MC3: A Cloud Caching Strategy for Next Generation Virtual Content Distribution Networks

Pietro Marchetta*, Jaime Llorca†, Antonia M. Tulino*†, Antonio Pescapé*‡
*Universitá di Napoli Federico II and ‡NM2 Srl, Italy. Email: {pietro.marchetta, pescape}@unina.it
†Bell Labs, Nokia, NJ, USA. Email: {jaime.llorca, a.tulino}@nokia.com

***Abstract*—With the advent of network functions virtualization and software defined networking, cloud content distribution network (CDN) providers can auto-scale their virtual CDN appliances in order to meet changing demands for commercial and user generated content services in a cost and energy efficient manner. However, existing caching policies, constrained to work with dedicated CDN resources and designed to maximize local cache hit rates, do not exploit the elasticity of virtualized cloud environments to adaptively guarantee service requirements with minimum cost. In this paper, we design and evaluate MC^3 (MinCostCloudCache), an adaptive distributed caching strategy whose fundamental goal is to guarantee content service requirements while minimizing the use and associated cost of the shared physical infrastructure. MC^3 estimates the global benefit of caching an object at a network node using only locally available information. The caching benefit is flexible and adaptive to the particular content service requirements, and is aware of the behavior of neighbor network caches, creating effective cache cooperation using only local information. Through simulation, we show how MC^3 not only reduces the experienced average delay with respect to existing caching policies, but it also uses significantly less storage and transport resources, leading to increased revenues and reduced operational costs.**

I. Introduction

Content distribution networks (CDNs) are highly distributed systems consisting of globally dispersed cache servers that allow hosting and delivering content items close to the end users, thus providing improved user experience and reduced transport costs. Traditional CDNs are composed of dedicated hardware appliances that the operator dimensions according to estimated peak demands. For a given dedicated CDN deployment, the extent to which the benefits of network caching can be obtained depends crucially on the efficacy of the implemented content caching strategy. Given that today's CDNs are composed of a relatively small number of fixed-size hierarchical caches, content caching strategies are typically designed to maximize local hit rates, or the fraction of requests served by a given cache, which has been shown to be a good proxy for average delay in hierarchical CDNs [1]. However, the increasing dynamics and heterogeneity of content types, popularity, and service requirements, challenge the efficiency of traditional CDNs, in which dedicated storage and delivery appliances need to be pre-provisioned based on estimated peak demands, resulting in excessive over-provisioning and/or degraded quality of service (QoS).

ISBN 978-3-901882-83-8 © 2016 IFIP

With current advances in network virtualization and programmability, network operators have the opportunity to design their content distribution solutions in the form of elastic virtual networks over a common cloud network infrastructure [2]-[4]. In this way, operators can adaptively optimize the combined use of storage and transport resources to meet application requirements with minimum cost. In this attractive scenario, existing caching solutions, designed to work with fixed-size dedicated CDN appliances, cannot exploit the flexibility of evolved virtualized cloud environments nor the properties of the different content services to adaptively guarantee service requirements with minimum use of the shared physical infrastructure. We therefore argue for a shift in the design of content caching strategies for future cloud CDNs, driven by the main goal of minimizing the overall network's operational cost while guaranteeing QoS requirements.

A. Contributions

In this paper, we look at content caching in the context of next generation virtual CDNs that can host a variety of content services and elastically scale their network resources. Our approach – instead of just maximizing local hit rates for typically small hierarchical fixed-size CDNs – aims at guaranteeing content services' QoS requirements with minimum overall use of the shared cloud network's infrastructure.

The contributions of this work are fourthfold. First, we analytically characterize the optimal cloud caching policy for a given first-order stationary input process (Sec. II and Sec. III). Second, based on the structure of the optimal stationary solution, we propose MC^3 (Minimum Cost Cloud Cache), a fully distributed cloud caching algorithm targeting optimal caching decisions based on local estimates of the global cost benefit (Sec. IV). MC^3 provides effective cache cooperation with negligible overhead via the adaptive learning of transport costs to access neighboring replicas and local content popularity. Specifically, with MC^3, network nodes: *i*) infer neighbors' actions from information in object arrivals; *ii*) infer local content popularity from information in request arrivals; and *iii*) compute the benefit of caching an object at a particular location based on the overall cost reduction it can provide. Third, we implement the proposed caching strategy in a custom-built discrete event simulator (Sec. V-A). Fourth, we perform a comparison with a number of well known caching strategies (LRU-LCE, Perfect-LFU, an Oracle), demonstrating the significant performance and efficiency gains that MC^3 can

provide in virtual CDN environments (Sec. V-B). We show the superiority of MC^3 in terms of average delivery delay and overall operational cost with varying transport-to-storage cost ratio, cache size, and content popularity settings.

B. Related Work

A substantial amount of research has been devoted to the content distribution problem (CDP), where the goal, in its most general setting, is to find the placement and routing of content objects in an arbitrary capacitated network that minimizes the combined transport and storage cost while satisfying possible delivery deadline constraints. The authors in [5] provide a comprehensive complexity classification of the CDP. Interestingly, while NP-Hard in general, the CDP is shown to be polynomial-time solvable in tree networks and in arbitrary networks that allow coding between objects of the same requested content. The work in [6] addresses the CDP in realistic AS-level topologies showing how the footprint of dedicated CDNs must expand to accommodate increasingly tighter user requirements. A number of works have addressed the design of approximated solutions for the CDP, mostly relying on LP-relaxation techniques (e.g., [7], [8], [9]) or greedy algorithms (e.g., [10], [11], [12]) that exploit special assumptions such as uniform object sizes, network symmetry, and hierarchical topologies. The solutions to the CDP are centralized and proactive, in the sense that content placement decisions are made based on global estimates of the users' average demands (e.g., content popularity) over a given time period, typically in the range of hours or even days. Network caches are then updated to best satisfy future requests over the given time period. The performance of centralized proactive solutions heavily depends on the system dynamics and its corresponding prediction accuracy. In fact, with the increasing volatility and unpredictability of next generation content services, errors in the popularity estimates and the overhead associated to frequent cache updates can significantly degrade the performance of centralized proactive solutions.

In highly dynamic and unpredictable scenarios, content distribution solutions must resort to distributed reactive algorithms that adapt to fast changes in content popularity with minimal overhead. An extensive line of work has also been devoted to the distributed dynamic content replacement problem, where the objective is to adaptively refresh the network caches as content objects travel through the network (e.g., [13], [14], [15]). The most common cache replacement algorithms are LRU (Least Recently Used) and LFU (Least Frequently Used), by which the least frequently/recently used content object is evicted upon arrival of a new object to a network cache. Due to its simplicity, LRU is the most widely used caching algorithm in today's CDNs and the most analyzed in the context of emerging paradigms such as ICN (see [16], [17] and references therein). These studies show the benefits of LRU-based caching to reduce dissemination latency and network load, and the little improvements provided by alternative caching policies proposed to date. However, as pointed out earlier, existing caching policies have been designed and compared against hit-rate performance metrics, ignoring the operational cost associated to the use of storage and transport resources. Hence, motivated by the increasing adaptability and programmability associated to the configuration of next generation cloud networks, and the dynamics and heterogeneity of next generation content services, we argue that centralized proactive content distribution solutions must be complemented with distributed reactive caching algorithms that are designed to achieve global system objectives, such as overall cloud network operational efficiency, via simple local interactions that incur minimal overhead.

