A Two-tiered Caching Scheme for Information-Centric Networks

Kelvin H.T. Chiu[1], Jason Min Wang[1], Ahmed M. Abdelmoniem[2,3], and Brahim Bensaou[1]
[1]*CSE Department, HKUST, Hong Kong*
[2]*CS Department, FCI, Assiut University, Egypt* [3]*CEMSE Division, KAUST, Saudi Arabia*
{htchiuaa@connect.ust.hk, jasonwangm@cse.ust.hk, ahmed.sayed@kaust.edu.sa, brahim@cse.ust.hk}

***Abstract*—In information centric networking (ICN), by default, forwarder nodes along the paths from content producers to consumers, cache and reuse content chunks ubiquitously, invoking the Least Recently Used (LRU) replacement policy when needed. Due to the *cache filtering effect*, this ubiquitous-LRU strategy is inefficient: popular contents that are cached a few hops away from the edge are of little utility. Most alternative proposals adopt unified on-path schemes that rely on popularity statistics or other global state information to improve the performance. In the Internet, it is difficult to imagine different administrative-entities exposing such information to each other, and so we argue that such schemes are not realistic; and secondly, a unified caching scheme across the network ignores the administrative autonomy, which is one of the tenets of the Internet. In this paper we argue for a two-tiered caching scheme that maintains an on-path caching scheme (e.g., Ubiquitous-LRU), yet embraces AS-autonomy by adding within the AS an off-path cooperative-caching and redundancy elimination, to make better use of caches in the edge where they are most valuable. We describe the implementation of our scheme and highlight the design choices to make the system practical. Our evaluation results show that our scheme can reduce cache misses and upstream traffic by up to 15% compared to the state-of-the-art.**

***Index Terms*—Cooperative Caching, Redundancy Elimination, Information-Centric Networks (ICN)**

I. Introduction

Information-Centric Networking (e.g., Named-Data Networking, NDN and Content-Centric Networking, CCN) [1] proposes a clean slate *redesign* of the Internet *service model*, by promoting *named-data* as the first class citizen in the network. While the controversial ambition of ICN to ultimately replace the Internet *architectural model* is not being debated in this paper, we argue that the independence enjoyed by individual autonomous systems (AS), which is a cornerstone of the current Internet, would remain uncontested with or without ICN. As such we argue that ICN proposals should take this into account.

By default, CCN forwarders[1] cache data *ubiquitously*, and use the least-recently-used (LRU) caching as the cache replacement strategy when caches are full. This strategy, although simple to implement, leads to highly redundant caches

[1]In the sequel, we will mostly talk about CCN as we have implemented our code on the open source Community ICN (CICN) project, however, since CCN and NDN are very similar and are rooted in the same original design, our scheme can also be ported easily to NDN.

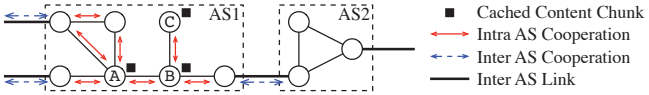

Fig. 1. intra AS vs. inter AS cooperation. Our proposed scheme only requires intra AS communication, thus fully embracing AS autonomy. **No** signalling information needs to pass from AS to AS. Our two-tiered scheme relies on Ubiquitous-LRU to populate the caches while it *reduces* redundancy, freeing up cache spaces that are not needed *within* the AS, by relying on intra-AS cooperation.

throughout the AS. In this paper, we design, implement, and evaluate a cooperative caching scheme on a production-ready code distribution of ICN, the so-called Community ICN (CICN) project. Our design embraces AS autonomy by enabling forwarders in an AS to *cooperate* on their caches. Our proposed approach has four notable strengths:

1. It fully embraces AS-autonomy. *i)* We only assume CCN to be widely deployed *within the AS of interest*, but not throughout the entire Internet. *ii)* We do *not* require inter-AS cooperation, by enforcing the scope of cooperation to *within* the AS only, as illustrated in Fig. 1); thus, no cache signalling information needs to cross AS boundaries.
2. It is *backward compatible* with basic CCN forwarders that do not support our scheme which makes it deployment-ready in real-world scenarios. This is essential because deployments are usually gradual and CCN forwarders not running our scheme may co-exist with ones running our scheme.
3. It strictly follows the CCN *design philosophy*, by making no assumption of a particular underlying network layer (i.e., it does not rely on TCP/IP). Our scheme is designed such that all signaling information exchanged among forwarders is carried in pure, native CCN packets only, which allows it to work seamlessly with future iterations of CCN and increase its deploy-ability. For example, one could bypass TCP/IP to run directly on top of Ethernet or adopt any new underlying network protocol without affecting the operations of our scheme.
4. As far as we know, there are no works in literature that discusses the actual *implementation* and design decision of a non-trivial cooperative caching scheme in a production-ready code base. We make several important design decisions that would otherwise make our scheme impractical to implement in a real system.

