A Content-aware Data-plane for Efficient and Scalable Video Delivery

Yoann Desmouceaux*†, Marcel Enguehard*‡, Victor Nguyen*, Pierre Pfister*, Wenqin Shao*‡, Éric Vyncke*
*Cisco Systems †École Polytechnique ‡Telecom ParisTech
{firstname.lastname}@cisco.com

***Abstract*—Internet users consume increasing quantities of video content with higher Quality of Experience (QoE) expectations. Network scalability thus becomes a critical problem for video delivery as traditional Content Delivery Networks (CDN) struggle to cope with the demand. In particular, content-awareness has been touted as a tool for scaling CDNs through clever request and content placement. Building on that insight, we propose a network paradigm that provides application-awareness in the network layer, enabling the offload of CDN decisions to the data-plane. Namely, it uses chunk-level identifiers encoded into IPv6 addresses. These identifiers are used to perform network-layer cache admission by estimating the popularity of requests with a Least-Recently-Used (LRU) filter. Popular requests are then served from the edge cache, while unpopular requests are directly redirected to the origin server, circumventing the HTTP proxy. The parameters of the filter are optimized through analytical modeling and validated via both simulation and experimentation with a testbed featuring real cache servers. It yields improvements in QoE while decreasing the hardware requirements on the edge cache. Specifically, for a typical content distribution, our evaluation shows a 22% increase of the hit rate, a 36% decrease of the chunk download-time, and a 37% decrease of the cache server CPU load.**

I. Introduction

Traffic from Video-on-Demand (VoD) and linear video streaming is projected to amount to 74 TB/s by 2021, thus representing 82% of the Internet traffic [1]. Not only does video consumption increase in terms of consumed hours, but expectations for Quality of Experience (QoE) are also becoming higher: better video quality, better start-up times, fewer re-buffering events, etc. In that regard, Content Delivery Networks (CDN) are the most common tool for scaling the network while providing better QoE [1]. However, the sheer scale of video traffic raises stringent engineering challenges [2]. Amongst those challenges, optimizing the use of resources (network, storage, and compute) is probably the most crucial. Indeed, as the load on edge caches increases, simply scaling up by using more machines is not sufficient to meet QoE requirements.

Thus, researchers have focused on addressing the challenge of serving more requests with better QoE while using fewer resources. In particular, seminal work argued for using content-awareness to perform traffic engineering (TE) [3]. The authors proposed to transfer the responsibility for TE from the Internet Service Provider (ISP) to the CDN, using ISP network monitoring information to influence server selection through the CDN DNS system. The TE is limited to server selection since the ISP network has no application-knowledge. Similarly, [4] proposes to build an overlay routing graph for video chunks between edge caches. The routing plane is used to redirect incoming requests to a server that has the corresponding chunk and to make local caching decisions. The reliance on an overlay routing plane at the application layer raises scalability issue as HTTP(S) proxies are known to decrease QoE [5]. These two pieces of work are limited by the lack of integration between the network and application layers, which raises capacity and scalability problems. Information-Centric Networking (ICN) [6], [7] tackles that specific issue, using content identifiers instead of network locators to perform hop-by-hop forwarding. The use of content identifiers as network addresses provides application-knowledge to the data-plane and has fostered research in joint-optimization of request forwarding and content-placement in cache networks [8], [9], [10], [11]. However, ICN architectures require fundamental changes to the way current networks are built and their deployment would require considerable efforts.

In this paper, we introduce a novel network design for video CDNs that uses standardized and deployed network technologies to bring application-knowledge in the network layer. The proposed approach relies on two main building blocks: (i) *chunk-level content addressing* and (ii) *in-network server selection*. First, *chunk-level content addressing* consists in assigning a unique and globally routable IPv6 address to each video chunk. Exposing the chunk and video identifiers into the IPv6 addresses space provides visibility about the requested content into the network layer. Second, *in-network server selection* takes advantage of the identifiers exposed as IP addresses to make in-band request forwarding decisions. We build on 6LB [12], a load-balancing framework which uses IPv6 Segment Routing (SRv6) [13] to steer client requests through a chain of *candidate* servers. These servers make local decisions on whether to serve the queries or forward them to the next server in the chain. In [12], 6LB is used to blindly steer requests between idempotent servers for load-balancing purposes. Our work differs in two major ways: first, CDN architectures are essentially vertical and requests must be forwarded first to an edge proxy and then (if the edge proxy

This work benefited from the support of NewNet@Paris, Cisco's Chair Networks for the Future at Telecom ParisTech (https://newnet.telecom-paristech.fr). The authors are grateful towards Thomas H. Clausen for his detailed comments which helped improve this manuscript.

refuses to serve the request) to an origin server; second, we can use the video chunk identifiers encoded in the IPv6 addresses to decide without needing to terminate HTTP sessions whether to serve the query at a given proxy. We thus enrich 6LB with content-awareness. Since our approach relies on standardized technologies, it is *deployable* in today's Internet.

In particular, upon arrival of a request at an edge proxy, the network-level chunk identifier is used to predict whether the chunk might be available in the cache. If not, 6LB is used to forward requests directly to the origin instead of proxying them at the edge cache, thus reducing the load on the edge cache and avoiding the negative effects on QoE. To predict the presence in the cache, we build on prior work by using a Least-Recently-Used (LRU) filter [14], which can be used to probabilistically estimate the popularity of a given chunk. Compared to [14], which optimizes offload costs for Fog applications under latency constraints, the decisive metric for video CDNs is the hit rate of the edge cache. We thus construct an analytical model to evaluate it and provide guidelines for tuning the LRU filter accordingly. 6LB then acts as a concrete and deployable framework for implementing the LRU filter.

The contributions of this paper are summarized below:

- **Video addressing scheme:** We propose an encoding for video/content ID into the IPv6 network address space to facilitate fine-grained analytics and in-network decision-making. This encoding allows corresponding queries to be routed over the global Internet.
- **In-network server selection:** We introduce a decentralized request placement technique that uses an LRU filter to decide in the network layer whether to accept requests at an edge proxy. Upon rejection, 6LB is used to forward the request to the origin, avoiding the proxying step. Using an analytical model validated through simulation, we provide settings to achieve the optimal hit rate without prior knowledge of the request pattern.
- **Realistic environment evaluation:** We evaluate our approach by implementing it in an open-source software router [15] and using a standard unmodified HTTP cache server [16]. The setup shows substantial improvements in terms of cache hit rate (+20%), video chunk download time (-36%), and edge cache CPU load (-37%).