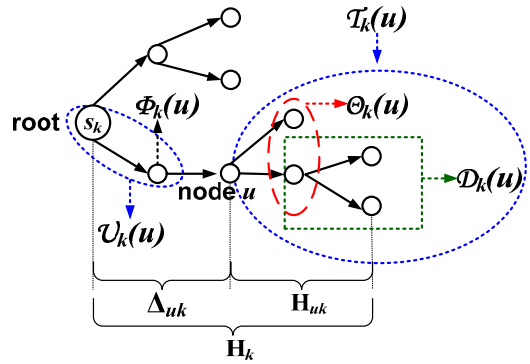


Fig. 1: The routing tree for object k, $\mathcal{T}_k \subset \mathcal{G}$, rooted at the source or repository of k, s_k, for a given time period. $\Phi_k(u)$ denotes node u's upstream neighbor in $\mathcal{T}_k$.

II. Network Model

In a distributed cloud network architecture, a virtual CDN consists of a set of virtual caches (vCache), implemented as virtual network functions (VNFs) in an NFV framework. The vCache nodes are interconnected by virtual links (vLink), representing the logical connectivity. We model a virtual CDN as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with V vCaches and E vLinks. We assume content items are partitioned into equal-size objects $k \in \mathcal{K}$ and denote by λ_{uk} the exogenous average request rate for object $k \in \mathcal{K}$ at node $u \in \mathcal{V}$ during a specified time period. We denote by e_u^{st} the per-object storage cost of vCache $u \in \mathcal{V}$ and e_{uv}^{tr} the per-object transport cost of vLink $(u, v) \in \mathcal{E}$.

Motivated by the different time-scales at which routing and caching operate, here we do not address routing optimization and assume that the goal is to design a caching strategy for a given routing policy. We denote by $\mathcal{T}_k = (\mathcal{V}, \mathcal{E}_k)$ the routing tree rooted at the source or repository of k, s_k, as shown in Figure 1. It will also be useful to define $\mathcal{T}_k(u)$ as the set of nodes in the subtree of $\mathcal{T}_k$ rooted at u, $\mathcal{D}_k(u)$ as the nodes downstream of u, $\mathcal{U}_k(u)$ as the nodes upstream of u, $\Theta_k^j(u)$ as the j-th hop downstream neighbors of u, $\phi_k^j(u)$ as the j-th hop upstream neighbor of u (since most of the time we will refer to the one hop neighbors of u, we denote $\Theta_k(u) \equiv \Theta_k^1(u)$ and $\phi_k(u) \equiv \phi_k^1(u)$), H_k as the height of $\mathcal{T}_k$, H_{uk} as the height of $\mathcal{T}_k(u)$, and Δ_{uk} as the depth of u in $\mathcal{T}_k$, as shown in Fig. 1. We refer to e_{uk} as the unit transport cost of link $(\phi_k(u), u)$.

III. Optimal stationary policy

In this section, we analytically characterize the optimal cloud caching policy under the setting of stationary request process and sufficiently large storage capacity. Note that the "sufficiently large storage capacity" is a reasonable assumption in a cloud CDN, for which virtual storage may be largely available, but whose usage comes at a cost. While these assumptions may not always hold in practice, the structure of the resulting optimal policy will show extremely useful in driving the design of the proposed general caching strategy described in Sec. IV.

Under the assumption of a first-order stationary request process, we focus on designing a stationary caching policy that minimizes the average CDN cost. Letting $\mathbf{x} = \{x_{uk}\}$ denote a stationary caching configuration, with $x_{uk} = 1$ if object k is cached at node u, and $x_{uk} = 0$ otherwise, we seek the caching configuration $\mathbf{x}^*$ that satisfies

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \mathcal{C}(\mathbf{x}), \tag{1}$$

where

$$\mathcal{C}(\mathbf{x}) = \sum_{u\in\mathcal{V}} \sum_{k\in\mathcal{K}} \left(e_u^{st} x_{uk} + e_{uk}^{tr} \lambda_{uk} \left(1 - x_{uk}\right)\right), \tag{2}$$

$$e_{uk}^{tr} = \sum_{j=0}^{h_{uk}-1} e_{\phi_k^j(u)k}^{tr}, \qquad (\phi_k^0(u) \equiv u), \tag{3}$$

$$h_{uk} = \sum_{j=1}^{\Delta_{uk}} j\, x_{\phi_k^j(u)k} \prod_{p=0}^{j-1} \left(1 - x_{\phi_k^p(u)k}\right). \tag{4}$$

The average CDN cost, $\mathcal{C}(\mathbf{x})$, in (2), is computed as the sum over all nodes and objects of two mutually exclusive terms. The first term is the storage cost if k is cached at u, and the second term is the transport cost incurred in fetching k from the closest upstream copy at a rate λ_{uk}, if k is not cached at u. The variable e_{uk}^{tr} denotes the cost of the path from u to the closest upstream node caching k, and h_{uk} is the number of hops from u to the closest upstream node caching k.

Note that without capacity constraints, i.e., under the assumption of "sufficiently large storage capacity", (1) can be solved independently for each object $k \in \mathcal{K}$. Let $\mathcal{C}_{uk}$ denote the total cost for the delivery of object k over the subtree $\mathcal{T}_k(u)$ when u stores a copy of k:

$$\mathcal{C}_{uk} = e_u^{st} + \mathcal{C}_{uk}^{(0)}, \tag{5}$$

where, for all $u \in \mathcal{V}$ and $h = \{0, \ldots, H_k\}$,

$$\mathcal{C}_{uk}^{(h)} = \sum_{v\in\Theta_k(u)} \mathcal{C}_{vk} x_{vk} + \left(e_{vk}^{tr} \lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right)(1 - x_{vk}) \tag{6}$$

is the total cost of the subtree $\mathcal{T}_k(u)$ when the closest upstream node caching k is h hops from u, and

$$\lambda_{vk}^{(h)} = \lambda_{vk} + \sum_{w\in\Theta_k(v)} \lambda_{wk}^{(h+1)} \left(1 - x_{wk}\right) \tag{7}$$

is the aggregate rate of requests for k at node v when the closest upstream node caching k is h hops from v.

TABLE I: Summary of the main variables used for the analysis of OSC, in addition to the routing tree variables in Fig. 1.