The remainder of this paper is organized as follows, in

Section II we describe our system, and the analytic model that it stems from, then argue in favor of some approximations that reduce the complexity of our scheme to finally propose the cooperation algorithm. In Section III, we describe the system architecture and discuss the details of the main components of the system. In Section VI we discuss some related works. Section 7 is dedicated to discussing some experimental results, then we conclude the paper in Section VII.

II. System Overview and Problem Formulation

A. Overview

In our system, the caches of the forwarders are populated with content chunks via the default on path caching and replacement scheme (e.g., Ubiquitous-LRU). Then, at the same time, forwarders inside each AS, *within one network hop* of each other, periodically exchange summary information of their cached content chunks, and node degrees (number of ingress edges to the node). Given these information, the following two steps occur: 1) redundant content chunks between two neighbor forwarders are identified; 2) the forwarder with the *higher* node degree (e.g., node A of degree 4 in Fig. 1) *holds* the redundant chunk, by *preventing cache replacement action of the caching policy*, while the lower degree node (e.g., node B of degree 3 in Fig. 1) *evicts* (Note: A content chunk that is being held is never evicted.) the chunk, and installs a forwarding rule to retrieve it from A. These two steps are repeated forever periodically, but a lifetime is associated with each held content chunk which expires when the content becomes less popular and so it becomes evict-able by the caching policy. When evicted, the *freed* cache slot can then be used to store new content chunks, which is the source of performance gain. Therefore, we should maximize the number of evictions *across an AS*.

B. Model

To expedite the modeling, we extend an existing model presented in our prior work [2]. Let the network topology of an AS be a *graph* $G = (V, E)$. V is the set of *forwarders* in the AS. $E \subseteq (V \times V)$ is the set of *links* between a pair of forwarders. $\forall v_1, v_2 \in V, (v_1, v_2) \in E$ if and only if v_1 and v_2 are 1-hop neighbors. Given a *content chunk* k, we construct a *content graph* $G_k = (V_k, E_k)$, illustrated in Fig. 2(b,c): $\forall v \in V_k \subseteq V$ if and only if k is cached in v, and $\forall (v_1, v_2) \in E_k \subseteq E$ if and only if $(v_1, v_2) \in E$ and $v_1, v_2 \in V_k$. The maximization problem is reduced to solving the well-known Minimum Dominating Set (MDS) problem for *each* of the content graphs G_k, that gives a dominator set D_k. Given a graph $G = (V, E)$, the solution of the MDS problem is to find a *dominator set* with *minimum size*, $D \subseteq V$, such that $\forall v \in V, v \in D \vee (\exists u \in V, (u, v) \in E \wedge u \in D)$ (All nodes in the graph are either in the dominator set or has a 1-hop neighbor that is in the dominator set, while the size of the dominator set is minimized.). If a forwarder is a member of D_k, then it should *hold* k. Otherwise, it should *evict* k, illustrated in Fig. 2(b,c). However, it is practically infeasible to construct the content graphs for *all* possible content chunks.

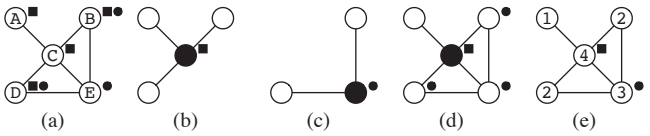

Fig. 2. **(a)** A network topology with 5 forwarders. Cached content chunks are shown as rectangles and circles. **(b,c)** Content graph of the square and circle chunk respectively, where the dominator sets are labeled in black. **(d)** The dominator set, labeled in black, for the graph in (a). Redundant content chunks in the dominator sets are held, and evicted at other forwarders forwarders. Thus, the circle chunks are not evicted. **(e)** Result of the adopted hybrid decision rule. Forwarders are labeled with the node degree. Forwarders C and E have highest degree among forwarders that cache the square and circle content chunk respectively.