II. A CONTENT-AWARE DATA PLANE FOR VIDEO DELIVERY

Multi-tiered video CDN architectures, as illustrated in Figure 1, consist of three main components [17]: (i) clients who request and consume video chunks, (ii) origin servers that serve content, and (iii) edge caches, located closer to the clients, *e.g.,* in an ISP network, which store the most popular video chunks to reduce the load on the origin servers. Achieving high hit rates in edge caches is essential to the scaling of CDN architectures, as this decreases the load on the origin servers. Moreover, the hit rate on edge caches has a strong impact on QoE factors, such as chunk download time: [5] reports that cache misses increase server latency by up to an order of magnitude, which in turn translate into increased client start-up times. According to [5], this degradation of server latency is notably due to the cost of proxying HTTP(S) connections to the origin server. However, as the load on edge caches increases, simply using more powerful machines and/or adding machines raises stringent economical issues. The approach introduced in this paper thus aims at not only increasing the hit rate at the edge but also reducing the impact of the cache misses. To that end, we rely on two main components: network-layer video chunk naming (Section II-A) and 6LB-based server selection (Section II-B).

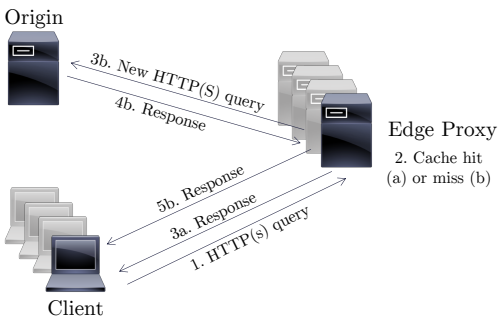

Fig. 1. Proxy-based CDN architecture: clients issue HTTPS queries to an edge proxy that either replies back upon cache hit, or terminates the session and fetches the content from an origin server upon cache miss.

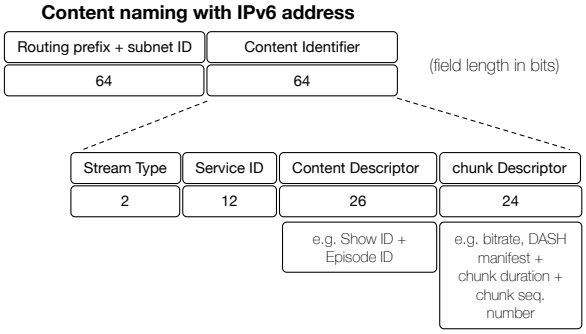

Fig. 2. Naming scheme: IDs and metadata encoded within IPv6 addresses.

A. Naming Scheme

The fundamental characteristic of our architecture is the use of named-video chunks (*e.g.,* DASH [18] or HLS segments) in the forwarding plane. Our proposal borrows concepts from Information-Centric Networking (ICN) [6], [7] (without offering all ICN features, *e.g.,* native multicast) while aiming at deployability in current IP-based networks. As in ICN, we match each video segment with a unique network identifier: a 64-bits encoding is used, as described in Figure 2. It contains the video identifier, the identifier of the segment within the video, and potentially additional metadata such as the segment duration and the requested video bitrate/quality. We then build an IPv6 address from this name: (i) the first 64 bits are a prefix that is specific to the video producer and acts as a network locator; (ii) the 64-bits suffix contains the aforementioned video metadata and acts as a content identifier. On our website,

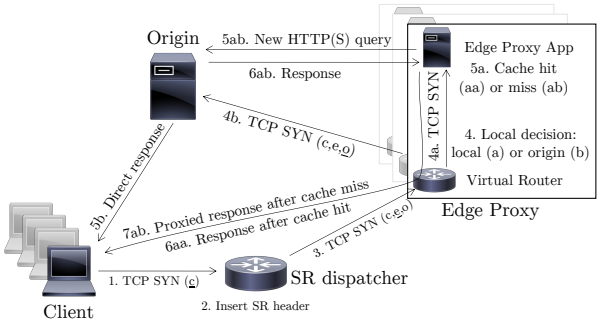

Fig. 3. 6LB-based CDN architecture: clients issue HTTPS queries to a dispatcher, which adds an SR header with the address of an edge and an origin. The data-plane at the edge decides to dispatch the query to the origin (without proxying), or locally to the proxy server.

we provide a tool to easily convert video chunk names to IPv6 addresses and vice-versa[1]. Our approach thus uses globally routable IPv6 packets while providing visibility of the content identifier within the forwarding plane. In addition, exposing these video metadata into the IP addressing space enables fine-grained analytics (*e.g.,* eyeball time) to be gathered by standard network flow analyzers. The rest of the network stack uses traditional internet protocols: TCP/TLS/HTTP.

B. 6LB-based Server Selection

Traditional CDN architectures (as depicted in Figure 1) make use of layer-7 proxies and DNS resolution [19], [17] to provide caching capabilities. When issuing an HTTP request for a piece of content, clients are directed to the geographically closest edge cache (1), which terminates the HTTP connection and replies with a cached copy of the content if available (2a) or opens another HTTP connection to an origin server (2b-3b) and replies to the client with the newly fetched content (4b) while possibly caching it (see Figure 1). A drawback of this architecture lies in the performance cost of terminating the client HTTP connection in case of a cache miss at the edge cache, especially when using TLS [20]. Furthermore, TCP/IP stacks provided by the native Linux kernel (often used in popular HTTP caching servers) are known to be inefficient due to context switches and the lack of batching [21], [22]. Given these limitations, when it is not beneficial to cache the content at the edge (*e.g.,* because it is unpopular), a more efficient approach would be to bypass the edge and open a direct connection between the origin server and the client rather than resorting to HTTP proxying.

To achieve in-network server selection, this paper builds on 6LB [12]. 6LB leverages SRv6 [23], a networking architecture standardized in RFC 8402 [13], which allows packets to traverse a source-specified sequence of "segments" represented by IPv6 addresses. SRv6 uses an IPv6 extension header [24] and allows traversal through SR-unaware nodes, and as such is deployable in any IPv6 network. Following the principle introduced in [12], 6LB-based server selection is performed as described in Figure 3:

- A *dispatcher* advertises routes towards the (anycast) content prefix and catches traffic addressed to this prefix. The location of the dispatcher is flexible: it can be integrated within the client's network stack, pushed to a set-top box in the client's premises, set in a gateway router at the entry of the Point of Presence (PoP) closest to the client, or co-located with the edge cache.
- Upon receipt of a TCP SYN, the dispatcher inserts an SRv6 header in the packet containing the list (e, o, c), where e is the address of an edge cache capable of serving the content, o is the address of the origin server, and c is the anycast address of the content (the original destination address of the packet).
- When receiving this SYN packet, the data-plane at the edge cache e decides to either (i) *accept* the connection and pass the SYN packet to its local HTTP cache server or (ii) *refuse* the connection and pass the packet to o without going through the local HTTP server.
- Downstream packets from e or o are sent and routed directly to the client.
- Subsequent (non-SYN) packets from the client corresponding to the same connection also reach the dispatcher and an SRv6 header with (e, c) or (o, c) is inserted, depending on whether e or o has accepted the connection[2].