$\{x_{uk}\}$	stationary cache configuration, with $x_{uk} = 1$ if object k is cached at node u, and $x_{uk} = 0$ otherwise
e_u^{st}	per-object storage cost of node u
e_{uv}^{tr}	per-object transport cost of link (u, v)
λ_{uk}	request rate of object k at node u
h_{uk}	number of hops from node u to the closest upstream node caching object k
e_{uk}^{tr}	transport cost of transferring object k to node u form the closest upstream node caching object k
$\lambda_{uk}^{(h)}$	rate of requests for object k at node u when the closest upstream node caching object k is h hops from node u
$\mathcal{C}_{uk}$	total cost for the delivery of object k over subtree $\mathcal{T}_k(u)$ when node u is caching object k
$\mathcal{C}_{uk}^{(h)}$	total cost for the delivery of object k over subtree $\mathcal{T}_k(u)$ when the closest upstream node caching object k is h hops from node u

In the following, we present OSC (Optimal Stationary Cache) - see Tab. I for the details on the adopted notation - a dynamic programming algorithm that computes the minimum cost for the delivery of object k over $\mathcal{T}_k$, $\mathfrak{C}_k$, as

$$\mathfrak{C}_k = \mathfrak{C}_{uk}\Big|_{u=s_k}, \tag{8}$$

with

$$\mathfrak{C}_{uk} = e_u^{st} + \mathfrak{C}_{uk}^{(0)}, \tag{9}$$

where, for all $u \in \mathcal{V}$ and $h = \{0, \ldots, H_k\}$,

$$\mathfrak{C}_{uk}^{(h)} = \sum_{v\in\Theta_k(u)} \min\left\{\mathfrak{C}_{vk}\,,\ e_{vk}^{tr} \mu_{vk}^{(h+1)} + \mathfrak{C}_{vk}^{(h+1)}\right\}, \tag{10}$$

$$\mu_{vk}^{(h)} = \lambda_{vk} + \sum_{w\in\Theta_k(v)} \mu_{wk}^{(h+1)} \left(1 - \mathfrak{X}_{wk}^{(h+1)}\right), \tag{11}$$

and

$$\mathfrak{X}_{wk}^{(h)} = \begin{cases} 1 & \text{if} \quad e_{wk}^{tr,h} \mu_{wk}^{(h)} + \mathfrak{C}_{wk}^{(h)} \geq \mathfrak{C}_{wk} \\ 0 & \text{otherwise} \end{cases}, \tag{12}$$

with $e_{wk}^{tr,h}$ defined as

$$e_{uk}^{tr,h} = \sum_{j=0}^{h-1} e_{\phi_k^j(u)k}^{tr}. \tag{13}$$

Note that (5)–(7) are the equivalent of (9)–(11) when evaluated at the solution of OSC given by (12).

The optimality of OSC is based on the following theorem.

Theorem 1: $\mathfrak{C}_k$, as defined in (8), is the minimum cost for the delivery of object k over $\mathcal{T}_k$ with average request rates $\lambda_{uk}, \forall u \in \mathcal{V}$.

Proof. In order to prove Theorem 1, we first state and prove the following Lemma.

Lemma 1: For all $u \in \mathcal{V}$, $\mathfrak{C}_{uk}^{(h)}$ is the minimum cost over $\mathcal{T}_k(u)$ for the delivery of object k, given that the closest upstream node caching object k is h hops away from u, *i.e.,* for all $h = \{0, \ldots, H_k\}$

$$\mathfrak{C}_{uk}^{(h)} = \min_{\mathbf{x}} \left\{\mathcal{C}_{uk}^{(h)}\right\}. \tag{14}$$

Proof. To prove Lemma 1, we proceed by induction on the tree height. Let $\mathcal{H}(j)$ be the set of nodes at height j in $\mathcal{T}_k$.

First, we prove that the claim of Lemma 1 holds at the bottom of the tree. In fact, for all subtrees rooted at $u \in \mathcal{H}(1)$,

$$\begin{aligned}
&\min_{\mathbf{x}} \left\{\mathcal{C}_{uk}^{(h)}\right\} = \\
&= e_u^{st} + \min_{\mathbf{x}} \sum_{v\in\Theta_k(u)} \left\{\mathcal{C}_{vk} x_{vk} + \left(e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right)(1-x_{vk})\right\} \\
&= e_u^{st} + \min_{\mathbf{x}} \sum_{v\in\Theta_k(u)} \left\{e_v^{st} x_{vk} + \left(e_{vk}^{tr}\lambda_{vk}\right)(1-x_{vk})\right\} \qquad (15)\\
&= e_u^{st} + \sum_{v\in\Theta_k(u)} \min\left\{e_v^{st},\, e_{vk}^{tr}\lambda_{vk}\right\} \qquad (16)\\
&= \mathfrak{C}_{uk}. \qquad (17)
\end{aligned}$$

where (15) follows from $\mathcal{C}_{vk} = e_v^{st}$, $\mathcal{C}_{vk}^{(h)} = 0$ and $\lambda_{vk}^{(h)} = \lambda_{vk}$ for all leaf nodes $v \in \Theta_k(u) \subset \mathcal{H}(0)$ and for all $h = \{0, \ldots, H_k\}$; (16) follows from the fact that each term in the summation only depends on x_{vk}; and finally (17) follows from $\mathfrak{C}_{vk}^{(h)} = 0$, $\mathfrak{C}_{vk} = e_v^{st}$, and $\mu_{vk}^{(h)} = \lambda_{vk}$ for all leaf nodes $v \in \mathcal{H}(0)$.

Next, we prove that if for all $v \in \mathcal{H}(\ell)$, $\ell = 0, \ldots, (j-1)$,

$$\mathfrak{C}_{vk}^{(h)} = \min_{\mathbf{x}} \left\{\mathcal{C}_{vk}^{(h)}\right\}, \qquad (18)$$

then it also holds that

$$\mathfrak{C}_{uk}^{(h)} = \min_{\mathbf{x}} \left\{\mathcal{C}_{uk}^{(h)}\right\}, \quad \forall u \in \mathcal{H}(j). \qquad (19)$$

To this end, notice that using (6),

$$\min_{\mathbf{x}}\left\{\mathcal{C}_{uk}^{(h)}\right\} = \sum_{v\in\Theta_k(u)} \min\left\{\min_{\mathbf{x}}\{\mathcal{C}_{vk}\}, \min_{\mathbf{x}}\left\{e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right\}\right\} \qquad (20)$$

$$= \sum_{v\in\Theta_k(u)} \min\left\{\mathfrak{C}_{vk,\mathbf{x}}, \left\{e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right\}\right\} \qquad (21)$$

where (20) is due to the fact that when the closest upstream node caching k is h hops away from v, then the optimal configuration of $\mathcal{T}_k(v)$ can be found by solving independently the optimal configuration for each of the subtrees rooted at $w \in \Theta_k(v)$; and (21) follows from the induction step in (18).