The problem is simplified by solving only *one* MDS problem for the graph G, via the greedy approximation algorithm which costs $O(|V|)$ in time. Forwarders in the dominator set should always *hold* redundant content chunks, while others should always *evict*. However, if two forwarders within one network hop belong to the *evict* set, the redundant content chunks among them are not evicted which is illustrated by the circle chunks in Fig. 2(d).

A *hybrid* rule between running MDS problem for *content graphs* and *network graphs* is adopted, which is inspired by the *degree heuristic* from the greedy approximation algorithm. Specifically, nodes with *higher* degree are chosen to join the dominator set. Thus, as illustrated in Fig. 2(e), the decision rule should be:

- High-degree forwarders should *hold* redundant chunks.
- Low-degree forwarders should *evict* redundant chunks.

C. Distributed Algorithm

The decision rule only applies for *two* forwarders. To extend it to the entire AS, the hold and evict operations must be *coordinated*. In Fig. 1, forwarders A, B and C all cache the same content chunk. Degree of B is lower than A, but higher than C. Under the decision rule, B should evict the chunk for A while holding it for C. To resolve the conflict, forwarders with *lower* degree should *evict first*. Therefore, C should evict *first* and *notify* B to hold. When B takes its turn to evict, the content chunk is being held and cannot be evicted.

Forwarders take turns to periodically execute a procedure `evict()` to evict redundant content chunks while *notifying other forwarders* within one network hop with *higher* node degrees to hold the content chunks. A round should finish when all forwarders in the AS have finished executing `evict()` once. We design a deadlock-free distributed algorithm to coordinate the execution, such that in the same round, forwarders with lower degree should execute `evict()` *not after* forwarders with higher degree within one network hop.

The algorithm maintains for each forwarder a *sequence number* representing how many times `evict()` has been executed. `evict()` is executed for a forwarder if *both* of the following conditions holds for its sequence number:

1. is *strictly less than* all sequence numbers of other lower degree forwarders within one network hop away. (All low-degree forwarders within one hop have finished the *current* round of `evict()`)

Algorithm 1 Distributed Algorithm

Input: A Forwarder running this algorithm

procedure CHECKEVICT(A)
  $N \leftarrow$ 1-hop neighbor forwarders of A
  for $v \in N$ **do**
    if $\deg(v) < \deg(A) \wedge \text{seq}(v) \leq \text{seq}(A)$ **then**
      return ▷ Condition 1 is violated
    else if $\deg(v) > \deg(A) \wedge \text{seq}(v) < \text{seq}(A)$ **then**
      return ▷ Condition 2 is violated
    end if
  end for
  EVICT(A) ▷ Conditions 1 and 2 are satisfied
  $\text{seq}(A) \leftarrow \text{seq}(A) + 1$
end procedure
for every eviction interval **do**
  CHECKEVICT(A)
end for

2. is *less than or equal to* all sequence numbers of other higher degree forwarders within one network hop away. (All high-degree forwarders within one network hop have finished the *previous* round of `evict()`)

Algorithm 1 illustrates the proposed distributed algorithm in detail[2]. The *eviction interval* determines the frequency of evicting redundant content chunks in the AS.

III. System Architecture

The proposed system uses the following main components: 1) A control plane using native CCN packets is built to support signaling among forwarders which ensures backward compatibility with vanilla CCN forwarders; 2) Cache summaries which are encoded by Bloom filters, with a mechanism to minimize the effect of *time decay* of cache encodings. 3) Lifetime for held content chunks which is set dynamically to filter popular content chunks. 4) The forwarding logic of CCN which is modified to redirect requests after eviction.

A. Control Plane

The control plane is built on top of CCN using native CCN packets for *control messages*. The control follows the *Pull* paradigm of CCN [3] in which a request is sent to obtain a response with the content[3].

CCN interest and data packets contain a *payload field*, which is overloaded in control messages to carry the protocol data of the control plane, which has the following header: *i)* 4-bit *version* field. *ii)* 4-bit *type* field that multiplexes at most 16 types of control messages. There are three types of control messages. For example, The `DISC` (discover) type (`0x00`) performs discovery among forwarders and exchange of short data such as node degree and sequence number.

