MuNCC: Multi-hop Neighborhood Collaborative Caching in Information Centric Networks

Travis Mick
New Mexico State University
tmick@cs.nmsu.edu

Reza Tourani
New Mexico State University
rtourani@cs.nmsu.edu

Satyajayant Misra*
New Mexico State University
misra@cs.nmsu.edu

ABSTRACT

Caching strategies in *Information-Centric Networks* (ICNs) can be classified into the categories of *individual caching*, *on-path caching*, and *collaborative caching*. Each has several drawbacks, such as high content redundancy in individual caching, unutilized caching capacity in on-path caching, and high coordination cost in collaborative caching. Despite the relatively higher cost of coordination, collaborative caching offers several advantages over the other categories, namely low latency and better cache utilization. A collaborative caching mechanism with low coordination costs that still possesses the inherent advantages has not been proposed in the literature.

In this paper, we aim to address this missing link by presenting MuNCC, a scalable collaborative caching scheme, which utilizes a node's neighborhood caching capacity and has negligible communication overhead. MuNCC uses attenuated Bloom filters, augmented by a two-level node cache structure and an efficient cache redundancy elimination technique. Exhaustive simulation-based comparison of MuNCC against the state-of-the-art shows that it reduces content retrieval latency by 30% to 40% while maintaining a high level of cache utilization and incurring low overheads.

Keywords: **In-network caching, collaborative caching, ICN, next generation network.**

1. INTRODUCTION

In-network caching is essential to ensure the scalability of the future Internet, and thus has been embraced within the Information-Centric Networking community. By storing content close to the users, the cost of its retrieval is reduced and user quality of experience is improved. However, cache utilization must be optimized to be effective in the face of a rapid explosion of Internet content. Since caching capacity is inherently limited, this leads us to two fundamental questions: *Which content to cache? And which content to evict?*

Proposed approaches to answering these questions can be divided into three categories: individual caching, on-path caching, and collaborative caching. In individual caching, each node makes these decisions independently, without input from others in the network. These schemes are simple, but achieve low cache-hit ratios. On-path caching attempts to optimize the placement of a content object on the path it travels between the provider and the consumer; though this outperforms individual caching, it still does not utilize off-path caches. In collaborative caching, nodes cooperate either locally or globally to determine the best content placement, optimized for either latency, network load, or cache utilization. Such collaboration invariably requires significant amounts of overhead, and can thus undermine network goodput.

In this paper, we address the drawbacks of current collaborative caching techniques and propose a scalable and efficient collaborative strategy. Our **contributions** include: ***(i)*** A design of a scalable cache location mechanism, through which nodes can be made aware of the content available in their neighborhoods. ***(ii)*** An augmentation of the least frequently used (LFU) replacement policy to allow nodes to evict redundantly cached content in a neighborhood without incurring additional overhead. ***(iii)*** Extensive simulations to show the significant improvement offered by our mechanism over the existing state-of-the-art.

In Section 2, we review the state of the art in ICN caching algorithms. Section 3 presents our system model and assumptions. We introduce the building blocks of our Multi-hop Neighborhood Collaborative Caching framework, ***MuNCC*** (pronounced "monk"), and the framework itself in Sections 4 and 5 respectively. In Section 6, we offer our simulation results and analyses. Finally, Section 7 offers some concluding remarks and a brief overview of future work.

2. RELATED WORK

The majority of work on caching in information-centric networking is dedicated to evaluating the performance of different caching schemes. In this section, we focus on collaborative caching schemes, classifying them into

*This work was supported in part by the US National Science Foundation grants 1241809 and 1345232 and US DoD grant 67311RTREP. The information reported here does not reflect the position or the policy of the federal government.

two categories based on how coordination is achieved: on-path coordination, and global coordination.

In on-path coordination, routers on the delivery path between a content provider and a consumer collaboratively decide where along that path each content should be cached. Leave Copy Down (LCD), Move Copy Down (MCD) [7, 8], Chunk caching Location and Searching (CLS) [9], ProbCache [15], and the weight based algorithm proposed by Ming *et al.* [11] fall into this category.

In global coordination, caching decisions consider all nodes; local coordination uses similar techniques, but considers only a subset of nodes. A hash routing technique proposed by [18] hashes content names to determine designated caches for each content; as a result, each router is responsible for a disjoint subset of content objects. Saha *et al.* exploited a similar concept in a preliminary proposal where they hashed objects at the autonomous system (AS) level [17]. A collaborative benefit-based eviction scheme to minimize traffic cost was proposed in [20], wherein routers iterative substitute a cached content with uncached content in such a way as to maximize the expected benefit. Wang *et al.* proposed the degree centrality heuristic for collaborative content eviction in [22], aiming to reduce redundancy, hence increasing cache diversity but also latency.

In addition to the aforementioned strategies, several other aspects of ICN caching have received attention. A unified approach for cache performance analysis was proposed in [10], utilizing an extension to Che's approximation to evaluate random and LRU-based replacement policies. In [13], the authors show that content-peering between ASes leads to a stable cache configuration, with or without coordination. Wang *et al.* investigated the impact of network topology structure, size, content popularity, and cache replacement strategy on optimal cache placement in [24].

We identified that most current ICN caching schemes target either the minimization of latency, or the maximization of cache utilization; conversely, coordination overhead is often neglected. Hence, we focus on designing a caching framework which balances latency and cache utilization, without inducing much overhead. We employ cache summaries in the form of Bloom filters, a technique previously employed in HTTP caching [4] and wireless ad-hoc networks [14]. Probabilistic cache summaries is not yet popular in ICN, but preliminary designs have been proposed by Tortelli *et al.* [21] and Wang *et al.* [23]. While interesting, these designs have not been implemented nor simulated, thus their practical effectiveness is unknown; thus, we believe that our work is the most thorough investigation into this type of strategy.

3. SYSTEM MODEL AND ASSUMPTIONS

We assume that the network is composed of a set of consumers, routers, and content providers. A consumer intermittently requests content items, while each provider serves a set of content objects, and each router has capacity to cache some of these contents. The h-hop neighborhood of a router includes that router and all other routers reachable in h hops or less. We assume that each router has a rank attribute, either assigned arbitrarily or based on some metric such as betweenness centrality, closeness centrality, or node degree [25].