Now, we prove by *reductio ad absurdum* that

$$\min_{\mathbf{x}} \left\{e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right\} = e_{vk}^{tr}\mu_{vk}^{(h+1)} + \mathfrak{C}_{vk}^{(h+1)}. \qquad (22)$$

To this end, first note that the non-strict inequality always holds:

$$\min_{\mathbf{x}} \left\{e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right\} \le e_{vk}^{tr}\mu_{vk}^{(h+1)} + \mathfrak{C}_{vk}^{(h+1)}; \qquad (23)$$

in fact, the right hand side of (23) is the cost of the subtree $\mathcal{T}_k(v)$ when k is cached $h+1$ hops away from v plus the cost of the upstream link (u, v), computed for the caching configuration given by (12), while the left hand side is the minimum over all possible caching configurations of the above function. Next, let us verify that strict inequality in (23) leads to a contradiction. To this end, assume (23) is strict, and let $\bar{\mathbf{x}} = \arg\min_{\mathbf{x}} \left\{e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\right\}$. Since, $\min_{\mathbf{x}} \left\{\mathcal{C}_{vk}^{(h+1)}\right\} = \mathfrak{C}_{vk}^{(h+1)}$ by induction, and $\mathfrak{C}_{vk}^{(h+1)} = \mathcal{C}_{vk}^{(h+1)}\Big|_{\mathfrak{X}}$ from (8)-(12), then

$$\mathcal{C}_{vk}^{(h+1)}\Big|_{\bar{\mathbf{x}}} \ge \mathfrak{C}_{vk}^{(h+1)} = \mathcal{C}_{vk}^{(h+1)}\Big|_{\mathfrak{X}}. \qquad (24)$$

From (6) and (7), we now have that

$$\begin{aligned}
&e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\Big|_{\bar{\mathbf{x}}} = \\
&= e_{vk}^{tr}\lambda_{vk} + \sum_{w\in\Theta_k(v)} \min\left\{\mathcal{C}_{wk}\Big|_{\bar{\mathbf{x}}}, \left(\left(e_{vk}^{tr} + e_{wk}^{tr}\right)\lambda_{wk}^{(h+2)} + \mathcal{C}_{wk}^{(h+2)}\right)\Big|_{\bar{\mathbf{x}}}\right\} \\
&= e_{vk}^{tr}\lambda_{vk} + \sum_{w\in\Theta_k^{\mathcal{C}_1}(v)} \mathcal{C}_{wk}\Big|_{\bar{\mathbf{x}}} + \sum_{w\in\Theta_k^{\mathcal{C}_2}(v)} \mathcal{C}_{wk}\Big|_{\bar{\mathbf{x}}} \\
&\quad + \sum_{w\in\Theta_k^{n\mathcal{C}_1}(v)} \left((e_{vk}^{tr} + e_{wk}^{tr})\lambda_{wk}^{(h+2)} + \mathcal{C}_{wk}^{(h+2)}\right)\Big|_{\bar{\mathbf{x}}} \\
&\quad + \sum_{w\in\Theta_k^{n\mathcal{C}_2}(v)} \left((e_{vk}^{tr} + e_{wk}^{tr})\lambda_{wk}^{(h+2)} + \mathcal{C}_{wk}^{(h+2)}\right)\Big|_{\bar{\mathbf{x}}} \qquad (25)
\end{aligned}$$

where $\{\Theta_k^{\mathcal{C}_1}(v), \Theta_k^{\mathcal{C}_2}(v), \Theta_k^{n\mathcal{C}_1}(v), \Theta_k^{n\mathcal{C}_2}(v)\}$ is a partition of $\Theta_k(v)$ such that:

$$\begin{aligned}
\Theta_k^{\mathcal{C}_1}(v) &\equiv \{w \in \Theta_k(v) :\ \mathfrak{X}_{wk}^{st,h+2} = 0 \text{ and } \bar{x}_{wk}^{st} = 1\},\\
\Theta_k^{\mathcal{C}_2}(v) &\equiv \{w \in \Theta_k(v) :\ \mathfrak{X}_{wk}^{st,h+2} = 1 \text{ and } \bar{x}_{wk}^{st} = 1\},\\
\Theta_k^{n\mathcal{C}_1}(v) &\equiv \{w \in \Theta_k(v) :\ \mathfrak{X}_{wk}^{st,h+2} = 1 \text{ and } \bar{x}_{wk} = 0\},\\
\Theta_k^{n\mathcal{C}_2}(v) &\equiv \{w \in \Theta_k(v) :\ \mathfrak{X}_{wk}^{st,h+2} = 0 \text{ and } \bar{x}_{wk} = 0\}.
\end{aligned}$$

Based on how the four regions are defined, using (24), the induction step (18), and the definitions of $\mathfrak{C}_{vk}^{(h+1)}$, $\mathfrak{X}_{wk}^{st,h+2}$, $\lambda_{vk}^{(h+1)}$ and $\mu_{vk}^{(h+1)}$ as in (6)-(12), it can be shown that:

$$\begin{aligned}
e_{vk}^{tr}\lambda_{vk}^{(h+1)} + \mathcal{C}_{vk}^{(h+1)}\Big|_{\bar{\mathbf{x}}} &\ge e_{vk}^{tr}\lambda_{vk} + \sum_{w\in\Theta_k^{\mathcal{C}_2}(v)\cup\Theta_k^{n\mathcal{C}_1}(v)} \mathfrak{C}_{wk} \\
&\quad + \sum_{w\in\Theta_k^{\mathcal{C}_1}(v)\cup\Theta_k^{n\mathcal{C}_2}(v)} \left(e_{vk}^{tr} + e_{wk}^{tr}\right)\mu_{wk}^{(h+2)} + \mathfrak{C}_{wk}^{(h+2)}\Big) \\
&= e_{vk}^{tr}\mu_{vk}^{(h+1)} + \mathfrak{C}_{vk}^{(h+1)}. \qquad (26)
\end{aligned}$$

Using (26) and (23), we show that a strict inequality in (23) leads to a contradiction, and hence (22) is proved. Replacing (22) in (21) and using (10), (19) follows. ■

Using Lemma 1, it immediately follows that:

$$\begin{aligned}
\min_{\mathbf{x}} \{\mathcal{C}_{uk}(\mathbf{x}_k)\} &= \left(e_u + \min_{\mathbf{x}} \left\{\mathcal{C}_{uk}^{(0)}\right\}\right)\Big|_{u=s_k} \\
&= e_u + \mathfrak{C}_{uk}^{(0)}\Big|_{u=s_k} \\
&= \mathfrak{C}_{uk}\Big|_{u=s_k} = \mathfrak{C}_k
\end{aligned}$$

which concludes the proof of Theorem 1. ■

We note that the complexity of OSC is linear with the product of the number of nodes and the height of the tree, $O(VH_k)$, and thus can find the optimal configuration for the delivery of $\mathcal{K}$ over $\mathcal{G}$ in $\sum_{k\in\mathcal{K}} O(VH_k) \leq O(V^2\Delta)$, with Δ the diameter of $\mathcal{G}$. Furthermore, OSC admits a distributed implementation, which requires $O(H_k - H_{uk})$ information exchange between each node $u \in \mathcal{V}$ at height H_{uk} and its upstream node $\phi_k(u)$.

