Experimental Evaluation of Memory Management in Content-Centric Networking

Giovanna Carofiglio, Vinicius Gehlen, and Diego Perino
Bell Labs, Alcatel-Lucent, France, first.last@alcatel-lucent.com

***Abstract*—Content-Centric Networking is a new communication architecture that rethinks the Internet communication model, designed for point-to-point connections between hosts, and centers it around content dissemination and retrieval.**
Most of the issues faced by the current IP infrastructure in terms of mobility management, security, scalability, which are accrued by today's Internet trends, find a natural solution in CCN shift from IP addresses to named data.

In this paper we explore the impact of storage management on the performance of multiple applications sharing the same CCN infrastructure and we quantify the effectiveness of static storage partitioning and dynamic management techniques in providing service differentiation. To this purpose, we implement a set of storage management techniques in the open source CCNx prototype and perform extensive experiments in a real test-bed under fairly realistic network conditions. Our experimental results allow to clarify the relation between CCN chunk-level caching and Quality of Experience (QoE) perceived by end users.

I. Introduction

The current usage of the Internet is increasingly focused around content dissemination and retrieval, while its communication paradigm, introduced in the 1960s, is still based on the host-to-host conversational model. The consequent mismatch in terms of communication semantics between "where" (location of the content) and "what" (the content to retrieve) is at the origin of a number of significant deficiencies of today's Internet. Such rising issues, like security, location-dependency, or efficient content search and dissemination, currently require ad-hoc application layer solutions (e.g. CDNs, P2P systems).

Content-Centric Networking is a new paradigm recently proposed by PARC [1] aiming at rethinking the network architecture with a fundamental shift from an end-to-end to a content-centric communication model. This approach is designed to deal with today's trends and proposes a unified, direct way to solve the aforementioned issues. In fact, CCN brings significant advantages, ranging from enhanced security to simplified network management. In addition, the enablement of generalized in-network caching is a key factor for realizing real economies by avoiding repeated transmissions of identical data over the same paths at the expense of cheap memory.

The CCNx open source project developed a software prototype of the CCN architecture, and some simple experiments are presented in [1] and [2] to validate CCN principles. In the CCN architecture the management of nodes' storage capacity plays a fundamental role on system performance, and therefore it deserves a separate study, which is the object of this paper.

Related work

The impact of storage management has been already analyzed in the context of *Web caching*. Previous work mainly focus on content-level replacement policies based on different criteria as recency, frequency, size, QoS, etc. [3]. The role played by Web caching architectures has been addressed by Rodriguez *et al.* in [4], while service differentiation via cache partitioning has been introduced by Lu *et al.* in [5], and evaluated using the Squid Web caching proxy. Benefits of caching in the context of Content Distribution Networks (CDNs) has recently been analyzed by experimental evaluations of CoBlitz [6] and Coral [7] CDNs deployed on PlanetLab.

However, previous work do not readily apply to the CCN architecture where every content is split into a sequence of uniquely named chunks, and *multiple applications* share a common distribution infrastructure. CCN is based on a *receiver-driven transport* protocol where data is only transmitted in response to chunk requests expressed by users. These requests are independently forwarded, and *chunk-level caching* is performed by every node in the network. The complex interplay of these factors affects chunks' location, hence the QoE perceived by the users and the cost for the network operator.

Contribution

In this paper, we clarify the role of storage management on the performance of multiple applications running on the top of the CCN infrastructure. We extend the CCNx prototype to support a generic function-based chunk replacement strategy, and to provide per-application storage management. We analyze the relation between transport, caching and users' QoE, and the impact of static storage partitioning and dynamic management techniques. Our evaluation is performed by means of experiments in a real test-bed under fairly realistic network conditions.

The rest of the paper is organized as follows. Section II describes CCN architecture and the CCNx software prototype. Section III presents the storage management techniques under study and their implementation in the CCNx prototype. Sec. IV details the experimental evaluation, while Sec. V concludes the paper.

II. CCN

According to the novel communication paradigm introduced by Jacobson *et al.*[1], the focus is on the dissemination and retrieval of information rather than on the interconnection of endpoints. Content can be of various nature, including data generated 'on the fly' for conversational and interactive services or the dissemination of live events.

As already observed, every content is split into chunks whose unique names follow a hierarchical structure inspired by that of web's URIs. The hierarchical naming structure enables aggregation as in IP routing tables and more generally allows system scalability.

During a first phase of content publication, nodes with content that may be interesting to others announce content names, or the prefixes of the names, as with IP routing announcements. Once a content is published, nodes that are interested in a particular content send out Interest packets containing names of the requested chunks. At transport level, a receiver-based content delivery is realized by a sliding window update as in TCP: a given number of Interests packets can be outstanding and the reception of the requested data packets enables the sending of following Interest packets.