We note that various models of content popularity exist, but no model is complete. In the stationary, or Independent Reference Model (IRM), the popularity distribution of content items does not change over time; the popularity is defined by the Zipf distribution [2]. However, it is more realistic to allow the popularity of contents to vary over time (temporal model). We will generate temporal workloads using GlobeTraff [6], which models a mix of content objects from various categories, which each have their own popularity models and temporal behaviors. We will account for both the IRM and temporal models in our discussions.

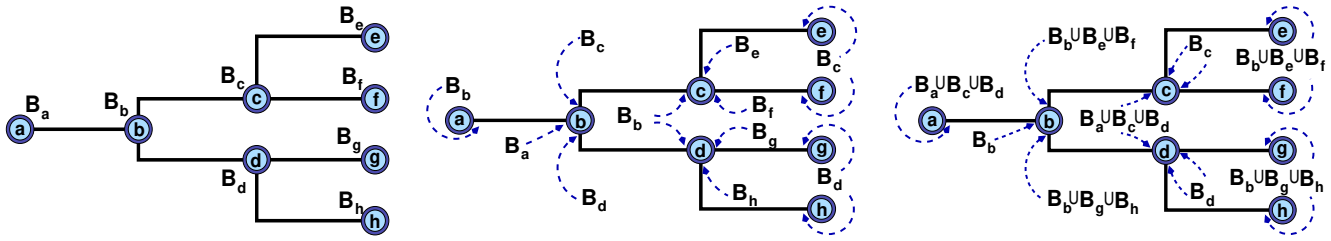
Although the fundamental design of MuNCC is applicable to all ICN architectures, we will focus on the popular NDN [5] design and nomenclature. In this paper, we augment NDN by partitioning the Content Store (CS), storing additional forwarding state in the Pending Interest Table (PIT), and altering the forwarding algorithm in order to collaboratively cache contents and satisfy interests.

4. BUILDING BLOCKS OF MUNCC

In our framework, a router b forwarding a request leverages information exchanged with its direct neighbors to assess whether another router in its h-hop neighborhood, $\mathcal{N}(b, h)$, can satisfy the request from its cache. This exchanged information provides aggregated cache state information in b's h-hop neighborhood and is stored in a set of attenuated Bloom Filters (BFs). Router b also collaborates with its direct neighbors to reduce content redundancy in its $\mathcal{N}(b, h)$ through cache eviction in its one-hop neighborhood. If a request cannot be satisfied in $\mathcal{N}(b, h)$, then b uses its FIB to forward the request towards the producer in the normal manner. In this section, we present the two building blocks of MuNCC: Attenuated Bloom Filter Construction and Exchange, and Coordinated Cache Eviction.

4.1 Attenuated Bloom Filter Construction and Exchange

First proposed by Rhea and Kubiatowicz in 2002 [16], a k-level attenuated BF is composed of k regular BFs, wherein the i^{th}-level BF summarizes objects reachable within i hops from a node. We propose an extension to this design and discuss its construction.



(a) Cache summary of each node (b) Level 0 of Attenuated BF (c) Level 1 of Attenuated BF

Figure 1: Two-level attenuated Bloom filter: (a) The cache summaries from which aggregate BFs are computed; (b) Level-0 BFs, which provide the cache summary of the direct neighbor; (c) Level-1 BFs, which are the union of the level-0 summaries held by the direct neighbor.

We will use the notation B_b to represent the BF summarizing a node b's own cache, and $\mathcal{B}^b_{(c,i)}$ to represent the i^{th} level of the attenuated BF stored by b for interface (b, c). The Level i BF corresponding to interface c summarizes the set of contents that c can reach in i hops, and thus b can reach in $(i + 1)$ hops. One h-level attenuated BF is stored for each interface of b; for the interface (b, c), b stores the set of BFs $\mathcal{B}^b_{(c,0)}, \ldots, \mathcal{B}^b_{(c,\,h-1)}$; thus, b obtains a cache summary for its entire h-hop neighborhood. The BF $\mathcal{B}^b_{(c,0)} = B_c$ summarizes content stored in c's own cache, while each BF $\mathcal{B}^b_{(c,i)}$ for $i > 0$ is an aggregated BF. If c has direct neighbors $d_1, \ldots, d_n$ (including b), then $\mathcal{B}^b_{(c,i)} = \mathcal{B}^c_{(d_1,i-1)} \cup \cdots \cup \mathcal{B}^c_{(d_n,i-1)}$ for each $0 < i < h$.

The construction of an attenuated BF occurs in h iterations. First, each node b transmits its cache summary B_b to its neighbors. This outgoing BF is tagged with a b's rank, $\boldsymbol{Ra(B_b)}$, assigned according to Section 3. A neighbor c stores the incoming BF as Level 0 of the attenuated BF for interface (c, b). Each node aggregates the Level 0 BFs it received in the first step by taking their union. The aggregate BF then takes the maximum rank among the rank values of all received Level 0 BFs. This aggregate BF is transmitted to the neighbors, which store it as Level 1 of the attenuated BF for their incoming interface. Each set of received BFs is aggregated and transmitted in the same fashion, until all h levels of the attenuated BFs have been populated.

Fig. 1 illustrates the generation of the attenuated BF with $h = 2$. We will focus on the state of the network as perceived by node b. The cache summary (BF) of each node is shown in Fig. 1(a) (e.g. B_b for b). In Fig. 1(b), we show the Level-0 BFs for each interface of b; these are the cache summaries of b's direct neighbors: B_a, B_d, and B_c respectively. Finally, Fig. 1(c) shows the Level-1 BFs. Here, b stores for each interface the aggregate summary of the caches of the corresponding neighbor's neighbors. For interface (b, c), b stores $\mathcal{B}^b_{(c,1)} = B_b \cup B_e \cup B_f$; for (b, a) it stores $\mathcal{B}^b_{(a,1)} = B_b$; and for (b, d) it stores $\mathcal{B}^b_{(d,1)} = B_b \cup B_g \cup B_h$.

False Positive Analysis: A BF is a probabilistic membership verification data structure which has no false negatives; that is, a negative reply to a membership query guarantees that the item is not indexed. Conversely, a positive reply to a membership query has a small false-positive probability (i.e., an item is not really in the set represented by the BF, but the query on the BF returns true). Lemma 4.1 formalizes the false positive probability of a conventional BF.

