [13]. To evaluate the impact of access patterns on the performance of NSCC, we consider the following two cases:

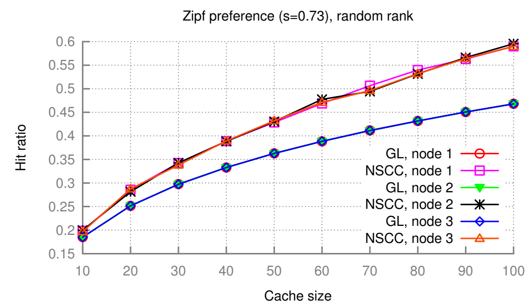
- Case 1: the access patterns at the three nodes follow Zipf distribution with a typical exponent 0.73 (see e.g., [14], [15]) and the rank of objects remains the same for all nodes.
- Case 2: the access patterns at the three nodes follow Zipf distribution with exponent 0.73, but the rank of objects at each node is randomly determined and thus they are different at these nodes.

We also consider the impact of cache sizes (in terms of the number of objects and these nodes are with the same cache sizes in each experiment) on the performance of the NSCC system and compare the hit ratio of each node under NSCC with that under *Greedy Policy* (GL), i.e., each node works independently without sharing and caches its most popular content, which is the most common caching algorithm for selfish caches [5] (traditional cooperative caching algorithms such as [3], [16] are improper here for selfish caches). In each experiment, the NDN traffic generator applications at the three NSCC nodes each send 100000 Interests to request content following their Zipf access patterns. In compute cache, we set $t_l = 0$, $t_r = 1$ and $t_s = 2$. The Interest/Data processor at each node additionally counts the hit ratio of its users' requests and each experiment is repeated for 100 runs.

B. Experiment Results



(a) Case 1.



(b) Case 2.

Fig. 8: The hit ratios of the three nodes under GL and under NSCC when cache sizes of nodes vary.

Figure 8(a) and Figure 8(b) illustrate how the cache capacity of nodes impacts the hit ratios of the three nodes under NSCC and that under GL when these nodes follow the above mentioned two cases of access patterns respectively. It can be seen that in both cases, all nodes have a similar yield (to some degree, the system offers fairness to these nodes) under NSCC as compared to under GL in terms of hit ratio and the improvement in hit ratio can be as much as 13%. And when nodes are with small cache sizes, if these nodes are with access patterns of case 1, their cooperation under NSCC would be more helpful (improving hit ratio by about 6%) as compared to that if they are with access patterns of

case 2 (improving only by about 1%). This is because the small caching space accommodates a small number of content. If these nodes follow the identical access patterns, the cached content tend to be useful for all these nodes. Otherwise, as the ranks of content at these nodes are different, each node tends to cache content only useful for itself due to its selfishness. Then the content cached at node 1 may be little popular at the other two. But as the cache sizes increase, more and more content can be cached in these nodes. Even if the ranks of content at these nodes are different, each node caches some content valuable for itself but also valuable for the others and can be shared by them.

V. Discussion

When new request rates information is reported to the compute cache component, compute cache starts a new round of a decision process. 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. As more users' request traffic is satisfied by these NSCC nodes, the network is less congested due to the relief from delivering these requests and their matching data between the requesters and their content servers, and related content servers are less overloaded by users' requests and thus can quickly response to others' requests. But NSCC also incurs overhead resulting from the request rate synchronization, the object placement decisions, and serving requests for data locally cached from other NSCC nodes. As the frequency that request rate synchronizer updates information determines the overhead in request rate synchronization and object placement decisions, a balance should be made between the gain in adapting access patterns and the overhead in computation and communication by adjusting the frequency of updating request rates information. And the CCBF for popular content identification at each node makes contribution to the reduction in space and communication cost. Our ongoing work is conducting more extensive larger-scale experiments on evaluating the overhead of the NSCC system and analyzing how to balance between the request rate synchronization frequency to quickly adapt to users' dynamic demand and the overhead involved.

VI. Conclusion & Future Work

This work proposes a scheme that enables selfish nodes to cooperate in caching (which we call *Not So Cooperative Caching* (NSCC)) and shows how the scheme can be implemented in *Named Data Networking* (NDN), a promising future Internet architecture, where caching is pervasive in network nodes. We consider 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 sharing content if and only if they each benefit.

We provide an implementation of the NSCC using a four component design. Our design includes a request rate synchronization component that coordinates membership in the NSCC group and ensures that nodes have a common view of object popularity distribution at different nodes. Our compute cache component solves the object placement and calculates what data should be stored in the local cache. The local cache is responsible for keeping and managing data. And the Interest/Data processor keeps track of local content popularity and processes Interests from local users or other NSCC nodes. Through deployment on PlanetLab, we explored the potential benefits of NSCC. Our system can be deployed at an organization without requiring any changes to other nodes at the organization and without any changes to the underlying CCNx library.

This work provides the foundation for a Not So Cooperative Caching system and our next step is to conduct extensive larger-scale experiments on the evaluation of the overhead of the system and add an Error Checker component to cope with cases where the selfish nodes unintentionally fail or intentionally misbehave to gain individual advantage.

Acknowledgment

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 and US NSF under Grant No.1039585. 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] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. Braynard, "Networking named content," in *CoNEXT*, 2009.

[2] J. Wang, "A survey of web caching schemes for the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 36–46, October 1999.

[3] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *Proceedings of the 29th conference on Information communications*, ser. INFOCOM'10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 1478–1486.

[4] A. Leff, J. L. Wolf, and P. S. Yu, "Replication algorithms in a remote caching architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 11, pp. 1185–1204, Nov. 1993.

[5] N. Laoutaris, O. Telelis, V. Zissimopoulos, and I. Stavrakakis, "Distributed selfish replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 12, pp. 1401–1413, Dec. 2006.

[6] E. Jaho, I. Koukoutsidis, I. Stavrakakis, and I. Jaho, "Cooperative content replication in networks with autonomous nodes," *Comput. Commun.*, vol. 35, no. 5, pp. 637–647, Mar. 2012.

[7] Ccnx project. Http://www.ccnx.org/.

[8] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[9] 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.

[10] Z. Zhu, C. Bian, A. Afanasyev, V. Jacobson, and L. Zhang, "Chronos: Serverless multi-user chat over ndn," NDN, Technical Report NDN-0008, October 2012.

[11] Planetlab. Http://www.planet-lab.org/.

[12] Ccndc. Http://www.ccnx.org/releases/latest/doc/manpages/ccndc.1.html.

[13] 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.

[14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM (1)*, 1999, pp. 126–134.

[15] H. Gomaa, G. Messier, R. Davies, and C. Williamson, "Media caching support for mobile transit clients," in *Proceedings of the 2009 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, ser. WIMOB '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–84.

[16] J. Ni and D. H. K. Tsang, "Large-Scale Cooperative Caching and Application-Level Multicast in Multimedia Content Delivery Networks," *IEEE Communications Magazine*, vol. 43, pp. 98–105, May 2005.