The node degree of CCN forwarders is the number of remote forwarders. But information producers and consumers are abstracted as *faces* in CCN, thus it is not possible to obtain the node degree by counting the number of faces. A `DISC` interest is sent to *all* faces periodically at every *discover interval*, to compute the node degree: In CCN, the ingress *face index* is associated with each incoming packet. Thus, we can maintain a *mapping* from face index to an entry storing *per-forwarder information*. A new entry is created if the face index does not exist. The node degree is simply the number of entries in the mapping. A `DISC` data packet is returned with the information, when received, the entry in the mapping is updated. If no `DISC` data packet is returned for the fifth time, the entry in the mapping is removed. The packet format of `DISC` control message is shown in Fig. 5(a).

In our experiments, the name `/ust.hk/co-op` is reserved for control messages and is not used by other CCN packets. Thus, the control plane processes only packets with this name on top of CCN. Control messages are not cached and only forwarded to one hop. Vanilla CCN forwarders are not affected by control messages because `/ust.hk/co-op` is not found in their cache nor forwarding rules. Therefore, our scheme is *backward compatible* with legacy systems, allowing AS operators to *partially deploy* our scheme, which is considered a practical advantage.

B. Cache Encodings

When a forwarder executes `evict()`, it is responsible for: *i)* Exchanging cache encodings with higher degree forwarders within one network hop, since they will be the only ones to hold the redundant content chunks. *ii)* Signaling those forwarders to hold redundant content chunks and evict the local chunks. *iii)* Installing special forwarding rules to redirect future requests for the evicted chunks.

Summary cache [4] demonstrated the efficiency of using Bloom filters to encode caches while supporting set-membership queries. In CCN, content chunks are uniquely represented by their names, thus it is sufficient to encode *content names* in Bloom filters. Every time cache encodings are needed, a new Bloom filter is created from scratch instead of *updating* the Bloom filter as the cache changes.

The challenge is that the accuracy of cache encodings *decay* with time as the encoded cache changes. To overcome this, the time between the Bloom filter creation and its use should be minimized. To this end, a 4-way handshake is designed to perform signaling and exchange cache encoding between *two* forwarders, using two types of control messages, `HOLD-REQ` (hold request, type `0x01`) and `HOLD-ACK` (hold acknowledgement, type `0x02`), as illustrated in Fig. 3.[4]

C. Lifetime of Cache Contents

The held content chunks are given a lifetime, after which they are amenable by cache replacement. When a content chunk is first being held, it is given an *initial* lifetime L_0 that allows popular content chunk to accumulate cache hits before expiration. The lifetime of contents is *extended* by the value L_1 whenever a content chunk receives a request (on a cache hit). Longer lifetime is given to popular content chunks.

If the number of cache misses per second is large, then the cache has failed to serve as many requests as possible. Hence,

[2] `deg()`, `seq()` returns the degree and sequence number of a forwarder.

[3] In the terminology of CCN, an *interest packet* requests for some content and *data packet* is returned carrying a *chunk* of that content.

[4] Cache encodings are large and the MTU (Maximum Transmission Unit) of the underlying network layer is likely to be exceeded. Hence, a *reliable protocol* is implemented to deliver the resulting chunks reliably.

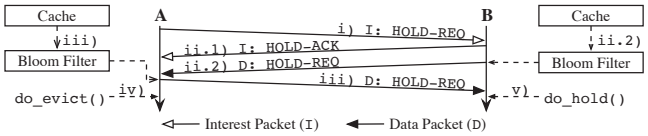

Fig. 3. Forwarder A initiates cache encoding exchange with forwarder B using 4-way handshake. i) A starts by sending B a `HOLD-REQ` interest. ii) B, after receiving `HOLD-REQ` interest, encodes the cache using Bloom filter and sends it to A using `HOLD-REQ` data packet and then sends A a `HOLD-ACK` interest. iii) Similarly, when A receives `HOLD-ACK` interest, its cache is encoded and the encoding is sent to B using `HOLD-ACK` data packet. iv) When A also receives `HOLD-REQ` data packets, it has everything from B, and evicts redundant content chunks, by running `do_evict()`. v) When B receives `HOLD-ACK` data packet from A, it has everything from A, and holds redundant content chunks, by running `do_hold()`.