Lemma 4.1. *Assuming that the probability of each bit in the BF being set is independent, a BF of m bits, k hash functions, and n indexed content objects has false positive probability given by:*

$$P_{fp}(BF) = \left(1 - \left[1 - (1/m)\right]^{kn}\right)^k.$$

Proof. The proof is available in [12]. □

In MuNCC, each router creates an h-level attenuated BF, per interface. Upon receiving a content request, a router first checks for the content in its own cache. If the content is not in the CS, the router checks the Level-0 BFs it has collected, followed by Level-1 BFs, and so on. This continues until either a match is found and the content is retrieved, or all h levels of the attenuated BF are exhausted. Because the BF can return a false positive, match in the BF does not guarantee the successful retrieval of the content. A false positive increases request satisfaction latency and induces extra traffic in the neighborhood, and is therefore undesirable. Theorem 4.1 formalizes the probability of at least one false positive occurring during a particular node's BF lookups.

Theorem 4.1. *Given a node b with a set of interfaces L, the probability of at least one false positive while searching for a content in its BFs is given by,*

$$P_{fp}(b) = 1 - \left(1 - \left\{1 - \left[1 - (1/m)\right]^{kn}\right\}^k\right)^{|L|\cdot h}.$$

Proof. The probability of a BF not having a false positive is given by $1 - P_{fp}$. The probability of none

of the $|L| \cdot h$ BFs of b BFs having a false positive is $(1 - P_{fp})^{|L|\cdot h}$. Hence the probability of one or more false positives is $1 - (1 - P_{fp})^{|L|\cdot h}$, equivalent to the above result. □

Example: Assume node b has 6 interfaces and $h = 4$. Considering the Bloom filter size $m = 15$KB, and using $k = 5$ hash functions to index $n = 5000$ contents, the probability of at least one false positive at node b is given by:

$$\begin{aligned} P_{fp}(b) &= 1 - \left(1 - \left\{1 - \left[1 - \frac{1}{m}\right]^{kn}\right\}^{k}\right)^{|L|\cdot h} \\ &= 1 - \left(1 - \left\{1 - \left[1 - \frac{1}{12 \times 10^4}\right]^{5000\times 5}\right\}^{5}\right)^{6\times 4} \\ &= 0.0056 \end{aligned}$$

Size of Cache Summaries: We chose the size of the BFs in MuNCC to be proportional to the number of contents that can be cached in the network as a whole. Specifically, we designed the BF to index a number of contents $M = CfN$ at the false positive rate $F = 0.05$, where N is the size of the global content catalog, f is the maximum proportion of these content objects that can be cached, and C is an arbitrary scalar, which may be tuned to reduce false positives. We chose the number of hashes $k = \left\lceil \log_2 \frac{1}{F} \right\rceil$ and number of bits $m = \left\lceil M \cdot \ln \frac{1}{F} / \ln^2 2 \right\rceil$ for each BF, in accordance with the error-bounding method given by Almeida *et al.* [1].

Cache Turnover Management: Given the inevitability of cache turnover, a mechanism is required to remove stale items from cache summaries. Deletion operations are typically omitted from traditional BFs, as they introduce false negatives. Counting BFs provide a lossless delete operation, but are much larger and would thus increase overhead. In order to achieve a delete operation with minimal overhead, each level of each interface in MuNCC has both a "positive" BF and a "negative" BF as opposed to one standard BF. Thus, each node b's cache summary $B_b = (B_b^+, B_b^-)$, and each Level-i aggregated cache summary of interface (b, c), $\mathcal{B}^b_{(c,i)} = (\mathcal{B}^{b+}_{(c,i)}, \mathcal{B}^{b-}_{(c,i)})$. Node b will add cached items to B_b^+ and add evicted items to B_b^-. This pair of BFs will then be used by the neighbors of b to compute aggregate cache summaries.

The aggregation of the two-part BFs consists of the union of positive filters and the intersection of negative filters; e.g., the aggregation $\mathcal{B}^b_{(c,1)}$ of $\mathcal{B}^c_{(b,0)}, \mathcal{B}^c_{(e,0)}, \mathcal{B}^c_{(f,0)}$ is defined as $\mathcal{B}^{b+}_{(c,1)} = \mathcal{B}^{c+}_{(b,0)} \cup \mathcal{B}^{c+}_{(e,0)} \cup \mathcal{B}^{c+}_{(f,0)}$ and $\mathcal{B}^{b-}_{(c,1)} = \mathcal{B}^{c-}_{(b,0)} \cap \mathcal{B}^{c-}_{(e,0)} \cap \mathcal{B}^{c-}_{(f,0)}$. To check for the membership of a content o in the BF $\mathcal{B}^b_{(c,1)}$, b would check that $(o \in \mathcal{B}^{b+}_{(c,1)}) \wedge (o \notin \mathcal{B}^{b-}_{(c,1)})$. This implementation of deletion still induces false negatives, though much fewer than when simply unsetting bits in a flat BF.

Because we transmit cache summaries intermittently, there remains a possibility that b's neighbor c has evicted a content i from its cache, but the corresponding negative BF at b, $\mathcal{B}^{b-}_{(c,0)}$, is not updated yet. Thus b may still send c a request for i. Since c does not have the content anymore, c notifies b with a NACK. This is considered a false positive at b, which has the same effect as the intrinsic false positive of the BF. To prevent additional false positives, b will update $\mathcal{B}^{b-}_{(c,0)}$ upon receipt of the NACK.

The overall false positive probability of b requesting a content from c is presented in Corollary 4.1.

Corollary 4.1. *The probability of a false positive when b requests a content from c after indexing $\mathcal{B}^b_{(c,0)}$ is:*

$$P_{fp}(\mathcal{B}^b_{(c,0)}) = 1 - \left((1 - P_{fp}(\mathcal{B}^b_{(c,0)}))(1 - P_{evic}(B_c))\right)$$

where $P_{evic}(B_c)$ is the probability of the desired content being evicted from node c's cache.

This probability accounts for both the intrinsic BF false positive as well as premature eviction of the content from c's cache. Either event would result in an insertion to $\mathcal{B}^{b-}_{(c,0)}$.

4.2 Coordinated Cache Eviction

Limited cache size implies that eviction is inevitable. Coordinated eviction increases caching diversity and thus the overall cache-hit ratio, though at the cost of increasing the average retrieval distance for popular contents. *We have designed a coordinated eviction scheme which increases diversity while ensuring that evicted contents remain available nearby, thereby minimizing this cost.*

In MuNCC, redundancy elimination can be initiated when a node b receives a Level-0 (non-aggregate) cache summary from its neighbor c. Node b decides whether to begin evicting by comparing its own rank with the rank tag of the BF received from c. The strategy may be configured to evict at b either when b's rank is higher or lower than rank of c; if eviction is to proceed, then b iterates through its cache to find items also present in the received BF, $\mathcal{B}^b_{(c,0)}$ and evict any that match. Generally, eviction at the node of lower rank gradually moves content towards the core of the network, while eviction at higher rank node pushes the content towards the edge.