The advantages of this approach are threefold. First, Direct Server Return (DSR) [25], [26] takes place between the server that has accepted the connection and the client: packets from the selected server directly reach the client. The load on the dispatcher is thus greatly reduced. Second, this approach increases QoE by (i) reducing the load on the proxy server because some requests are now directly sent to the origin server and (ii) reducing the response time for these requests by removing the proxying overhead. Third, it offers management benefits as any *local, pluggable* policy can be used at the edge cache to decide whether to accept a connection. For example, decisions can be made depending on the current load of the server, an estimation of the popularity of the content, or a combination of both.

Note that, while the dispatcher could work at the application layer (e.g., through DPI or proxying), its implementation directly in the network layer via the identifiers exposed in the IPv6 address yields a higher throughput (due to DSR and per-packet operation) and thus scalability. Indeed, it means that the dispatcher can be implemented using state-of-the-art software routers (see Section IV-A) or even in hardware.

III. An example of data-plane enabled application: In-network edge admission control

As described in Section II-B, the combination of per-content naming and in-network server selection allows of-

[1]http://demo.6cn.solutions/nf/decode2.php

[2]The dispatcher is notified, in-band, of the identity of the accepting server. This is achieved by steering (with SR) the first downstream packet to the dispatcher, and embedding the address of the accepting server therein, as described in [12].

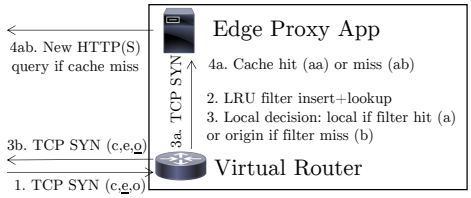

Fig. 4. LRU filter as in data-plane acceptance policy. Queries are either (3a) routed locally to the edge cache or (3b) routed to the origin. In case (3a), the edge cache has to proxy the connection in case of edge cache miss.

floading certain client requests transparently to an upstream server through in-data-plane decisions. This section introduces an in-data-plane popularity-aware edge admission policy that takes advantage of this capability. This policy decides which requests are better served at the edge cache or at the origin server by continuously adapting to the client request pattern – thus increasing the hit rate and protecting the edge from proxying unpopular requests.

A. Network-layer Acceptance Policy for Edge Proxies

The cache admission policy presented in this paper consists in *pre-selecting* popular content at the network-layer before handing it to a (black-box) edge cache. In particular, we take advantage of the naming scheme described in Section II-A to profile the popularity of content within the data-plane. An efficient admission module should indeed aim at accepting only requests for popular content at the edge, thus increasing the hit-rate. Conversely, unpopular content (that should not be cached) can be directed to the origin server without going through the edge proxy. Based on [14], [8], we build an in-data-plane filter admission policy that uses an LRU meta-cache to decide whether requests should be handled at the edge or forwarded upstream: this is called an *LRU filter* [14].

This LRU filtering module has an *identifier cache* $\mathcal{C}$ of size C_1, which is an LRU meta-cache storing the *identifiers* (*i.e.,* content addresses) of the lastly-requested video chunks. Upon arrival of a TCP SYN packet with an SRv6 header (e, o, c) for content c at the LRU filter of an edge cache:

- If $c \notin \mathcal{C}$, c is deemed unpopular and the packet is forwarded to the origin server o (bypassing the edge e). In addition, c is added to the head of $\mathcal{C}$ (and the last entry of $\mathcal{C}$ is removed if $\mathcal{C}$ is full).
- If $c \in \mathcal{C}$, this is (at least) the second time that c has been requested since c has entered $\mathcal{C}$ and thus c is deemed popular: the packet is forwarded to the edge e. In addition, c is moved back to the head of $\mathcal{C}$.

Hence, with high probability, unpopular content is not served by the edge cache but rather directly offloaded (at the network layer) to the origin server. The offloaded connections no longer need to be proxied at the edge, thus avoiding unnecessary HTTP terminations and the cache of the edge proxy is not polluted with unpopular content, consequently increasing the hit rate.

B. Optimal sizing of the LRU Filter

The LRU filter has a single tunable parameter, the size of the identifier cache C_1, whose influence is studied in this section. When $C_1 = 0$, all requests are deemed unpopular by the filter and are thus served by the origin server – equivalent to disabling edge caching. Conversely, if C_1 is greater than the total number of objects, all requests are handed to the edge proxy – equivalent to having no filter. The remainder of this section presents an analytical model for determining the size C_1 yielding the best hit rate and quantifies the benefits versus having no filter.

To make the model tractable, we make the assumption that the *object* cache in the HTTP edge proxy uses an LRU caching policy. Since it is desirable to maximize the use of the edge cache while minimizing the number of proxied requests, the metric that we optimize is the *global hit rate* at the edge cache. This is to be understood as the ratio of queries that are served by the cache without proxying – in other words, those that result in a hit at the LRU filter followed by a hit at the edge cache.

Let us fix notation for the remainder of the paper. We consider a catalog of size N, in which objects have a Zipf popularity of parameter $\alpha > 0$. Denoting by $q(n)$ the (unnormalized) popularity of the n-th most popular object, this means that $q(n) = 1/n^\alpha$. We assume that the filter has size $C_1 = \delta_1 N$, and that the edge cache has size $C_2 = \delta_2 N$, where $\delta_1, \delta_2 \in (0, 1)$ are (fixed) cache/catalog size ratios. We assume large catalogs, as is the case in real-life CDNs, and therefore make the assumption that $N \to \infty$.

The aim of the optimization is to, given a cache size δ_2, find the filter size δ_1 which maximizes the global hit rate. To that purpose, four steps are taken. First, we derive an expression of the hit rate and the *characteristic time*[3] of both the LRU filter and the edge cache. Second, we approximate these characteristic times, assuming $N \to \infty$. Third, with the help of these approximations, the global hit rate of the system can be estimated for $N \to \infty$. Finally, having expressed the global hit rate as a function of δ_1, δ_2, we can derive the optimal filter size δ_1, with respect to the cache size δ_2.