L_0 is set to be inversely proportional to the *number of cache misses per second* which aims at reducing the lifetime of held contents and speeding up the cache replacements. Let h_0 be the tunable *hold initial constant* and μ be the moving average of the cache misses per second, then $L_0 = \frac{h_0}{\mu}$. Similarly, L_1 is inversely proportional to the time between two successive requests to account for the popularity of the content. Let h_1 be the tunable *hold extent constant*, t_{prev} be the time-stamp of previous request, and t_{curr} be the current time-stamp of the incoming request then $L_1 = \frac{h_1}{(t_{curr} - t_{prev})}$.

D. Request Redirection

Special forwarding rules called *Redirect Rules* are installed after evicting a content chunk. The Redirect Rule associates the name of the evicted content chunk to the *face index* of the forwarder that holds it. They have precedence over normal forwarding rules, and are associated with the content *lifetime*, such that they would expire when the corresponding chunk expires. Moreover, the number of cache misses per second is exchanged among forwarders using `DISC` control messages.

False Redirection refers to the event when an interest is redirected to a forwarder that does not cache the content chunk, yet it is identified as redundant because of Bloom filters' *false positives*. Redirected interest should *not* be forwarded again, since it could end up at *another* forwarder that does not cache the content chunk, resulting in *forwarding loops*, and False redirection errors could *accumulate*. A timer is maintained for each pending redirected interest to resend the interest without using Redirected Rules and redirected interests are dropped on cache misses. Since requests are redirected only to forwarders within one network hop, and not forwarded further, request redirection must provide a shorter path than retrieving the content chunk from the origin server.

When content chunks return from request redirection, they should *not* be cached to avoid the problem of cache redundancy. A flag bit is added to both the interest and data packets, and the modified forwarding logic is illustrated in Fig. 4.

IV. Implementation Details

A. Configuration of Bloom Filter Parameters

A Bloom filter is a m-bit bit vector associated with k independent hash functions. Suppose n names have been added, then the *false positive probability* p is given by $p = (1 - [1 - 1/m]^{kn})^k$. n is equal to the cache size since Bloom filters are used to encode caches. According to [4], the optimal number of hash functions $k*$ that minimizes p provided m and n is given by $k^* = (m \ln 2)/n$. Double hashing [5] can construct k hash functions from 2 hash functions, while keeping the same assymptotic p: All forwarders share the same *family* of hash functions, $h_i(x) = (g_1(x) + ig_2(x)) \mod m, i \in \{1, 2, ...\}$. g_1 and g_2 are the most and least significant 16 bits of the well-known 32-bit Fowler-Noll-Vo hash function. Thus it is sufficient to transfer the value of m, k, and the bit vector over the network as shown in Fig. 5(c).

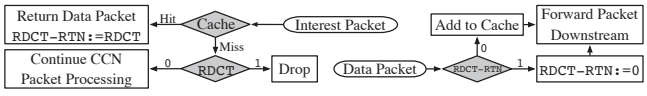

Fig. 4. Modified CCN packet processing logic. `RDCT-RTN` (redirect return) is introduced to data packets. It is 0 *except* the data packet is returned from request redirection. If it is 1, then the data packet is not cached. `RDCT` (redirect) is introduced to interest packets to indicate request redirection. It is 1 if the interest packet is forwarded using Redirect Rules. The data packet is returned with `RDCT-RTN` set to the value of `RDCT` set in the interest packet.

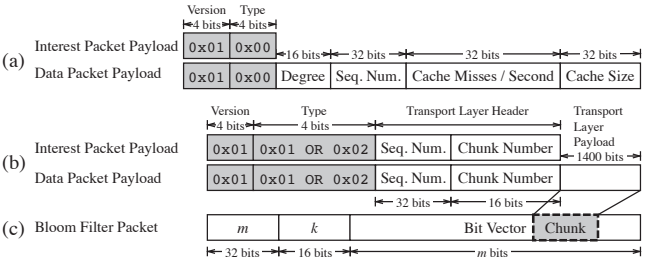

Fig. 5. (a) Packet format of `DISC` control message (type `0x00`). (b) Packet format of `HOLD-REQ` (type `0x01`) and `HOLD-ACK` (type `0x02`) control messages. (c) Packet format to transfer a large Bloom filter over the network, split into chunks that are carried by the reliable transport layer.

The content is typically split into chunks that can fit into a single packet as it is more communication efficient to fill a packet entirely. For example, the size of Ethernet packets is 1500 bytes. Assuming all headers including CCN and transport layer take at most 100 bytes, then 1400 bytes are usable as the *payload* in the transport layer. A packet is used for every 500 content chunks in the cache[5]. For a cache with 500 chunks, $m = (1400 - 6) * 8 = 11152$ bits, after subtracting 6-byte header for m and k in Fig. 5(c). $k* = 15, p = 2.23 \times 10^{-5}$.