Eviction without advertising an update may cause neighbors to continue requesting the item, resulting in false positives as explained in Section 4.1. To reduce the number of eviction-related false positives, we employ a two-layer content store which allows each node to provide some guarantee of availability for content items it has advertised but wishes to evict. We partition the CS into a primary cache, which uses the majority of the capacity ($\sim 90\%$), and a smaller secondary cache ($\sim 10\%$). The items in the primary cache are advertised in cache summaries, while the secondary cache is used to ensure the availability of the contents advertised

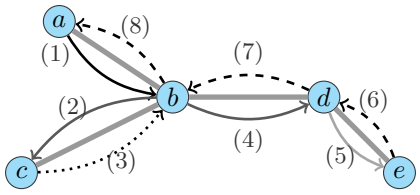


Figure 2: Request forwarding procedure: a's query for a content results in success in its $D = 2$ level BF: (1) a sends request to b with $D = 2$. (2) b sends request to c with $D = 1$. (3) c does not have content hence sends NACK to b. (4) b then sends request to d with $D = 1$. (5) d sends request to e with $D = 0$. (6, 7, 8) e serves the request from its cache, and the content propagates back to a.

in the past. If a content needs to be evicted from the primary cache, it will be transferred to the secondary cache instead of being deleted immediately.

Both the primary and secondary caches utilize the least frequently used (LFU replacement policy; thus, if the secondary cache becomes full, the least popular content will be evicted permanently. Although some advertised contents may still be evicted, this mechanism prevents the majority of eviction-related false positives. In the event that a node b queries its neighbor c for a content which has been evicted despite this, b can update its copy of the attenuated BF on (b, c), as explained in Section 4.1; thus, future requests for that object will not be forwarded to c.

5. MUNCC: THE COORDINATED CACHING STRATEGY

In this section, we finalize the design details of MuNCC and explain its forwarding procedure.

5.1 Routing Decisions

Algorithms 1 and 2 abstractly describe the forwarding decisions at a node b; Algorithm 1 (Request Handler) receives incoming requests and calls Algorithm 2 (Request Helper) as needed to route a request within a particular level of the neighborhood. In MuNCC, we tag interest packets with a *maximum distance* field, D, which specifies the number of hops across which the interest may propagate. The value of D can be either ∞ or an integer in the interval $[0, h]$. When a requester generates a new interest, it sets $D = \infty$, indicating that the request may be forwarded as far as needed in order to be served. Though similar to a time-to-live (TTL), D is different and does not replace the TTL used at the network layer.

When $D = \infty$, each router will attempt to serve an interest first from its own cache, and then from its neighborhood, before forwarding the request towards the content provider (Algorithm 1, Lines 2–12). If a node b has a cache miss, it checks the BFs $\mathcal{B}^b_{(n_1,0)}, \ldots, \mathcal{B}^b_{(n_k,0)}$, summarizing the caches of its direct neighbors $n_1, \ldots, n_k$ (Algorithm 1, Lines 4–7). If the content is not found within these Level-0 cache summaries, then the next level of BFs $\mathcal{B}^b_{(n_1,1)}, \ldots, \mathcal{B}^b_{(n_k,1)}$ is checked. Each level $0, \ldots, (h-1)$ of the attenuated BF is checked iteratively until the content is found. At each level, the BFs are checked in order of decreasing rank (Algorithm 2, Line 7). When a match occurs, the interest is re-tagged with a value of D corresponding to the level of the attenuated BF where the item was found and forwarded to the appropriate peer (Algorithm 2, Line 9). If all filters return no match, the request is routed toward the content source (D's value remains ∞) and the next router performs the same procedure (Algorithm 1, Lines 8–12).

If a request arrives at a router with $D = 0$, the router only attempts to serve the request from its own cache (Algorithm 1, Line 13; Algorithm 2, Lines 2–5). When $0 < D \leq h$, the router should only check the $(D-1)^{\text{th}}$ level of its attenuated Bloom filters (Alg. 2, Lines 7–15). If $D \neq \infty$ and all valid options to serve the request have been exhausted, a NACK is generated to indicate a failed probe (Algorithm 1, Line 14). The NACK is sent to the previous hop to inform it that the request could not be satisfied.

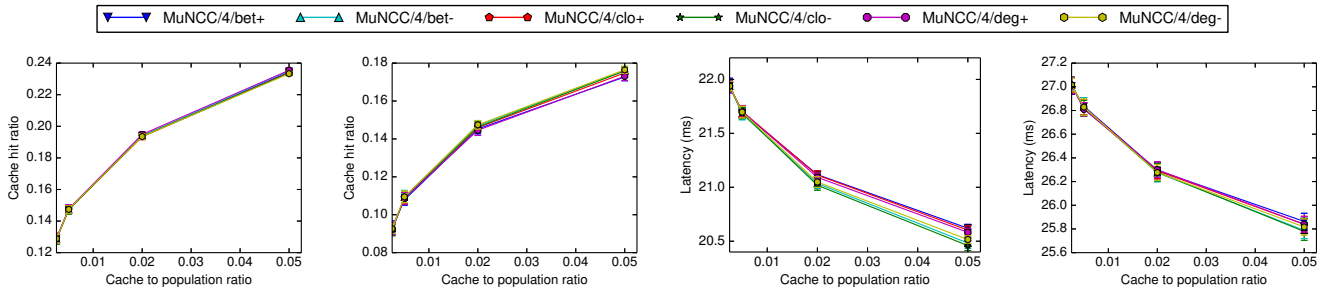
A node receiving a NACK will update the appropriate BF (corresponding to the interface on which the interest was forwarded, and its D field) in order to reflect that the content is no longer available through that interface. Depending on whether the node has exhausted

Algorithm 1 Request Handler at b

```
1:  function requestHandler(i, D)
2:      if D = ∞ then
3:          requestServed = false
4:          for D' = 0 to (h − 1) do
5:              if requestHelper(i, D') = true then
6:                  requestServed = true
7:                  break
8:          if not requestServed then
9:              forward request toward content provider
10:             wait for response
11:             cache the response if appropriate
12:             send response to previous hop
13:     else if requestHelper(i, D) = false then
14:         send NACK to previous hop
```