However, the optimality of OSC is constrained to the availability of sufficiently large storage capacity and the stationarity of the input request process. While, as stated earlier, large storage capacity may be available in cloud CDNs, the increasing dynamics of content service demands can degrade the performance of OSC in practice. In the following, we propose MC^3, a fully distributed dynamic caching policy that builds on the structure of OSC, to drive local caching decisions that adapt to the system dynamics, while completely eliminating the need for any explicit exchange of information between neighbor nodes. In particular, with MC^3, local caching decisions are based on the binary criterion described in (12), where information about the closest upstream content copies and the caching configuration of the downstream nodes is inferred from the dynamic arrivals of requests and objects themselves. A detailed description of MC^3 and its key mechanisms are provided in Sec. IV. Also, while omitted due to space limitations, it is worth mentioning that for a hierarchical topology with homogeneous resources at each layer, under a first-order stationary request process, it can be shown that MC^3 achieves the optimal steady-state configuration given by OSC.

IV. MC^3: Algorithm Design

In this section, we describe MC^3 (Min Cost Cloud Cache), an on-line fully distributed cloud caching algorithm that allows nodes to make local caching decisions based on real-time estimates of the global cost benefit. Recall that in a cloud CDN, the goal is to guarantee QoS requirements (e.g., average delivery delay) while minimizing the overall operational cost. Hence, in MC^3, objects only get cached if doing so contributes to the global system benefit by: *i)* reducing the combined transport-storage cost, or *ii)* reducing the average delay.

As illustrated by the structure of the optimal stationary policy, OSC, a caching decision for object k at node u at time t is essentially a trade-off between the cost incurred in writing and keeping object k in the cache of node u, and the cost incurred in fetching k from the closest upstream node that has already cached k. We remark that while the cost of writing and keeping an object at a network nodes is pure storage resource cost, the cost of fetching an object from the closest upstream copy captures both transport resource cost and QoS, since the further the closest copy is, the higher is the delay in delivering the object to the requesting user.

Based on this observation, we can evaluate at time t, the benefit of caching object k at node u, as the difference between the average transport cost needed to transfer object k to node u based on the current network conditions, and the storage cost involved in writing and keeping k at u, as:

$$\mathrm{CB}_{uk}(t) = e^{tr}_{uk}(t)\hat{f}_{uk}(t) - e^{st}_u \tag{27}$$

$$e^{tr}_{uk}(t) = \sum_{(u,v)\in\Gamma_{uk}(t)} e^{tr}_{uv} \tag{28}$$

In (27), (28), $\hat{f}_{uk}(t)$ represents an estimate of the aggregate rate of requests for object k at node u; $e^{tr}_{uk}(t)$ is the transport cost paid at time t to transfer object k to node u from its closest upstream copy along the path $\Gamma_{uk}(t)$; and e^{st}_u represents the cost needed to write and store an object over a time unit in the cache of node u.

Note that in the case of homogeneous resources, i.e., $e^{tr}_{uv} = e^{tr}, \forall(u,v) \in \mathcal{E}$, (27) reduces to

$$\mathrm{CB}_{uk}(t) = h_{uk}(t)\hat{f}_{uk}(t)e^{tr} - e^{st}_u, \tag{29}$$

where $h_{uk}(t)$ is the number of hops to the closest upstream node caching k at time t.

In general, in MC^3, e^{tr}_{uv} represents a generic cost of transporting an object over a link, which may include transport equipment CAPEX and OPEX, as well as link delay. We remark that in the case that the link delay model is load-dependent, $e^{tr}_{uv}(t)$ would be a function of t, indicating the dependence on the current load.

In MC^3, each vCache node maintains a data structure named *shadow cache*, where key metadata related to both cached and not-cached objects is stored in order to compute the global benefit of caching an object at any given time. Each entry in the shadow cache contains the following information:

1) Object Identifier
2) Estimated request inter-arrival time, $\widehat{\Delta t_{uk}}(t) = 1/\hat{f}_{uk}(t)$
3) Estimated transport cost to closest replica, $e^{tr}_{uk}(t)$

Note that the algorithm is based on two main estimates: *i*) the cost of fetching k from the closest copy at the time of the next request arrival, $e^{tr}_{uk}(t)$, and *ii*) the request inter-arrival time of object k, $\widehat{\Delta t_{uk}}(t)$. In order to locally estimate *i*), we propose to store an additional field inside the packets carrying the objects through the network: field E indicates the transport cost incurred by an object as it travels through the network since the last time it was cached. When an object arrives at a cache node, field E is increased to take into account the cost of transferring the object across the last traversed link. The obtained value is then used as $e^{tr}_{uk}(t)$ in Eq. (27) in order to compute the global benefit of caching the object. If the node decides to cache the object or the object has been already cached, field E is reset to 0. This approach allows nodes to share the information they need to compute (29) with negligible constant-size communication overhead. In order to locally estimate *ii*), every time node u receives a new request for object k, it updates the estimated request inter-arrival time in the shadow cache, $\widehat{\Delta t_{uk}}(t)$, based on a predictor. A simple approach is to adopt a moving average computed based on past request arrivals with a suitable window size, as used in LFU and its variants [15].

Algorithm 1 : MC^3

```
For every node u ∈ V, v = Φ_k(u)
if Request for object k at node u (time t) then
    if Object k in the cache then
        Forward k downstream (set E = 0)
        Update Δ̂t_uk(t) in shadow cache entry
        Compute CB_uk(t)
        if (CB_uk(t) > 0) then
            Keep k in the cache and update its position based on
            CB_uk(t) (decreasing order)
        else
            Remove k from the cache
        end if
    else
        Update Δ̂t_uk(t)
        Forward request upstream
    end if
end if
if Object k at node u from v = Φ_k(u) (time t) then
    Get E from packet
    Update e^tr_uk(t) = E + e^tr_vu
    Get Δ̂t_uk(t) from the shadow cache entry
    Recompute CB_uk(t)
    if (CB_uk(t) > 0) then
        Cache k based on CB_uk (decreasing order)
        Set E = 0
        if Cache full then
            Remove last object (least cost benefit)
        end if
    else
        Set E = e^tr_uk(t)
    end if
    Forward k downstream (including E)
end if
```

By relying on the shadow cache and the information carried by the objects travelling through the network, each node is able to identify the subset of objects with the highest cost benefit. This result is achieved by maintaining a list of object entries sorted in terms of decreasing cost benefit. Objects are added and removed to this list every time their cost benefit is recomputed. Note that only objects with a positive cost benefit are potentially cached. This implies that depending on the ratio between transport and storage cost as well as the characteristics of the stream of object requests, nodes will make use of different portions of the available virtual cache space. Finally, we use a negative cost benefit for those objects for which we do not have information inside the shadow cache since we do not have enough information to compute the request inter-arrival time. The object is potentially cached only starting from the second received request.