Complexity Analysis. The forwarder needs to exchange Bloom filter between each one-hop neighbor forwardering, thus the message complexity is linear in the node degree. For each 500 cache slots, an extra packet is used to carry the Bloom filter. Thus, the message complexity is also stepwise-linear in the cache size.

B. Transport Layer

To achieve reliability, a transport layer is implemented on top of `HOLD-REQ` and `HOLD-ACK` control messages. The messages carry cache encodings and supports chunking to

[5]The transport layer needs to know how many Bloom filter chunks to request. This is obtained by first having forwarders exchange cache sizes using `DISC` control messages, then divide the cache size by 500.

perform reliable transfer similar to selective repeat and timeout report after the fifth retry.

Suppose forwarder A requests cache encoding from forwarder B by sending one interest for each chunk to B, with a *transport layer header* comprising: *i)* 16-bit *chunk number*. *ii)* 32-bit *sequence number*, used to invalidate *stale* interests since A periodically requests cache encoding from B. The sequence number is increased by one when A receives *all* chunks from B successfully. A timer is associated with each pending interest, when it elapses before data returns, the interest is retransmitted. After the fifth retransmission, *timeout* is reported to the 4-way handshake and the transfer is aborted. The last received sequence number is recorded at B. When an interest with greater sequence number is received, B *creates* a new cache encoding and interests with out-of-order sequence numbers are discarded. Otherwise, B returns the requested chunk of cache encoding. The packet format of `HOLD-REQ` and `HOLD-ACK` control messages are shown in Fig. 5(b).

V. Evaluation

Our scheme is implemented on a real-world production-ready code distribution, the CICN's Metis forwarder [3].

A. Experimental Settings

Fig. 6 shows the network topology, expanded from the router-level stub AS topology from [6]. A Metis forwarder instance is deployed to each router, represented by a virtualized container in a data center. As such we can build any topology within a data center by using `tc` tool for controlling the traffic. Thus, we can emulate and run on a real network with the real code of Metis instead of using an event driven simulation. Note that, in the experiments, caching is disabled at the ISP routers.

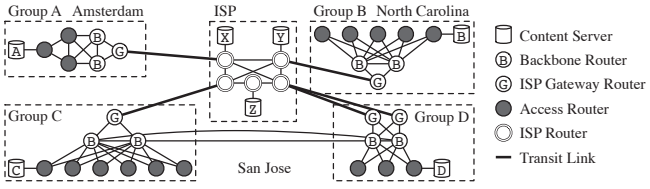

Fig. 6. Expanded stub AS topology used in evaluation, consisting three geographical regions and four groups connected by one ISP, whose connectivity is emulated by using five connected routers. Using `tc`, round-trip latencies in the ISP are set to the following: San Jose – North Carolina: 50 ms; North Carolina – Amsterdam: 100 ms; San Jose – Amsterdam: 150 ms.

Content servers and clients are placed across the topology to generate the CCN traffic. Each server serves content with a distinct prefix. To resemble content generated *within* the AS, a server is placed at an access router of each group. To maintain content from external origins, multiple servers are placed at ISP routers. At the remaining access routers (2 in Group A, 4 in Group B, 5 in Group C and 2 in Group D), an *internal client* is placed, requesting for content available internally and externally. To generate requests *coming from* external sources, an *external client* is placed at one of the ISP router, requesting content generated within the AS only.

There are three types of traffic for internal clients: *i)* Local traffic: Traffic served within the same group. *ii)* Internal traffic: Traffic crossing groups but within the AS. *iii)* External traffic: Traffic crossing the AS boundaries. Each client generates in total one million interests, requesting for content that are equally split among servers according to the *traffic patterns*, which are combinations of the three types of traffic below:

Traffic Pattern	Local Traffic	Internal Traffic	External Traffic
Mostly Local	50%	25%	25%
Mostly Internal	25%	50%	25%
Mostly External	15%	10%	75%

Thus, a total of 14 million interests are generated. Each client receives one million chunks, assuming one chunk per content. Interest generation follows Poisson process with mean rate $\lambda = 500$ interests per second. Forwarding rules are manually inserted to Metis forwarders using the equal-cost multi-path routing strategy. A random face is chosen uniformly if more than one face is available.