Algorithm 2 Request Helper at b

```
1:  function requestHelper(i, D')
2:      if D' = 0 then
3:          if c in local cache then
4:              send response to previous hop
5:              return true
6:      else
7:          for each neighbor p, in decreasing Ra(B^b_(p,D'−1)) do
8:              if i in B^u_(p,D'−1) then
9:                  forward request to p with D = D' − 1
10:                 wait for response
11:                 if response is a NACK then
12:                     remove i from B^b_(p,D'−1)
13:                 else
14:                     send response to previous hop
15:                     return true
16:     return false
```



(a) Cache-hit ratio: GEANT (b) Cache-hit ratio: TISCALI (c) Latency: GEANT (d) Latency: TISCALI

Figure 3: Effect of rank metric on cache-hit ratios and content retrieval latencies with cache-to-population ratios $\{0.25\%, 0.5\%, 2\%, 5\%\}$ under the temporal popularity workload on real-world graphs. Error bars indicating 95% confidence intervals are illustrated.

its forwarding options, it may then attempt to route the request elsewhere in its neighborhood, route toward the content provider, or send a NACK back to the previous hop. An example of the forwarding behavior is given in Fig. 2.

In MuNCC, a node will never cache a content if it was found in a neighborhood cache. However, the node may choose to cache a content if the corresponding request was forwarded toward the producer. In this case, the augmented LFU policy described in Section 4.2 will make the ultimate caching decision. It is possible that several nodes on the path will cache the same content as it is forwarded back to the requester; in this scenario, the collaborative eviction mechanism from Section 4.2 will help eliminate the redundancies.

5.2 Refreshing Cache Summaries

Due to cache turnover, BFs must be refreshed intermittently to prevent false positives. In the case of a false positive in a positive BF, a node may route a request toward a neighbor which cannot serve it, or in the case of a false positive in a negative BF it will route toward the content source when the request could actually be served by a neighbor. To prevent this, we define regularly occurring epochs; at a new epoch, every node b recomputes its cache summary B_b from the content cached at that time. Thus, at the beginning of the epoch, each positive filter will only consist of items which are currently cached, while each negative filter will be empty. After reconstructing the cache summary, h rounds of transmission and aggregation are performed as described in Section 4.1. At the end of this procedure, all nodes in the neighborhood will have up-to-date aggregate summaries.

In order to keep caches fresh between epochs, we also schedule several sub-epochs. At the beginning of a sub-epoch, a node does not compute its cache summary from scratch. Rather, the node continually updates its cache summary as it caches and evicts contents, then at the sub-epoch transmits the updated BF to its peers. Again, there are h iterations in order to update the entire neighborhood. When the sub-epoch occurs, each node may also clear its secondary cache, as it has now informed its neighbors about the eviction decisions it intended to make.

In our design, we do not immediately delete the content, but rather mark it as "stale." This allows new contents to be moved to the secondary cache if necessary, but prevents the secondary cache space from being underutilized.

We have also optimized the BF transmission procedure such that a node need not transmit its cache state if no changes have occurred. If this is the case, a neighbor can carry the previous cache state into the new epoch or sub-epoch. Similarly, aggregate cache state summaries do not need to be recomputed or transmitted if no updates have been received from neighbors.

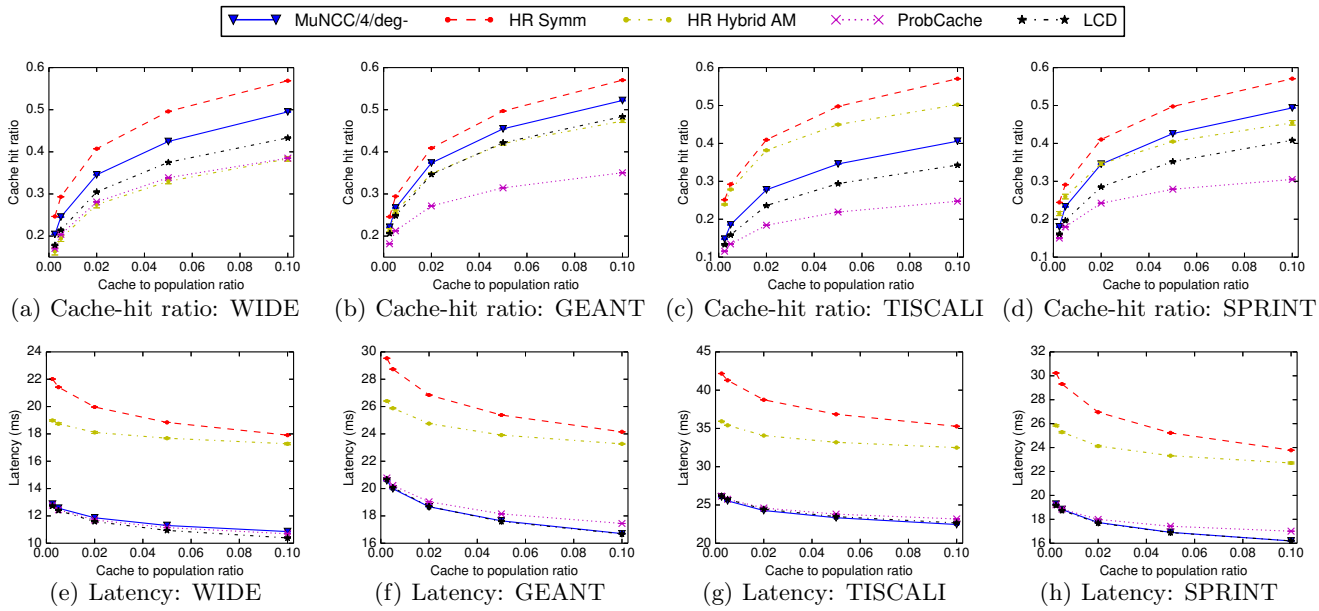
6. SIMULATION AND ANALYSES

6.1 Simulation Setup

We implemented MuNCC in the Icarus simulator [19] and compared its effectiveness against Leave Copy Down (LCD) [8], ProbCache [15], and two of the best hash-routing schemes proposed by Saino *et al.* [18]: Symmetric Hash Routing (HR Symm) and Hybrid Asymmetric-Multicast Hash Routing (HR Hybrid-AM). Each strategy was backed by an LFU-replacement content store; for MuNCC, we used the two-level LFU cache described in Section 4.2. Caches were installed on all routers in the network and cache indexing was performed at the content level.