1) Deriving the Hit Rates and Characteristic Times: According to Che's approximation [27], the hit rate of the n-th content at the filter can be estimated as:

$$h_1(n) = 1 - e^{-q(n)t_1} \tag{1}$$

where t_1 (the *characteristic time* of the filter) is the unique solution to:

$$C_1 = \sum_{n=1}^{N} (1 - e^{-t_1 \cdot q(n)}) \Leftrightarrow \delta_1 = \frac{1}{N} \sum_{n=1}^{N} (1 - e^{-t_1/n^\alpha}) \tag{2}$$

With respect to the edge cache, the filter acts as a pre-processing that un-skews the popularity distribution of the

[3]The characteristic time is a metric qualifying the behaviour of a caching system that proves useful to estimate the hit rate of each individual content, according to [27].

objects. The popularity as seen by the edge cache is then:

$$q_2(n) = q(n)h_1(n) = h_1(n)/n^\alpha \tag{3}$$

Using Che's approximation again, the hit ratio of the n-th content at the edge cache is:

$$h_2(n) = 1 - e^{-q_2(n)t_2} = 1 - e^{-t_2 h_1(n)/n^\alpha} \tag{4}$$

where t_2 (the *characteristic time* of the edge cache) is the unique solution to:

$$C_2 = \sum_{n=1}^{N}(1 - e^{-t_2 \cdot q_2(n)}) \Leftrightarrow \delta_2 = \frac{1}{N}\sum_{n=1}^{N}(1 - e^{-t_2 \cdot h_1(n)/n^\alpha}) \tag{5}$$

2) Estimating the Characteristic Times: When $N \to \infty$, a first-order approximation of the characteristic time of the filter, t_1, can be computed as (according to [28]):

$$t_1 = \psi^{-1}(\delta_1)N^\alpha + \mathcal{O}(N^{\alpha-1}) \tag{6}$$

where $\psi(\beta) \coloneqq \int_0^1 (1 - e^{-\beta/x^\alpha})\mathrm{d}x$ for $\beta \in [0, +\infty)$ (as defined in [28]). In particular, $\psi(\beta)$ is the average size of an LRU cache when the average sojourn time is β.

Using this approximation to compute $q_2(n)$ (the popularity as seen by the edge cache), it is possible to further compute a first-order approximation of the characteristic time of the edge cache, t_2, as:

$$t_2 = \Psi^{-1}_{\psi^{-1}(\delta_1)}(\delta_2)N^\alpha + \mathcal{O}(N^{\alpha-1}) \tag{7}$$

where:

$$\Psi_\gamma(\beta) \coloneqq \int_0^1 \left(1 - \exp[\frac{-\beta}{x^\alpha}(1 - e^{-\gamma/x^\alpha})]\right)\mathrm{d}x$$

for $\beta \in [0, +\infty)$ and a parameter $\gamma > 0$.

Proof of the derivation of (7). Intuitively, the proof consists of replacing $h_1(n)$ with $(1 - e^{-\psi^{-1}(\delta_1)N^\alpha/n^\alpha})$ in equation (5), thanks to equation (1) and equation (6). Then, similarly as in [28], the idea is to replace the (pseudo) Riemann sum $\frac{1}{N}\sum_{n=1}^{N}(1 - \exp[-\frac{t_2/N^\alpha}{(n/N)^\alpha}(1 - e^{-\frac{\psi^{-1}(\delta_1)}{(n/N)^\alpha}})]$ with $\int_0^1(1 - \exp[-\frac{t_2/N^\alpha}{x^\alpha}(1 - e^{-\frac{\psi^{-1}(\delta_1)}{x^\alpha}})]\mathrm{d}x$ in equation (5), leading to $\delta_2 \approx \Psi_{\psi^{-1}(\delta_1)}(\frac{t_2}{N^\alpha})$, and finally $t_2 \approx \Psi^{-1}_{\psi^{-1}(\delta_1)}(\delta_2)N^\alpha$. A rigorous proof of convergence can be found in the appendix of the technical report extending this paper [29]. □

3) Estimating the Hit Probability: The global hit rate H of the system corresponds to requests that generate a hit in the filter followed by a hit in the edge cache:

$$\mathbf{E}[H] = \frac{\sum_{n=1}^{N} q(n)h_1(n)h_2(n)}{\sum_{n=1}^{N} q(n)} \tag{8}$$

Using the asymptotic approximations of t_1 and t_2 computed in Section III-B2, it is possible to derive an approximation of the global hit rate for $N \to \infty$:

$$\mathbf{E}[H] = 1 - \begin{cases} (1-\alpha)I(\delta_1,\delta_2) + \mathcal{O}(\frac{1}{N}) & \alpha < 1 \\ \frac{1}{\log N}I(\delta_1,\delta_2) + \mathcal{O}(\frac{1}{\log^2 N}) & \alpha = 1 \\ \frac{1}{\zeta(\alpha)N^{\alpha-1}}I(\delta_1,\delta_2) + \mathcal{O}(\frac{1}{N^\alpha}) & \alpha > 1 \end{cases} \tag{9}$$

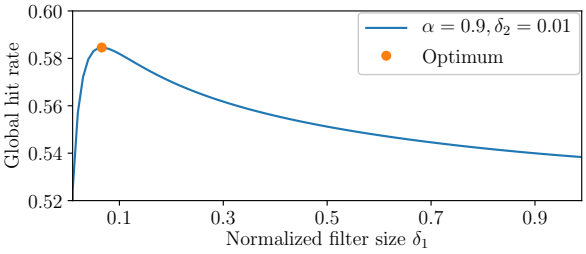

Fig. 5. Global hit rate vs filter size δ_1 for α=0.9, δ_2=0.01.

where $\zeta(\alpha) = \sum_{n=1}^{+\infty} 1/n^\alpha$ is Riemann's zeta function, and $I(\delta_1, \delta_2)$ is defined as:

$$I(\delta_1,\delta_2) = \int_0^1 \frac{1}{x^\alpha}\Big[1 - (1 - e^{-\frac{\psi^{-1}(\delta_1)}{x^\alpha}})\times \left(1 - \exp[-\frac{\Psi^{-1}_{\psi^{-1}(\delta_1)}(\delta_2)}{x^\alpha}(1 - e^{-\frac{\psi^{-1}(\delta_1)}{x^\alpha}})]\right)\Big]\mathrm{d}x$$