The Zipf–Mandelbrot distribution is used for the per-chunk popularity distribution which has the following p.m.f.: $f(k; N, q, s) = \frac{1/(k+q)^s}{\sum_{j=1}^{N} 1/(j+q)^s}$, where N is the number of unique objects in the content universe, s is the *skewness* parameter, while the popularity of the first q content chunks are largely similar. For each server, requests are generated by the algorithm described in ProwGen [7], where the fraction of *unique* content chunk is 0.3, and the fraction of *one-timers* is 0.7 as recommended. Each client requests in total $1,000,000 \times 0.3 = 300,000$ *unique* content chunks, where $300,000 \times 0.7 = 210,000$ chunks are requested *only once*.

B. Experimental Results

Default Parameters	**Value**	**Default Parameters**	**Value**
Cache Size (No. of Entries)	1500	Eviction Interval (ms)	1500
Zipf–Mandelbrot *Skewness*	0.9	Zipf–Mandelbrot q	0
Hold Initial Constant h_0	2000	Hold Extend Constant h_1	100

Default parameters are shown in the table above. Each setting is run for 5 times with different random seeds to obtain 95% confidence interval for the metrics. Performance is compared with ubiquitous–LRU, since it is the only available caching strategy implemented in Metis.

Fig. 7(a,b) shows the performance compared to ubiquitous–LRU. We achieve up to 15% *reduction* of server load, quantified by the number of interests that reach the content servers. As a result, we achieve up to 15% *reduction* of traffic volume in transit links, regardless of cache sizes and skewness in the Zipf–Mandelbrot distribution. In general, we perform *better* when: *i)* Cache sizes are large. *ii)* The content popularity distribution is more *skewed* (higher skewness and lower q). This is because with larger cache sizes, or more skewed Zipf–Mandelbrot distributions, the cache can absorb a greater amount of popular content, and precisely hold the content chunks that are actually more popular.

Fig. 7(c) shows the traffic overhead, which is the proportion of traffic volume of control messages compared to the

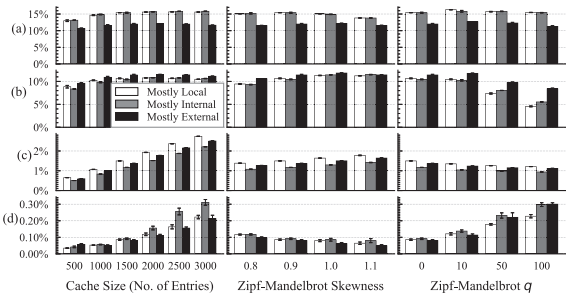

Fig. 7. Performance and overhead of our proposed scheme. (a) Reduction of server load compared to ubiquitous–LRU (the higher the better) . (b) Reduction of transit link traffic compared to ubiquitous–LRU (the higher the better) (c) Traffic overhead incurred by control messages. (d) Ratio of False Redirection compared with the total number of interests redirected.

total traffic volume, is less than 3%. This does not include additional traffic resulting from False redirection. There is a linear increase of traffic overhead with respect to cache size, due to the number of packets used to carry cache encoding scaling linearly with cache sizes. The number of False Redirects (Fig. 7(d)) are insignificant (i.e., less than 0.35% of total interests redirected). These results show the benefits of employing the 4-way handshake and configuration of Bloom filter parameters in the system design.

VI. Related Work

Caching is arguably the area that has received the most attention from the CCN community. In [8], the PCP caching scheme is proposed to avoid caching unpopular data (especially the predominant one-timers) at access routers and bring popular data to end users in a progressive way. Extending the concept of centrality, [9] introduces content popularity and cache placement to betweenness and closeness centrality, and proposed a caching scheme that aims at increasing such centralities. A lightweight method to approximate betweenness centrality without global topology knowledge is proposed in [10]. It is used to cache data at node(s) along the request-return path with highest centrality. In [11], attenuated Bloom filters are used to retrieve content cached at multiple hops away and eliminate redundant content at one hop away, where the cache is split into two partitions to ensure availability for retrieval. Taking advantage of in-network caching, [12] describes proactive caching under mobility using entropy as an indicator to place content while limiting redundancy. Object caching is introduced in [13], that aims at caching a sequence of continuous chunks from the start of an object. In [14] an analysis of looped replacement is conducted, where requests to chunks of an object results in cache misses despite the cache containing part of the object. The article also evaluated different cache admission policies which would suppress looped replacement. Using consistent hashing, forwarders in [15] are only responsible for caching disjoint subsets of content. Caching is modeled as the ski-rental problem, and a random online algorithm is proposed. [16] proposed an efficient Convolution Neural Network (CNN) based cache manager. [17] modeled the Pending Interest Table (PIT) which plays integral part in the dynamics of the cache and [18] proposed a congestion controller for the PIT to regulate the traffic incoming to the cache. In [19], NDN is applied to the area of content delivery networks (CDN) and a CDN that is built on top of NDN is proposed.