A total of eight topologies were used in our evaluations. Four were real-world ISP-like topologies (nodes, edges): WIDE (30, 33), GEANT (53, 61), TISCALI (240, 810), and SPRINT (604, 2268). Four were scale-free topologies generated with BRITE [3] following a 2-level hierarchical Barabasi-Albert model ($m = 2$) (nodes, edges): Topo1 (250, 492), Topo2 (500, 972), Topo3 (750, 1447), Topo4 (1000, 1897). Each result was averaged over eight distinct runs.

Both stationary (IRM) and temporal popularity work-



(a) Cache-hit ratio: WIDE (b) Cache-hit ratio: GEANT (c) Cache-hit ratio: TISCALI (d) Cache-hit ratio: SPRINT

(e) Latency: WIDE (f) Latency: GEANT (g) Latency: TISCALI (h) Latency: SPRINT

Figure 4: Comparison of cache-hit ratios and content retrieval latencies with cache-to-population ratios $\{0.25\%, 0.5\%, 2\%, 5\%, 10\%\}$ under the stationary popularity workload on real-world graphs. Error bars indicating 95% confidence intervals are illustrated.

loads were tested. The stationary scenarios used a catalog of $N = 3{\cdot}10^5$ content objects with popularity defined by the Zipf distribution ($\alpha = 0.8$). A total of $1.2{\cdot}10^6$ requests were generated in these scenarios. The first $6{\cdot}10^5$ requests were used to allow caches to converge and were not used for gathering statistics; the remaining $6 \cdot 10^5$ requests were logged and used to gather statistics. In the temporal scenarios, we used workloads generated by GlobeTraff [6] with the default parameters. Each workload consisted of about $2.1 \cdot 10^5$ content objects. All requests were used for statistics in the temporal workloads, as content popularity does not converge.

In the stationary scenarios, we set the parameter $C = 1.25$; in the temporal scenarios, we set $C = 1.79$ to compensate for the higher rate of cache turnover. These values of C also imply that the size of the BF is the same for a given network-wide cache size, under both the temporal and stationary workloads. With a network-wide cache-to-population ratio of 10% (the largest used in our simulations), the number of hashes $k = 5$ and each BF was less than 30 kilobytes in length, as given by the formula in Section 4.1.

For each run, we configured our strategy with $h = 4$, epoch intervals of 1800 seconds, and 10 sub-epochs per epoch. In all cases, cache capacity was distributed uniformly amongst all routers and the caches were initially empty.

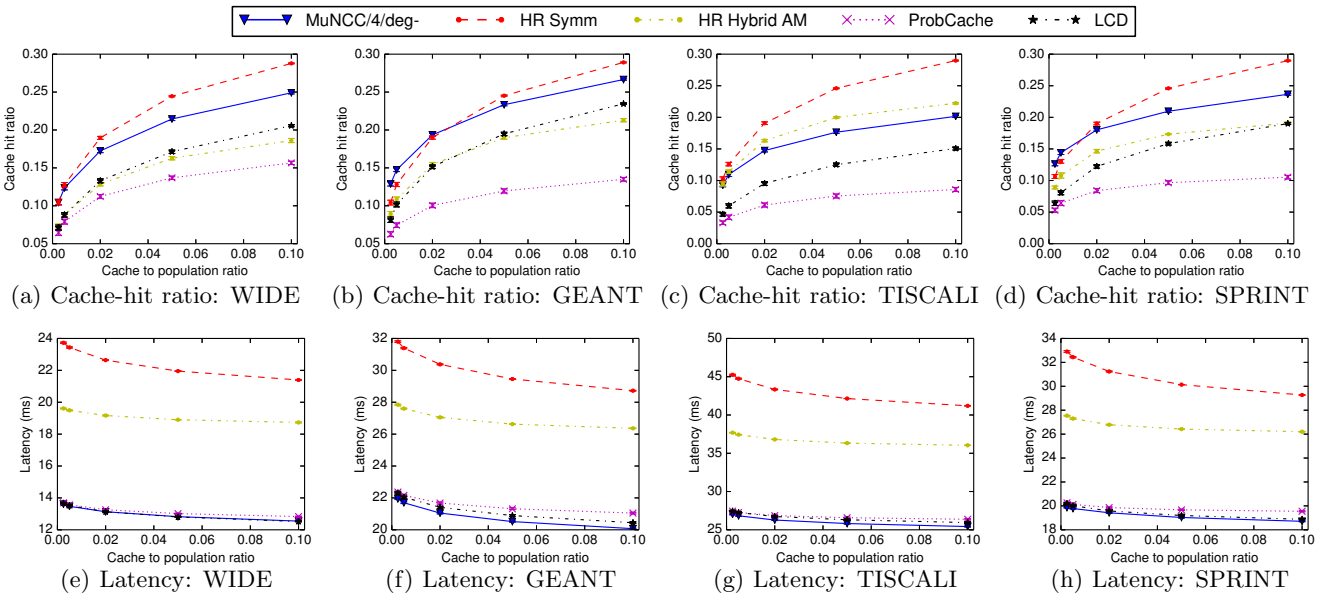
6.2 Effect of Various Rank Metrics

MuNCC uses the ranks of nodes to make eviction decisions. As stated in Section 3, these ranks can be assigned arbitrarily. In our simulations, we explored the effectiveness of three different rank attributes: betweenness centrality (`bet`), closeness centrality (`clo`), and degree (`deg`). For each choice of metric, we tested two variations: positive (`+`) and negative (`-`). Under the positive variation, redundancy elimination is performed at the lower-ranked node; thus, high-rank nodes (typically core routers) are treated preferentially. For the negative variation, redundancy elimination happens at higher-rank nodes, thus the lower-rank nodes (typically edge routers) are preferred.

For this evaluation, we used the temporal popularity workload with {GEANT, TISCALI} graphs and network-wide cache fractions equivalent to $\{0.25\%, 0.5\%, 2\%, 5\%\}$ of the global content catalog. Fig. 3 shows the effect of metric choice on observed cache-hit ratios and request fulfillment latency. `MuNCC/4/bet+`(`-`) refers to the configuration with $h = 4$ and betweenness as the rank metric, preferring core nodes (edge nodes). Same naming convention is used for closeness centrality (`clo`) and degree centrality (`deg`). In the majority of scenarios, there is no major difference between the tested metrics. However, positive (`+`) metrics sometimes marginally increase the cache-hit ratio, while negative (`-`) metrics sometimes reduce latency slightly. We conclude that while a positive or negative metric may be preferred based on whether cache-hit ratio or latency is the priority, choosing a centrality metric (which is often expensive to compute) over a degree metric (which is inexpensive) provides no significant benefit.