Proof of the derivation of (9). The proof uses similar tools to the one for (7), and can be found in the appendix of the technical report extending this paper [29]. □

4) Optimal Sizing of the Filter: The estimation of the hit rate $\mathbf{E}[H]$ computed in Section III-B3 can be used to find the optimal filter size δ_1^* for a given size of the cache δ_2, defined as the size which maximizes the hit rate of the system for $N \to \infty$. (Note that, when $\alpha \geq 1$, the hit rate of the system goes to one as $N \to \infty$, therefore for such cases we define the optimal filter size as the one which minimizes the first-order term in the asymptotic expansion of the miss rate.) Given equation (9), δ_1^* can be found by solving the following optimization problem:

$$\min_{\delta_1 \in (0,1)} I(\delta_1, \delta_2) \tag{10}$$

Figure 5 gives an example of such an optimization: for $\alpha = 0.9$ and $\delta_2 = 0.01$, the hit rate when $N \to \infty$ is depicted as a function of δ_1. It can be observed that, when $\delta_1 \to 0$, the hit rate vanishes, and that when $\delta_1 \to 1$, the hit rate reaches that of a single LRU cache of size δ_2. In between, the hit rate exhibits a bell-curve-like shape, with a maximum attained when $\delta_1 = 0.067$.

C. Numerical Results

Figure 6 depicts a heatmap of the hit rate for $\alpha = 0.9$ and $N \to \infty$, as a function of (δ_1, δ_2). It can be observed that, for a given cache size δ_2, the hit rate evolves rapidly for small values of δ_1, reaches an optimum, and then evolves more slowly for larger values of δ_1, before converging to the hit rate of a single LRU when $\delta_1 = 1$. Due to the rapid evolution of the hit rate when δ_1 is smaller than its optimal value, it is, therefore, preferable to over-estimate δ_1 when sizing the filter.

Figure 7 illustrates the benefits of correctly sizing the LRU filter, by plotting the hit rate obtained by three policies: (i) no LRU filter, *i.e.,* just an LRU cache as edge, (ii) "blind" LRU filtering, where the filter is sized with $\delta_1 = \delta_2$ as in [30],

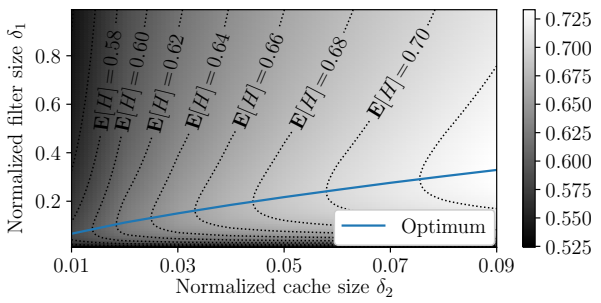

Fig. 6. Sizing the LRU filter: heatmap of *global hit rate* for a cache size $\delta_2 \in [0.01, 0.09]$ and a filter size $\delta_1 \in (0, 1)$, for a Zipf distribution with $\alpha = 0.9$. Black lines are isolines for the global hit rate. Blue lines give the optimal filter size δ_1 as a function of the cache size δ_2.

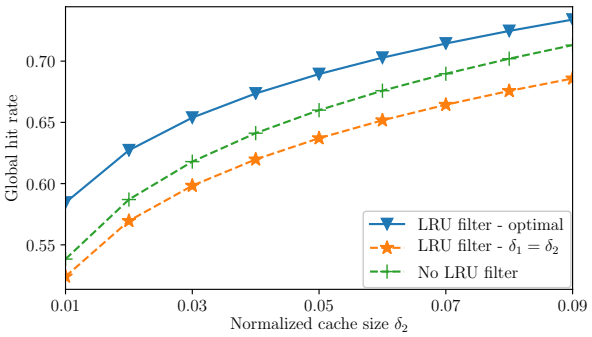

Fig. 7. Comparison the hit rate of three policies: no LRU filtering, "blind" LRU filtering ($\delta_1 = \delta_2$) and "optimal" LRU filtering ($\delta_1 = \delta_1^*(\delta_2)$), for a Zipf distribution with $\alpha = 0.9$.

(iii) "optimal" LRU filtering, where the filter is sized with $\delta_1 = \delta_1^*$. For instance, for $\delta_2 = 1\%$, using the "optimal" LRU filter offers an improvement of 9% over LRU, whereas using a "blind" LRU decreases the performance by 3%.

Finally, in order to understand the behavior of the optimal filter size δ_1^*, Figure 8 represents δ_1^* for caches sizes in the "reasonable" range $\delta_2 \in \{1\%, 2\%, \ldots, 9\%\}$. It can be observed that δ_1^* approximately follows a power law:

$$\delta_1^* \approx B\delta_2^A$$

where the coefficients A and B only depend on α. Table I gives the value of A and B obtained by linear regression in log-log space, with the corresponding R^2 value. The closeness of R^2 to 1 confirms the accuracy of this description.

Guideline: Given the results from Table I, it is possible to provide a simple *guideline formula* to choose a filter size δ_1 as a function of a cache size δ_2, for "reasonable" values of the parameters δ_2 and α. Indeed, the coefficients A obtained in Table I also exhibit a linear dependence on α, and the coefficients B vary in a narrow range for the considered values. Therefore, we can define a guideline δ_1^{*g} as:

$$\delta_1^{*g} = \hat{B}\delta_2^{C\alpha+D} \tag{11}$$

where C, D are obtained by linear regression $A(\alpha) \equiv C\alpha + D$ on the points of Table I, and $\hat{B}$ is the mean of the values of B

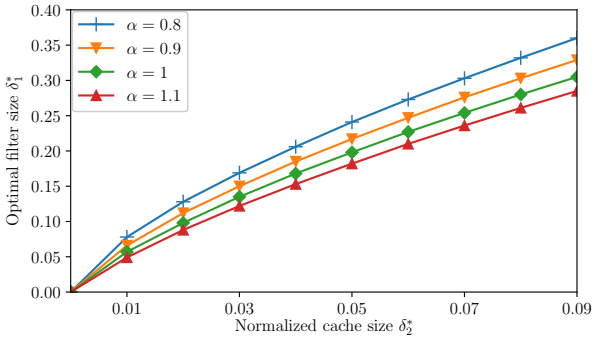

Fig. 8. Optimal size of filter δ_1^* as a function of cache size δ_2 for $\alpha \in [0.8, 1.1]$

TABLE I
FITTING THE OPTIMAL FILTER SIZE δ_1^* AS $\delta_1^* = B\delta_2^A$ WHEN $\delta_2 \in [0.01, 0.09]$

α	A	B	$1 - R^2$
0.75	0.680	1.950	1.45×10^{-5}
0.80	0.694	1.924	7.23×10^{-5}
0.85	0.710	1.911	7.56×10^{-5}
0.90	0.729	1.919	2.23×10^{-4}
0.95	0.748	1.937	2.72×10^{-4}
1.00	0.763	1.936	2.60×10^{-4}
1.05	0.780	1.950	2.82×10^{-4}
1.10	0.799	1.979	4.63×10^{-4}

in Table I. Numerical values of $\hat{B}, C, D$ are given in Table II. Note that (11) should be seen as a convenient helper to size an LRU filter, rather than an explanatory model of the underlying structure.