At network level, routers are responsible for processing incoming Interests and send the corresponding chunk to the interface from which it was requested. CCN routers are composed of three main blocks: a content store (CS), a pending interest table (PIT) and a forwarding information base (FIB). For every incoming interest, the router first checks CS and, if the requested chunk is present, it returns the chunk directly. Otherwise, it must forward the Interest to a next hop determined via the FIB and create an entry in the PIT, so that when the chunk is received, the router will know where to send it.

Depending on forwarding, strategies defined by a 'strategy layer' [1], Interest packets may propagate along multiple paths towards potential locations of the data, and pull the data down to the requesting nodes over paths followed by Interest packets, with no need for explicit routing. Unlike IP forwarding, intermediate nodes may transparently cache data packets in the CS for later transmission, e.g. in response to an Interest for the same chunk.

If several users request the same chunk, only one interest is forwarded upstream and a single PIT entry records all the interfaces over which the chunk must be returned. This naturally realizes multicast. The CS stores received chunks and plays the role of a cache, avoiding the need to repeatedly fetch popular contents. Within a cache, content chunks are stored and replaced according to a specific policy, like LRU (Least Recent Used) or LFU (Least Frequently Used).

A. *CCNx prototype*

A software prototype implementing the basic functionalities of the CCN architecture is developed by the CCNx project [8]. In this paper we focus on CCNx version 0.2.0, released in December 2009, which is composed of the following main modules (Fig. 1):

- the core networking daemon, *ccnd*, providing caching, forwarding, and packet authentication functionalities. It is currently developed in C.
- a *repository*, responsible to permanently store applications' data. It is currently developed in Java.
- a number of proof of concept applications developed in C or Java, e.g. *ccncatchunks* a tool for file download.

Concerning cache management, chunks are stored in a hashtable and an array of pointers is used to keep track of data order. A skip list of pointers sorted by chunk name is used for fast content search, instead.

Chunks are also removed from the cache when they become obsolete. A *FreshnessSeconds* attribute is associated to every chunk by the content source, and indicates how long data can be stored in cache after being received.

III. Storage management policies

The present section describes static and dynamic storage management policies with the common objective of providing service differentiation to multiple applications sharing the same CCN infrastructure. These techniques can be used to manage the CCN Content Store (called cache in the rest of the paper) together with some chunk replacement policies. In this paper we consider LRU and In-cache LFU (or simply LFU) replacement techniques which are described in [3].

A. *Static cache partitioning*

With static cache partitioning we refer to differentiated per-application storage allocation, where each application is statically assigned a fraction of the shared memory.

In the literature, cache partitioning has been analyzed in the context of web caching and content distribution by Lu *et al.* [5]. The authors propose an adaptive control scheme based on approximate linear differential equations, to self-tune the size of different partitions. Such mechanism guarantees convergence towards a proportional hit rate differentiation among applications, but differs from our work as it is not able to guarantee a minimum hit probability to each of them, and additionally requires an on-line monitoring of applications' performance for adaptive partition re-sizing.

B. *Dynamic storage management*

Caching systems in the Internet often allow to store a given content for a limited period of time, called Time To Live (TTL). The reason behind is twofold: *content obsolescence*, that is contents become obsolete after a period of time which is specific to the generating application (e.g. chunks of conversational applications become obsolete in hundreds of milliseconds, or live streaming in few seconds), and *cache scalability*, which can be obtained exploiting TTL for opportunistic caching without cache coordination. The choice of TTL is therefore strongly related to such two factors and particularly to the application/service the chunk belongs to. The impact of TTL on overall caching performance is for instance considered in [9]. In the following, we assume that chunk TTL is a property of the application and that it is directly signaled in the chunk header.

The amount of valid data stored in cache for a given application is approximately given by its arrival rate times its TTL. As a consequence, some applications may not completely fill their partitions all the time, and part of memory may be wasted. We consider two dynamic storage management schemes, proposed in [10]: priority and weighted fair management. These two schemes dynamically share the memory between different applications so that all capacity left available in a cache may be reused.

C. Priority storage management

This algorithm aims at prioritizing some applications or services w.r.t. others. Such choice may be driven by QoE requirements and/or pricing motivations. We assume N applications $\{A_i\}_{i=1}^N$ with different sorted priorities such that A_i has priority over A_{i+1}, sending queries to a cache with finite memory size. In case of hit, the relevant chunk is sent to the querying application, otherwise the data is retrieved from an upstream cache and priority storage management algorithm, detailed in Algorithm 1, is applied. If the cache is not full, the chunk is stored without removing any other chunk. If the cache is full, a chunk of an application with lower or equal priority should be removed to accommodate the new one.