In order to mitigate the impact of possible overestimates (too short) of request inter-arrival times, a timer is used to update the entries in the shadow cache if no request arrives within a guard time (set as a factor of the estimated inter-arrival time). This approach is useful to correct inaccurate or stale metadata such as the estimated next request arrival time. Note that underestimates of request inter-arrival times are naturally updated when the actual request arrives.

The pseudo-code in **Algorithm** 1 describes the procedures invoked by MC^3 upon *i*) a new request arrival, and *ii*) a new object arrival to a vCache node. Note that MC^3 exhibits constant-time computational complexity and constant-size communication overhead, as neither the number of computations nor the information objects carry grow with the number of nodes and objects in the system. Indeed, in MC^3, objects themselves carry how much cost they incur as they travel through the network. This allows vCache nodes to adaptively learn relevant system information, effectively creating cache cooperation with minimal overhead.

V. Experimental Analysis

We analyze the benefit of MC^3 in the context a 2-layer vCache hierarchy with Internet video workloads that exhibit different content types, daily viewing patterns, and object popularity. The main parameter settings are derived from the work in [18], as described in the following.

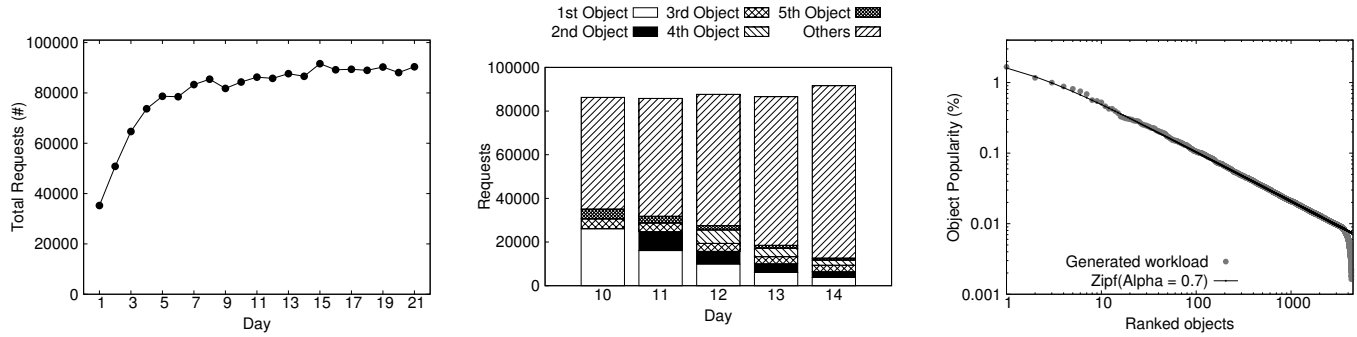
A. Simulation Methodology

Adopted topology. We consider a 2-layer tree structure of vCache nodes: three leaf vCache nodes are connected to a root vCache connected to the *library*, which stores all available content objects. User requests are first forwarded to the leaves in the hierarchy. A request is then forwarded to the root node or to the library only in case of a cache miss. The links between users and leaf vCache nodes are characterized by a delay of 20 ms, while all other links experience a delay of 50 ms. As in a number of previous works (e.g., [18], [19], [20]), we test the vCache hierachy against synthetic yet realistic streams of user requests for video objects.

Object types. We consider two types of video objects: *TV shows* and *Movies*. They differ in terms of size (Movies are typically twice as long as TV shows), and popularity trends (TV shows become unpopular much faster than Movies). The number of TV shows is typically much larger than the number of Movies: in our library, we adopt a shows-to-movies ratio of 4:1, as also suggested in [18].

Object requests. Video object requests are generated according to a Poisson process with average rate determined by the total number of requests to be generated during a specific portion of the day, as described in the following. In our experimentation, we generate $80,000$ requests every day, on average.

Daily pattern. The temporal evolution of the video object requests is known to show a clear time-of-the-day effect [19], with a peak during evening prime time and a lull during the night. To take into account this pattern, we partition the day into four time intervals: morning [6 a.m., 12 p.m.), afternoon [12 p.m., 6 p.m.), evening [6 p.m., 12 a.m.), and night [12 a.m., 6 a.m.). Letting R denote the total number of requests to generate during the day, we inject 10%, 20%, 30%, and 40% of the R requests during the night, morning, afternoon, and evening, respectively.



(a) Overall requests per day. (b) Requests per object. (c) Object popularity after 14 days.

Fig. 2: Details on the workload used in the simulation.

Object popularity. The stream of requests is generated as a series of independent trials drawn from a Zipf (or Zipf-like) distribution over the set of possible objects [15]. However, generating object requests according to a Zipf distribution does not capture the temporal evolution of the popularity of each object. Recently, Balachandran et al [19] observed a specific temporal trend in case of video objects. There is a peak of requests the first day the object becomes available, while the number of requests decreases exponentially during the following days. Hence, while the popularity of video objects over the entire observation period follows a Zipf distribution, a realistic workload must take into account this day-by-day temporal evolution of the object popularity. To achieve this effect, we follow the steps recently proposed by Akhtar et al. [18]: for each object k, *i*) we compute the total number of requests R_k over the entire observation period according to the object popularity extracted from the Zipf distribution; *ii*) given an observation period of N days, we randomly select the day X in which object k becomes publicly available: the R_k requests are then packed in the time range $[X, N]$; *iii*) to determine the first burst and the successive exponential decrease in the number of requests, we adopt a power series expansion such that the final number of requests generated over the time range $[X, N]$ is R_k. New objects are injected into the library at the beginning of the injection day causing the library to grow in size day by day. In our simulation, we inject 300 new video objects every day. Moreover, since empirical observations show that the popularity of TV shows decreases faster than that of Movies [19], we set the day-by-day popularity decrease rate to vary in $[0.3, 0.5]$ for TV shows and in $[0.05, 0.2]$ for Movies.

Fig. 2 shows an instance of the workload used in our simulations. We generate an average of $80,000$ requests per day, although we reach this value after a transitory period of 7 days (see Fig. 2a). Fig. 2b, provides a breakdown of the requests generated between the $10th$ and the $14th$ day. Considering the 5 most popular objects in this time range, we can notice that: *i*) the first, third, and fifth most popular objects are already available on the $10th$ day, whereas the second and forth objects become available only starting from the $11th$ and $12th$ day, respectively; *ii*) the number of requests for each of these objects decreases rapidly day after day. Despite this dynamic evolution in the number of requests, the object popularity computed over the first 14 days is very close to the expected Zipf distribution with Zipf parameter $alpha$ set to 0.7 (see Fig. 2c).

Observation period. We consider a long observation period of 21 days, *i.e.,* 3 weeks. At the end of this period, our library contains 6300 video objects.