IV. Experimental Testbed

This section details the implementation of the proposed architecture in a realistic testbed. Note that, for lack of access to commercial solutions and to increase reproducibility , the testbed is built using *state-of-the-art open-source* software.

A. Implementation

Two of the modules in Figure 3 have been implemented as plugins of the high-performance software router VPP [15], and are available in open-source at [31]: (i) the *dispatcher*, which inspects the destination VIP (video chunk ID in IP address) of packets and inserts SR headers accordingly, and (ii) the *server agent*, located in the edge proxy and implementing the LRU filter described in Section III-A. Upon refusal, the server agent forwards packets to the next hop in the SRv6 segment list (the origin server) via its egress interface; upon acceptance, to a local interface bound to a VM containing the actual edge proxy. In particular, the LRU-filter is implemented as an acceptance policy for the 6LB VPP implementation described in [12], using a linked list to store LRU entries and a flow table to map IPv6 addresses to the corresponding entries.

B. Testbed Description

To highlight the effects of in-network server selection, we built a testbed with one dispatcher, one edge proxy and one

TABLE II
REGRESSION OF $A(\alpha)$ AS $A(\alpha) = C\alpha + D$ AND $B(\alpha)$ AS $\hat{B}$

C	D	$1 - R^2$	$\hat{B}$
0.342	0.422	8.36×10^{-4}	1.938

origin server. In real deployments, there would likely be more than one dispatcher and proxy; yet reasoning on a single chain enables fine-grained understanding of the system's behavior.

The testbed consists of four physical machines, equipped with Intel Xeon E5-2667 CPUs (3.20 GHz) and Intel XL710 40 Gbps NICs, connected to a single 40 Gbps fabric, on which the services are virtualized through KVM [32]. The first machine plays the role of a *query generator*, the second of the *dispatcher*, the third of the *edge proxy*, and the last of the *origin server*. The VPP dispatcher and the VPP server agents in the edge proxy and origin server are each pinned to one CPU thread. The edge proxy VM and origin server VM are, each, pinned to 4 CPU cores (8 CPU threads). Locust [33] is used as query generator, *nginx* [34] as origin HTTP server and Apache Traffic Server (ATS) [16] as edge HTTP cache proxy. To model the physical distance between the edge and the origin server, the traffic shaper `tc` [35] is used to add a 40 ms delay for packets egressing the origin server. The origin server is filled with $N = 10^7$ video chunks of size 1.125 MB. The edge cache is equipped with an SSD cache of size 112.5 GB, allowing to store $C_2 = 10^5$ chunks (normalized size $\delta_2 = 0.01$). Finally, the LRU filter in the VPP admission module is set with different sizes throughout the experiment, ranging from $C_1 = 0$ to $C_1 = 10 \times C_2 = 1.0 \cdot 10^6$ (*i.e.,* $\delta_1 \in [0, 0.10]$). The content popularity follows Zipf distributions with different α, representing different content popularity skewness.

Unless specified otherwise, the query generator simulates 1400 clients that simultaneously request video chunks according to the popularity distribution. As a baseline, the same experiments are repeated without in-network filtering – *i.e.,* with requests always assigned to the edge cache.

V. EXPERIMENTAL RESULTS

In this section, we first verify the predictive power of the analytical model presented in Section III. We then verify if and quantify how LRU filtering optimization yields QoE improvements (chunk download time) and operational benefits (CPU consumption reduction).

A. Model Validation

To validate the model of Section III-B, we show in Figure 9 the average hit rate as a function of the normalized filter size δ_1 for $\alpha = 0.8$ and $\alpha = 0.9$ (as in equation (8), the *global* hit rate corresponds to those queries served by the edge cache without proxying). Values corresponding to the theoretical model (•) and a Poisson simulation thereof (▲) are reported alongside the experimental results (×). According to the model, the global hit rate starts at 0 when $C_1 = 0$, then hits an optimum before decreasing to that of a configuration without the filter. Since

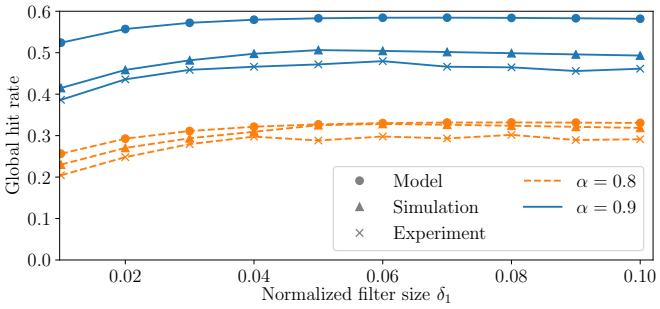

Fig. 9. Global hit rate as a function of LRU filter size for the model, simulation, and experiments for N=10^7, δ_2=0.01, $\alpha \in \{0.8, 0.9\}$.

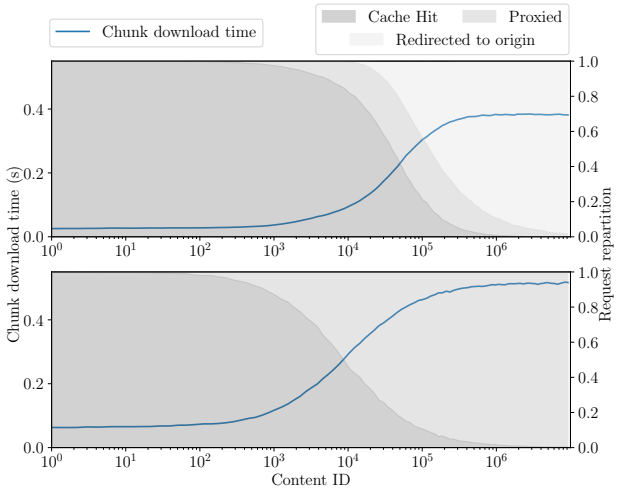

Fig. 10. Average per-content hit rate and response time for N=10^7, α=0.9, δ_2=0.01 (content IDs are sorted by increasing popularity). In-network LRU filter with $\delta_1 = \delta_1^* = 0.06$ (top) vs no filter (bottom).