6.3 Comparison with State of Art

Now we will compare the performance of MuNCC



(a) Cache-hit ratio: WIDE (b) Cache-hit ratio: GEANT (c) Cache-hit ratio: TISCALI (d) Cache-hit ratio: SPRINT

(e) Latency: WIDE (f) Latency: GEANT (g) Latency: TISCALI (h) Latency: SPRINT

Figure 5: Comparison of cache-hit ratios and content retrieval latencies with cache-to-population ratios $\{0.25\%, 0.5\%, 2\%, 5\%, 10\%\}$ under the temporal popularity workload on real-world graphs. Error bars indicating 95% confidence intervals are illustrated.

with LCD, ProbCache, HR Symm, and HR Hybrid AM. For this comparison, we used the cache fractions $\{10\%, 5\%, 2\%, 0.5\%, 0.25\%\}$ and selected degree as our rank metric with eviction at the higher-ranked nodes (`deg-`).

Fig. 4 presents the comparison for the real-world graphs under the stationary popularity workload. In terms of cache-hit ratio we did not perform as well as HR Symm, which has the best cache utilization in the literature. In three out of four topologies, we outperformed HR Hybrid AM, wherein a request may not be relayed to its designated cache if the resultant path stretch is greater than a certain threshold. In every scenario we outperformed ProbCache and LCD, which do not utilize neighborhood caches.

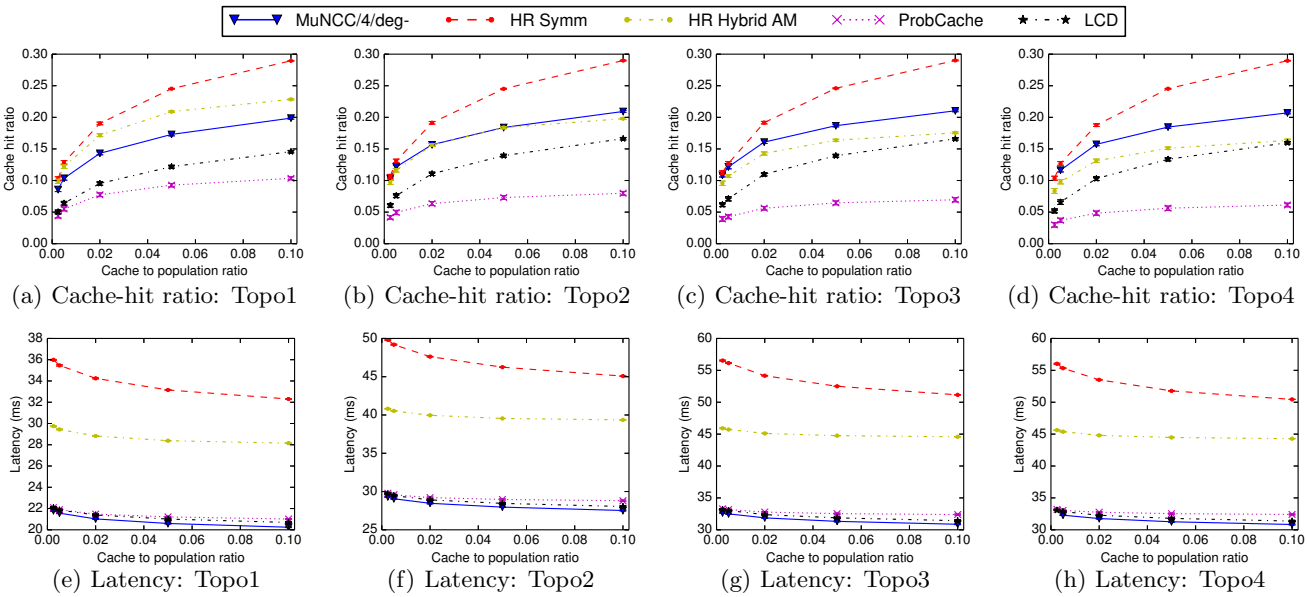
Though HR Symm obtained a higher cache-hit ratio, MuNCC was superior in latency reduction, as it leveraged neighborhood caches instead of preferring a potentially-distant authoritative cache. The average latency under our strategy was consistently within one millisecond of LCD (which has the lowest latency in the majority of cases), as both LCD and MuNCC place popular content in caches near consumers. In general, we achieve between 30% and 40% reduction in latency compared to HR Symm by avoiding path stretch.

Fig. 5 presents the comparison for the real-world graphs under the temporal popularity workload. Here, we outperformed HR Hybrid AM on most topologies. Interestingly, we also outperformed HR Symm on GEANT with cache-to-population ratios less than 5%, as well as on SPRINT with cache-to-population ratios less than 2%. Here, the HR schemes suffer from slow convergence due to popularity dynamicity and the use of the LFU replacement policy; LFU's effects are more visible on HR due to the compounding effect of redirection toward authoritative caches. Again, we outperformed ProbCache and LCD in all cases. In terms of latency, we outperformed all strategies in almost all scenarios.

In Fig. 6, we show the cache-hit ratio and latency statistics collected on the scale-free graphs under the temporal popularity workload. Qualitatively, these results are very similar to the corresponding results on real-world graphs. In all scale-free scenarios, our strategy provided the lowest latency.

6.4 False Positives and Cache Exchange Overhead

In this subsection, we demonstrate the low impact of false positive events and cache exchange overhead in MuNCC. We will use the `deg-` rank metric and the temporal popularity workload with the set of cache fractions $\{0.5\%, 2\%, 5\%\}$ on the real-world graphs. The fractions of requests affected by false positives and the overhead transmission rates for each scenario are given in Table 1. The false positive incidence shown in this table includes both false positives caused by the probabilistic nature of the BF and false matches due to outdated cache summaries (e.g., when an eviction occurred after the latest BF exchange). Note that in all cases less than 1% of requests experience a false positive. Here, we have presented overhead in terms of BF transmissions per-second per-node. Actual overhead bandwidth scales with the chosen BF size; however, note that even in the worst case (SPRINT, 5%), the effective overhead bandwidth induced by each node will be less than 100



(a) Cache-hit ratio: Topo1 (b) Cache-hit ratio: Topo2 (c) Cache-hit ratio: Topo3 (d) Cache-hit ratio: Topo4

(e) Latency: Topo1 (f) Latency: Topo2 (g) Latency: Topo3 (h) Latency: Topo4

Figure 6: Comparison of cache-hit ratios and content retrieval latencies with cache-to-population ratios $\{0.25\%, 0.5\%, 2\%, 5\%, 10\%\}$ under the temporal popularity workload on scale-free graphs. Error bars indicating 95% confidence intervals are illustrated.

bytes per second.