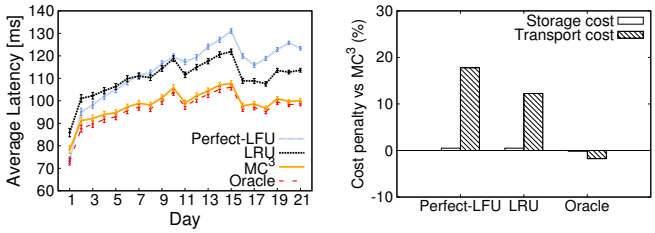
MC^3 settings. We adopt a moving average approach to estimate the object request inter-arrival and, after a first tuning phase, we set this weight to 0.5. In our experiments, we observe a negligible impact of slight modifications to this value on the overall vCDN performance.

Other caching strategies. We compare MC^3 with other caching strategies: *i)* LRU-LCE (*Least Recently Used - Leave Copy Everywhere*) – each cache node applies a least recently used replacement policy; *ii)* Perfect-LFU (*Perfect Least Frequently Used*) – each cache node applies a least frequently used replacement policy that tracks the number of requests for all objects in a shadow cache. This solution is known to well approximate the optimal hit-rate in case of static object popularity; *iii)* Oracle – each cache node can take omniscient caching decisions since nodes are informed about the future object popularity of each day. LRU and LFU represent simple and effective caching strategies made available in commercial products such as Apache Traffic Server, Squid, and Varnish, solutions widely adopted in operational environments.

We remark that the cache management cost of MC^3, while slightly higher than LRU – the lightest caching policy – is exactly the same as that of LFU or any other policy that is ordering-based (i.e., objects are kept in a specified order in the cache) and shadow-cache-based (i.e., policies that track non-cached objects' metadata).

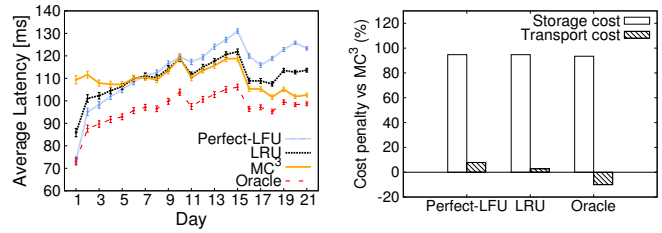
B. Experimental Results

We now describe the performance of MC^3 by varying the transport-to-storage cost ratio, cache size, and object pop-



(a) End to end latency. (b) Operational cost after 21 days.

Fig. 3: Performance when the transport-to-storage cost ratio is 10,000:1.



(a) End to end latency. (b) Operational cost after 21 days.

Fig. 4: Performance when the transport-to-storage cost ratio is 2,000:1.

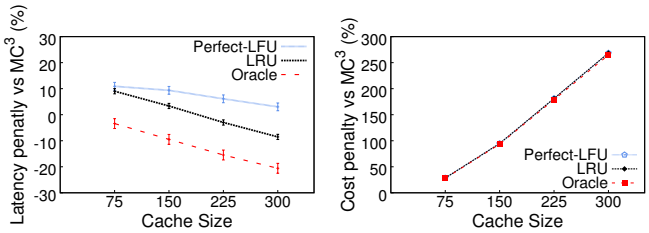
ularity. As done in similar works [22], [21], we relied on Omnet++ [23] to instrument our simulation.

Performance with a varying transport-to-storage cost ratio. We evaluate the caching strategies with a transport-to-storage cost ratio, e^{tr}_{uv}/e^{st}_{u}, of 10,000:1 and 2,000:1. Recall that e^{tr}_{uv} is used to capture not only transport resource costs but also QoS related penalties such as average delay. In this setting, we instrument each node in the hierarchy to cache no more than 50% of the objects injected every day (*i.e.,* 150 objects), while the popularity of each object is generated from a Zipf distribution with Zipf parameter $alpha$ equal to 0.7.

Fig. 3 shows the performance achieved when the ratio is 10,000:1. Fig. 3a shows the daily average latency, *i.e.,* the time the cache hierarchy takes to deliver a requested object to the user. For most of the caching strategies, the performance stabilizes after 7 days. This result is expected since the library is empty at the beginning of the simulation and gets filled day after day with new objects (see Fig. 2a): the equilibrium between previously injected unpopular objects and recently injected highly popular objects is reached only after this initial transitory. Hence, in the following, we discuss the average performance achieved between the $7th$ and $21th$ day. In this setting, the performance of MC^3 is very close to Oracle, the cache strategy that takes omniscient caching decisions. On an average day, the hierarchy instrumented with MC^3 delivers objects to the users with a latency only 1.8% higher than using Oracle, taking 12 ms and 12.4 ms less than LRU and Perfect-LFU, on average. This result is a direct consequence of the higher hit rate achieved by MC^3. Indeed, the average combined hit rate achieved by MC^3 is 28.3% and 33.3% higher compared to LRU and Perfect-LFU, respectively. At the same time, MC^3 guarantees a lower operational cost. Fig. 3b shows the total transport and storage cost penalty paid when using the other caching strategies relative to MC^3. Due to the high transport-to-storage cost ratio, MC^3 uses almost all the available space at the cache nodes in this setting. For this reason, we observe only a limited gain in terms of storage cost. On the other hand, MC^3 carefully selects which objects to cache according to the cost of transferring them over the network, thus achieving a significant gain in terms of total transport cost. Indeed, the transport cost registered when using LRU and Perfect-LFU is 17% and 12% higher than MC^3, respectively. Hence, for high transport-to-storage cost ratio, MC^3 achieves higher performance compared to LRU and Perfect-LFU in terms of hit rate and latency, with a similar storage cost, but a much lower transport cost.

Fig. 4 shows the performance achieved by the tested caching policies in the case of a transport-to-storage cost ratio of 2,000:1. Note that MC^3 is the only strategy modifying its behavior: in this setting, the latency achieved by MC^3 is similar to that achieved by LRU, and lower than that of Perfect-LFU. At the same time, MC^3 achieves this result by using only a fraction of the cache space available at each node: while all other strategies fully use the entire available storage space, MC^3 uses on average only 47% of the space in each cache node. The direct consequence is a significantly lower total operational cost as reported in Fig. 4b: using LRU (Perfect-LFU) leads to a total storage cost of 92% (93%) higher, and a total transport cost 8% (3%) higher than when using MC^3. In conclusion, for less unbalanced ratio between transport and storage cost, MC^3 is able to provide similar or higher performance than LRU and Perfect-LFU in terms of hit rate and latency, with a much lower storage and transport cost.

Performance with a varying cache size. Fig. 5 reports the results achieved when varying the cache size. Each node is configured to cache up to 25%, 50%, 75% and 100% of the amount of objects injected every day, *i.e.,* 75, 150, 225, and 300 objects, respectively. Fig. 5a shows the latency penalty paid when using all other caching strategies relative to MC^3: for larger cache sizes, the latency gain of MC^3 decreases. This happens because the other caching strategies fully use the available storage space in the cache disregarding the associated operational cost, while MC^3 keeps caching objects according to the transport-to-storage cost ratio (2,000:1 in this setting). Fig. 5b shows the cost penalty of operating the cache hierarchy with these strategies compared to using MC^3. The total operational cost increases sharply with the cache size: compared to all the other strategies, MC^3 guarantees savings that go from 28% up to 264% with the increasing cache size. Finally, by being aware of the transport and storage relative costs, MC^3 is able to select the objects to cache in order to guarantee a reasonable average latency while significantly saving in the overall operational cost. Note that one may easily



(a) Average end to end latency. (b) Total operational cost.