ATS uses FIFO rather than LRU as cache admission policy and the model assumes an infinite catalogue, the curves do not match perfectly. However, the optimal value of the hit rate is attained for the same value of C_1 (*e.g.,* at $\delta_1 = 0.06$ for the experimental data when $\alpha = 0.9$, while the value predicted by equation (11) yields $\delta_1^* = 0.067$). This indicates that equation (11) provides a good approximation for sizing the LRU filter in real deployments. Finally, in terms of hit rate, when using the optimally-sized LRU filter with $\alpha = 0.9$, the global hit ratio reaches 48%, as compared to 44% when without the filter, thus yielding a 9% improvement. Further validation of the model (for various values of α and δ_1) has been conducted through simulation. The results (not presented here for lack of space) are reported in Figure 5 of [29]. In particular, they illustrate that LRU-filtering is less efficient for smaller values of α (as the distribution is more uniform, popularity variations have less impact [14]) but so are caching systems in general.

B. Quality of Experience

To quantify the QoE benefits of the proposed architecture after LRU filter optimization, download times were recorded

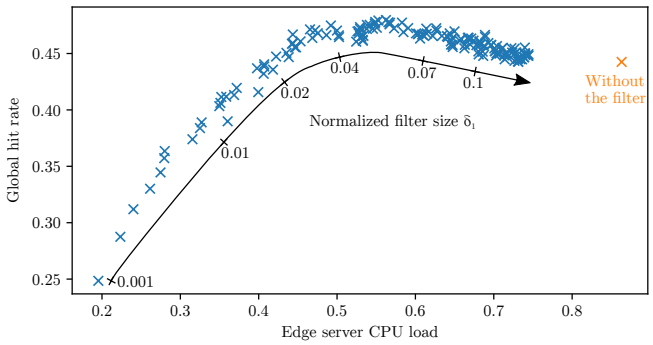

Fig. 11. CPU load versus hit rate for $N = 10^7, \alpha = 0.9, \delta_2 = 0.01$.

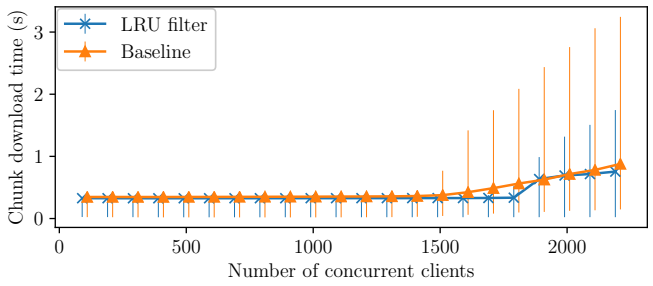

Fig. 12. Median chunk download time versus number of concurrent clients, for $N = 10^7, \alpha = 0.9, \delta_2 = 0.01$. Error bars represent the 10th and 90th percentiles. In-network LRU filter with $\delta_1 = \delta_1^* = 0.06$ vs no filter.

when using the optimal value δ_1^* as filter size, and with $1.5{\cdot}10^6$ queries injected into the system. Figure 10 depicts the average per-content download time and per-content hit rate, for α=0.9. As expected, the LRU filter helps distinguish between popular and unpopular content: without LRU filtering, the cache can serve the ≈300 most popular chunks (amounting to ≈20% of the queries) with a hit rate greater than 95%, whereas LRU filtering provides a 95% hit rate for the ≈2100 most popular video chunks (amounting to ≈30% of the queries). This results in a consistently lower download time with LRU filtering as popular chunks enjoy an improved hit-rate. Particularly, the 300 most popular chunks benefit from a sub-31 ms download time, compared to 88 ms with the baseline. Furthermore, unpopular chunks also enjoy a lower download time thanks to the removal of the proxying step. Requests for these chunks are indeed routed directly to the origin server instead of going through the proxy. More precisely, chunks with IDs $10^6 \rightarrow 10^7$ are redirected to the origin server with probability greater than 90%. To summarize, requests for popular content enjoy a lower download time thanks to increased hit-rate and reduced CPU usage at the cache, while requests for unpopular content (which would have had to be proxied with high probability) benefit from avoiding the proxying cost, as well as from lower network utilization on the upstream link and lower CPU usage on the origin server.

TABLE III
PERFORMANCE FOR DIFFERENT CONTENT POPULARITY DISTRIBUTIONS

α	Hit rate (%)		Chunk load time		Edge CPU load (%)	
	Filter	Baseline	Filter	Baseline	Filter	Baseline
0.8	30.0	21.9	379 ms	755 ms	49.0	83.5
0.9	47.0	38.6	220 ms	345 ms	54.4	85.9
1	65.3	59.5	139 ms	189 ms	56.8	83.9
1.1	81.8	77.8	85.6 ms	107 ms	53.8	74.4

C. Operational Benefits

Apart from client-facing metrics (hit rate, download time), we inspect the server-side performance gain brought by our architecture. Figure 11 depicts the CPU load on the edge cache alongside the global hit rate when the filter size varies (parameters identical to Figure 9, and $\alpha = 0.9$). As the filter size increases, the edge CPU load keeps increasing, even after the hit rate starts decreasing – because the edge has to proxy more and more queries, and becomes polluted by unpopular content. The benefits of having an optimally-sized filter thus become clear: it decreases the CPU footprint as compared to bigger filters (or no filter) while improving the global hit rate. For instance, the CPU load went down from 86% (no filter) to 56% (optimal filter), reducing the CPU footprint by one third.

Finally, the scalability of the architecture is evaluated in Figure 12, which illustrates the impact of the number of simulated users on the chunk download time. We report the median download time (markers) as well as the 10th and 90th percentiles (error-bars). The baseline architecture can sustain 1500 clients before chunk download time starts increasing, whereas our architecture allows up to 1800 clients before a degradation can be observed. In particular, the LRU filter keeps the 90th percentile below 2.3× the median, thus providing improved QoE for most users even under high load.