We observe that on WIDE and GEANT, increasing cache fractions decreases the incidence of false positives. However, the behavior is different on the larger graphs TISCALI and SPRINT, where the incidence peaks at 2% cache size. In every case, increased cache size leads to an increase in overhead. This may be attributed to lower-popularity content being more turbulent, and thus more likely to be evicted; then, BFs which would otherwise have been unchanged must be refreshed every sub-epoch due to the increased number of evictions. The SPRINT topology, which has a greater average node degree than the other topologies, has the greatest overhead. In a real deployment, overhead may be reduced by tuning the parameters h and C, as well as the epoch and sub-epoch lengths. We have not tuned these parameters to each graph in our simulations; the same parameters are used in every case.

Table 1: Request false positive incidence and overhead transmissions per second per node under the temporal popularity workload.

		False Positive Rate	Overhead Rate
WIDE	0.5%	0.001372	0.000497
	2%	0.000819	0.000949
	5%	0.000668	0.001170
GEANT	0.5%	0.003139	0.000725
	2%	0.002315	0.001250
	5%	0.002120	0.001527
TISCALI	0.5%	0.004133	0.000415
	2%	0.004948	0.000809
	5%	0.004093	0.001139
SPRINT	0.5%	0.006620	0.001422
	2%	0.006979	0.002333
	5%	0.004843	0.002962

7. CONCLUSION AND FUTURE WORK

In this paper we introduced MuNCC, a coordinated caching strategy which employs attenuated Bloom filters to aggregate cache states within the h-hop neighborhoods of each router. This allows neighbors' caches to be utilized to serve requests with low latency and overhead. We leverage redundancy elimination to improve cache diversity in a neighborhood and achieve competitive cache-hit ratios. Simulation results show that MuNCC achieves latency levels comparable to simple on-path caching schemes while increasing cache hit ratio to levels comparable to network-wide content hashing.

Where other schemes have achieved higher cache hit ratios, they require significant communication overhead or induce path stretch that makes them unsuitable for real-world deployment. As MuNCC achieves low latency while keeping overhead at a minimum, it would be more practical than such schemes. We envision that Internet Service Providers could eventually employ a mechanism like MuNCC in order to improve user quality of experience (QoE) as well as reduce costs by reducing the need to transit other networks.

MuNCC can be further improved to reduce its coordination overhead and reduce false positive rates, as well as potentially increase cache-hit ratios. Such avenues may be explored in future work.

8. REFERENCES

[1] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Information Proc. Letters*, 101(6):255–261, 2007.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.

[3] Brite: Boston university representative internet topology generator, 2014. http://www.cs.bu.edu/brite.

[4] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[5] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking named content. In *ACM CoNEXT*, pages 1–12, 2009.

[6] K. V. Katsaros, G. Xylomenos, and G. C. Polyzos. Globetraff: a traffic workload generator for the performance evaluation of future internet architectures. In *IFIP Intl. Conf. on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2012.

[7] N. Laoutaris, H. Che, and I. Stavrakakis. The LCD interconnection of LRU caches and its analysis. *Performance Evaluation*, 63(7):609–634, 2006.

[8] N. Laoutaris, S. Syntila, and I. Stavrakakis. Meta algorithms for hierarchical web caches. In *IEEE IPCCC*, pages 445–452, 2004.

[9] Y. Li, T. Lin, H. Tang, and P. Sun. A chunk caching location and searching scheme in content centric networking. In *IEEE Intl. Conf. on Communications (ICC)*, pages 2655–2659, 2012.

[10] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, pages 2040–2048. IEEE, 2014.

[11] Z. Ming, M. Xu, and D. Wang. Age-based cooperative caching in information-centric networks. In *IEEE INFOCOM WKSHPS*, pages 268–273, 2012.

[12] J. K. Mullin. A second look at bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.

[13] V. Pacifici and G. Dan. Content-peering dynamics of autonomous caches in a content-centric network. In *IEEE INFOCOM*, pages 1079–1087, 2013.

[14] E. Papapetrou, E. Pitoura, and K. Lillis. Speeding-up cache lookups in wireless ad-hoc routing using bloom filters. In *IEEE PIMRC*, pages 1419–1423, 2005.

[15] I. Psaras, W. Chai, and G. Pavlou. Probabilistic in-network caching for information-centric networks. In *ACM ICN*, pages 55–60, 2012.

[16] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *IEEE INFOCOM*, pages 1248–1257, 2002.

[17] S. Saha, A. Lukyanenko, and A. Yla-Jaaski. Cooperative caching through routing control in information-centric networks. In *IEEE INFOCOM*, pages 100–104, 2013.

[18] L. Saino, I. Psaras, and G. Pavlou. Hash-routing schemes for information centric networking. In *ACM Conference on Information-Centric Networking*, pages 27–32, 2013.

[19] L. Saino, I. Psaras, and G. Pavlou. Icarus: a caching simulator for information centric networking (ICN). In *ICST SIMUTOOLS*, pages 66–75, 2014.

[20] V. Sourlas, L. Gkatzikis, P. Flegkas, and L. Tassiulas. Distributed cache management in information-centric networks. *IEEE Trans. on Network and Service Management*, 10(3):286–299, 2013.

[21] M. Tortelli, L. Alfredo Grieco, and G. Boggia. CCN forwarding engine based on bloom filters. In *ACM Conference on Future Internet Technologies*, pages 13–14, 2012.

[22] J. Wang, J. Zhang, and B. Bensaou. Intra-AS cooperative caching for content-centric networks. In *ACM ICN*, pages 61–66, 2013.

[23] Y. Wang, K. Lee, B. Venkataraman, and *et al.* Advertising cached contents in the control plane: Necessity and feasibility. In *IEEE INFOCOM WKSHPS*, pages 286–291, 2012.

[24] Y. Wang, Z. Li, G. Tyson, S. Uhlig, and G. Xie. Optimal cache allocation for content-centric networking. In *21st IEEE Intl. Conf. on Network Protocols (ICNP)*, pages 1–10, 2013.

[25] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.