VII. Conclusions

In this paper, we present the design and implementation of a two-tiered (on- and off-path) caching scheme for CCN. The on-path caching simply relies on ubiquitous-LRU for its simplicity whereas the off-path tier deploys cooperative redundancy elimination and caching to increase the utility of the caches. We discuss several design choices and present control protocols to make it possible to deploy the scheme in a real CCN network. Our protocols do not rely on TCP/IP and adopt CCN itself for communication. We extended the Metis forwarder of the CICN project with these protocol and experimented with it in a data center where we emulated a stub AS network topology. The results show considerable improvements compared to the ubiquitous-LRU on many performance metrics. Porting the scheme to the faster VPP implementation of CCN is part of our future work plan.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton *et al.*, "Networking Named Content," in *ACM CoNEXT*, 2009.

[2] J. M. Wang, J. Zhang, and B. Bensaou, "Intra-AS Cooperative Caching for Content-Centric Networks," in *ACM ICN*, 2013.

[3] The Linux Foundation. (2021) Fast Data project (fd.io) Community ICN (CICN). [Online]. Available: https://wiki.fd.io/view/Cicn

[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*, vol. 8, no. 3, pp. 281–293, 2000.

[5] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," *Random Structures & Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[6] P. Mérindol, V. Van den Schrieck *et al.*, "Quantifying Ases Multiconnectivity Using Multicast Information," in *ACM IMC*, 2009.

[7] M. Busari and C. Williamson, "ProWGen: a synthetic workload generation tool for simulation evaluation of web proxy caches," *Computer Networks*, vol. 38, no. 6, pp. 779–794, 2002.

[8] J. M. Wang and B. Bensaou, "Progressive caching in CCN," in *IEEE GLOBECOM*, 2012.

[9] J. A. Khan *et al.*, "NICE: Network-Oriented Information-Centric Centrality for Efficiency in Cache Management," in *ACM ICN*, 2018.

[10] J. Pfender *et al.*, "Easy as ABC: A Lightweight Centrality-Based Caching Strategy for Information-Centric IoT," in *ACM ICN*, 2019.

[11] T. Mick *et al.*, "MuNCC: Multi-Hop Neighborhood Collaborative Caching in Information Centric Networks," in *ACM ICN*, 2016.

[12] N. Abani *et al.*, "Proactive Caching with Mobility Prediction under Uncertainty in Information-Centric Networks," in *ACM ICN*, 2017.

[13] Y. Thomas, G. Xylomenos, C. Tsilopoulos *et al.*, "Object-Oriented Packet Caching for ICN," in *ACM ICN*, 2015.

[14] Y. Yamamoto *et al.*, "Analysis on Caching Large Content for Information Centric Networking," in *ACM ICN*, 2018.

[15] K. Thar *et al.*, "Online Caching and Cooperative Forwarding in Information Centric Networking," *IEEE Access*, vol. 6, pp. 59 679–59 694, 2018.

[16] K. H. Chiu, J. Zhang, and B. Bensaou, "Cache Management in Information-Centric Networks using Convolutional Neural Network," in *IEEE GLOBECOM*, 2020.

[17] A. J. Abu, B. Bensaou, and A. M. Abdelmoniem, "A Markov Model of CCN Pending Interest Table Occupancy with Interest Timeout and Retries," in *IEEE ICC*, 2016.

[18] A. J. Abu, B. Bensaou, and A. M. Abdelmoniem, "Leveraging the Pending Interest Table Occupancy for Congestion Control in CCN," in *IEEE LCN*, 2016.

[19] C. Ghasemi *et al.*, "ICDN: An NDN-Based CDN," in *ACM ICN*, 2020.