To sum up these results, Table III reports the global hit rate, average download time and edge CPU load, when using optimal LRU filtering versus no filtering, and for different values of α. It highlights that our architecture brings improvement on those three dimensions: it improves the hit rate by up to 37% while decreasing the average chunk download time by up to 50% and the average CPU load by up to 41% (for $\alpha = 0.8$).

VI. CONCLUSION

This paper details the functioning of a content-aware data-plane for video CDNs. By addressing named video-chunks in the network layer, the forwarding plane becomes application-aware. Decisions can then be made at the network layer, removing the need from terminating HTTP sessions and thus decreasing CPU costs and increasing scalability. For instance, the performance of edge caches can be improved through network-layer popularity estimation using an LRU filter and traffic redirection using 6LB. We provide an analytical model for the performance of such a filter and guidelines for its optimization. An implementation of the LRU filter on state-of-the-art virtual routing software shows significant improvements in terms of user QoE (−36% of chunk download time) and of resource utilization at the edge (−37% of CPU utilization).

REFERENCES

[1] Cisco Virtual Networking Index. "The zettabyte era: Trends and analysis.", 2017. URL https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf. [Accessed 2018/06/12].

[2] A. Balachandran, et al. "Developing a predictive model of quality of experience for internet video." *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13*, p. 339, 2013.

[3] I. Poese, et al. "Enabling content-aware traffic engineering." *Computer Communication Review*, vol. 42, no. 5, pp. 22–28, 2012.

[4] H. Huang, et al. "Joint optimization of content replication and server selection for video-on-demand." In *Communications (ICC), 2012 IEEE International Conference on*, IEEE, pp. 2065–2069, 2012.

[5] M. Ghasemi, et al. "Performance characterization of a commercial video streaming service." In *Proceedings of the 2016 ACM on Internet Measurement Conference - IMC '16*, pp. 499–511, 2016.

[6] L. Zhang, et al. "Named data networking." *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[7] The Linux Foundation. "FD.io CICN project.", 2017. URL https://wiki.fd.io/view/Cicn.

[8] G. Carofiglio, et al. "FOCAL: Forwarding and caching with latency awareness in information-centric networking." In *Globecom Workshops (GC Wkshps), 2015 IEEE*, IEEE, pp. 1–7, 2015.

[9] A. Chanda, et al. "Content based traffic engineering in software defined information centric networks." In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, IEEE, pp. 357–362, 2013.

[10] E. Yeh, et al. "Vip: A framework for joint dynamic forwarding and caching in named data networks." In *Proceedings of the 1st ACM Conference on Information-Centric Networking*, ACM, pp. 117–126, 2014.

[11] Y. Jin, et al. "Towards joint resource allocation and routing to optimize video distribution over future internet." In *2015 IFIP Networking Conference (IFIP Networking)*, IEEE, pp. 1–9, 2015.

[12] Y. Desmouceaux, et al. "6LB: Scalable and application-aware load balancing with segment routing." *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, 2018.

[13] C. Filsfils, et al. "Segment routing architecture." RFC 8402, Jul 2018. URL https://rfc-editor.org/rfc/rfc8402.txt.

[14] M. Enguehard, et al. "A popularity-based approach for effective cloud offload in fog deployments." In *30th International Teletraffic Congress (ITC 30)*, Vienna, Austria, Sep 2018.

[15] The Fast Data Project (fd.io). "Vector Packet Processing (VPP).", 2016. URL https://wiki.fd.io/view/VPP.

[16] The Apache Software Foundation. "Apache Traffic Server.", 2017. URL http://trafficserver.apache.org.

[17] A.-m. K. Pathan and R. Buyya. "A taxonomy and survey of content delivery networks." *Grid Computing and Distributed Systems GRIDS Laboratory University of Melbourne Parkville Australia*, vol. 148, pp. 1–44, 2006.

[18] I. ISO. "23009–1: 2014: Information technology-dynamic adaptive streaming over HTTP (DASH)."

[19] M. K. Mukerjee and D. Naylor. "Practical , real-time centralized control for CDN-based live video delivery." *Sigcomm 2015*, pp. 311–324, 2015.

[20] D. Naylor, et al. "The cost of the S in HTTPS." In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, ACM, pp. 133–140, 2014.

[21] I. Marinos, et al. "Network stack specialization for performance." In *ACM SIGCOMM Computer Communication Review*, vol. 44, ACM, pp. 175–186, 2014.

[22] E. Jeong, et al. "mTCP: a highly scalable user-level TCP stack for multicore systems." In *NSDI*, vol. 14, pp. 489–502, 2014.

[23] C. Filsfils, et al. "The segment routing architecture." In *Global Communications Conference (GLOBECOM), 2015 IEEE*, IEEE, pp. 1–6, 2015.

[24] S. Previdi, et al. "IPv6 Segment Routing Header (SRH)." Internet-Draft draft-ietf-6man-segment-routing-header-13, Internet Engineering Task Force, May 2018. URL https://datatracker.ietf.org/doc/html/draft-ietf-6man-segment-routing-header-13. Work in Progress.

[25] D. E. Eisenbud, et al. "Maglev: A fast and reliable software network load balancer." In *NSDI*, pp. 523–535, 2016.

[26] P. Patel, et al. "Ananta: Cloud scale load balancing." *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[27] H. Che, et al. "Hierarchical web caching systems: Modeling, design and experimental results." *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[28] C. Fricker, et al. "A versatile and accurate approximation for LRU cache performance." In *Proceedings of the 24th International Teletraffic Congress*, International Teletraffic Congress, p. 8, 2012.

[29] Y. Desmouceaux, et al. "A content-aware data-plane for scalable video delivery (tech. rep.).", 2018. Available at https://enguehard.org/papers/content_cdn_techrep.pdf.

[30] M. Garetto, et al. "Efficient analysis of caching strategies under dynamic content popularity." In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, IEEE, pp. 2263–2271, 2015.

[31] P. Pfister and Y. Desmouceaux. "Segment routing based load balancer plugin.", 2019. URL https://github.com/Oryon/vpp-dev/blob/6lb/src/plugins/srlb/srlb_plugin_doc.md.

[32] A. Kivity, et al. "kvm: the linux virtual machine monitor." In *Proceedings of the Linux symposium*, vol. 1, Ottawa, Dntorio, Canada, pp. 225–230, 2007.

[33] C. Byström, et al. "Locust.", 2018. URL https://locust.io/.

[34] W. Reese. "Nginx: the high-performance web server and reverse proxy." *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[35] *tc(8) - Linux man page*, 2.7.5 ed., Dec 2001.