Fig. 5: Performance compared to MC^3 when the transport-to-storage cost ratio is 2,000:1 for different cache sizes.

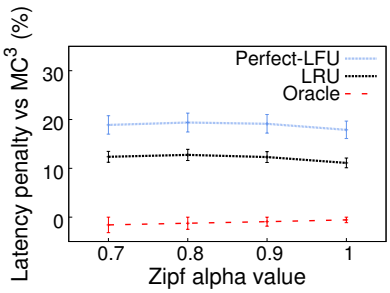


Fig. 6: Latency penalty compared to MC^3 when the transport-to-storage cost ratio is 10,000:1 for different object popularity.

improve the latency performance of MC^3 by simply increasing the transport-to-storage cost ratio to induce the MC^3 nodes to cache more objects, leading to higher hit rate, lower latency, but at the expense of higher operational cost.

Performance with varying popularity. Finally, we also investigate whether and how the performance of MC^3 changes when varying the object popularity. Results are reported in Fig. 6. We consider a cache size of 150 objects and a transport-to-storage cost ratio of 10,000:1. In this setting, the MC^3 nodes are induced to fully use their available storage space. We vary the Zipf parameter $alpha$ in the range $[0.7, 1.0]$. On average, we observe an almost constant gain over LRU and Perfect-LFU, with a latency reduction of 12% and 18%, respectively. Note that the latency achieved by Oracle is only very slightly lower than the one achieved by MC^3.

VI. Conclusions

Motivated by the dynamics and heterogeneity of next generation cloud-based CDNs, and the crushing burden that content storage and transport costs pose on cloud network operators, in this paper we took a fresh look at the dynamic content distribution problem from an overall cost-oriented perspective. We proposed a novel fully distributed online caching solution, we called MC^3, aiming at guaranteeing QoS requirements with minimum overall use of the shared cloud network's infrastructure. We first analytically characterized the optimal cloud caching policy for a given first-order stationary input process, and then – inspired by the structure of the optimal stationary solution – we developed MC^3, an online caching policy that guides local caching decisions based on real-time estimates of the global cost benefit. We implemented MC^3 in a custom-built CDN simulator and presented performance results for different settings of the transport-to-storage cost ratio, cache size, and object popularity. We also provided a comparison with three well known caching strategies (LRU-LCE, Perfect-LFU, and an Oracle), demonstrating the significant performance and efficiency gains – in terms of average latency and overall operational cost – that MC^3 can provide in virtual CDN environments.

References

[1] H. Che, Y. Tung, Z. Wang, "Hierarchical Web caching systems: modeling, design and experimental results," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 7, pp. 1305–1314, 2002.

[2] Bell Labs Strategic White Paper, "The Programmable Cloud Network - A Primer on SDN and NFV," June 2013.

[3] Marcus Weldon, "The Future X Network," *CRC Press*, October 2015.

[4] J. Llorca, C. Sterle, A. M. Tulino, N. Choi, A. Sforza, A. E. Amideo, "Joint Content-Resource Allocation in Software Defined Virtual CDNs," *IEEE ICC'15 CCSNA Workshop*, London, England, 2015.

[5] J. Llorca, A.M. Tulino, "The content distribution problem and its complexity classification," *Bell Labs technical report*, 2013.

[6] S. Hasan, S. Gorinsky, C. Dovrolis, and R. Sitaraman, "Trade-offs in Optimizing the Cache Deployments of CDNs", *IEEE INFOCOM'14*, pp. 460-468, 2014.

[7] I.D. Baev, R. Rajaraman, C. Swamy, "Approximation algorithms for data placement in arbitrary networks," *ACM SODA'01*, 2001.

[8] I.D. Baev, R. Rajaraman, C. Swamy, "Approximation algorithms for data placement problems," *SIAM Journal on Computing*, vol. 38, pp. 1411-1429, 2008.

[9] S. Borst, V. Gupta, A. Walid, "Distributed Caching Algorithms for Content Distribution Networks," *IEEE INFOCOM'10*, San Diego, 2010.

[10] P. Krishnan, D. Raz, Y. Shavitt, "The cache location problem," *IEEE/ACM Trans. on Networking*, vol. 8, no. 5, pp. 568–582, 2000.

[11] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of web server replicas," *IEEE INFOCOM'01*, vol. 3, 2001.

[12] M.R. Korupolu and M. Dahlin, "Coordinated placement and replacement for large-scale distributed caches," *IEEE Transactions on Knowledge and Data Engineering*, vol.14, pp. 1317–1329, 2002.

[13] Wang, J., "A survey of web caching schemes for the internet," *ACM SIGCOMM CCR*, v. 29, n. 5, pp. 36-46, '99.

[14] P. Cao, S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Usenix symposium on internet technologies and systems*, vol. 12, no. 97, pp. 193–206, 1997.

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," *IEEE INFOCOM'99*, vol. 1, pp. 126–134, 1999.

[16] G. Carofiglio, M. Gallo, L. Muscariello, D. Perino, "Modeling data transfer in content-centric networking," *IEEE ITC'11*, pp.111–118, 2011.

[17] E. J. Rosensweig, J. Kurose, "A Network Calculus for Cache Networks," *IEEE INFOCOM'13*, pp. 85–89, 2013.

[18] S. Akhtar, A. Beck, I. Rimac, "HiFi: A Hierarchical Filtering Algorithm for Caching of Online Video," *Proc. of the 23rd ACM international conference on Multimedia (MM '15)*. ACM, NY, USA, 421-430.

[19] A. Balachandran, V. Sekar, A. Akella, and S. Seshan. "Analyzing the potential benefits of CDN augmentation strategies for Internet video workloads." *ACM SIGCOMM IMC*, pp. 43-56. 2013.

[20] B. Paul and C. Mark, "Generating representative web workloads for network and server performance evaluation," *Proc. ACM SIGMETRICS*, Madison, USA, 1998.

[21] J. Llorca, A. M. Tulino, K. Guan, J. Esteban, M. Varvello, N. Choi, D. C. Kilper, "Dynamic in-network caching for energy efficient content delivery," *IEEE INFOCOM'13*, Turin, Italy, 2013.

[22] K. Stamos, G. Pallis, A. Vakali, D. Katsaros, A. Sidiropoulos, and Y. Manolopoulos. "CDNsim: A simulation tool for content distribution networks." *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 20, no. 2 (2010): 10.

[23] A. Varga, "The OMNeT++ discrete event simulation system." *ESM 2001*, vol. 9, no. S 185, p. 65. sn, 2001.