Algorithm 1 Priority storage management

```
At chunk arrival of class A[i]
for (j=N; j>=i;j--)
      if( IsNotEmpty(A[j]) )
             remove_from(A[j]);
             insert(chunk, A[i]);
             return;
      endif
```

D. Weighted fair storage management

The weighted fair policy has the objective to share the storage among different applications proportionally to some predefined weights, while guarantying full resource utilization.

In this case we assume N applications $\{A_i\}_{i=1}^N$ with associated weights $\{w_i\}_{i=1}^N$, and we denote as x_i the amount of memory used by application A_i. The weighted storage management technique, detailed in Algorithm 2, can guarantee a minimum amount of memory $x_i = \frac{x w_i}{\sum_{i=1,\dots,N} w_i}$, $\forall i = 1, ..., N$. To this goal, when a chunk should be removed to make room for a new one because the cache is full, the algorithm tries to equalize x_i/w_i, $\forall i = 1, ..., N$. If the cache is not full, the new chunk can be stored without removing any other data.

E. Implementation in CCNx

We extend the CCNx prototype to support a generic cache replacement policy and the management techniques described above. For portability to future CCNx releases, we do not modify the original cache organization but we introduce an additional data structure. In details, we implement a doubly-linked list where pointers to data are sorted according to a given parameter. This parameter is computed by the replacement policy according to a specified function, and chunks are removed from the tail of the list. For instance, under LRU the parameter is the timestamp of the last request for the chunk, while under LFU it is a counter of chunk requests.

Algorithm 2 Weighted fair storage management

```
At chunk arrival of class A[i]
max   = 0;  j_max = 0;
for (j=1; j<=N;j++)
     if( x[j]/w[j] > max )
           max = x[j]/w[j];
           j_max =j;
     endif
remove_from(A[j_max]);
insert(chunk, A[i]);
```

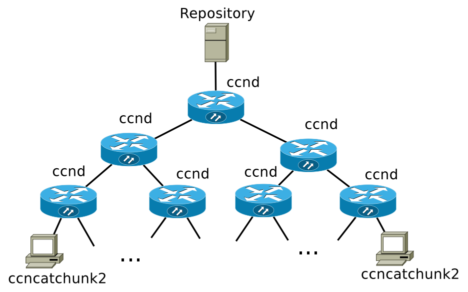


Fig. 1. Topology used for the evaluation. Links have a capacity of 10 Gbps.

We use a separate list per each application in order to reduce the overhead of replacement operations. These lists are stored in a multi-list structure and each of them is identified by a unique ID. We assume the application a chunk belongs to is explicitly specified in the chunk name.

If the cache is statically partitioned, once a partition is full, a chunk is removed according to the corresponding replacement policy in order to insert a new one. With dynamic storage allocation, chunks are removed only when the whole cache is full, and a management algorithm is required to select the target application. The corresponding replacement policy is then applied on the selected application list.

We also develop a customized version of the *repository* by extending the *CCNFileProxy* utility. Our tool allows us to scale experiments to a larger number of users and content items.

IV. Experimental evaluation

The experimental evaluation is performed running our modified version of CCNx prototype on the Grid5000 platform [11]. We use a set of machines with dual 2.83 GHz Intel Xeon E5440 Quad-core and 8 GB of RAM. Each machine runs Linux Debian 2.6.18-6-amd64 and it is connected to a switch through a 10 Gbps line card. The chosen virtual topology is set-up configuring nodes' forwarding table with the *ccndc* control program, and the duration of every experiment is 6 hours.

A. Reference scenario

We consider the *aggregation network* topology reported in Fig. 1 where all links have a capacity of 10 Gbps and the propagation delay is negligible. Content items are stored in our customized version of the *repository* connected to the root node, and they are split into chunks of 4 KB each: this is the default data packet size in the current CCNx implementation. We further suppose the cache can store x=10000 chunks and, unless otherwise specified, it implements LRU replacement policy.

Content items belong to two applications, A1 and A2, both counting 400 content items grouped in K=200 classes of decreasing popularity. Class popularity distribution is Zipf with parameter $\alpha_1 = 0.7$ and $\alpha_2 = 2.4$ for A1 and A2 respectively, i.e. the probability to request a content of class k, $k = 1, 2, ..., K$, is $q(k) = c/k^{\alpha}$, where c is the normalization constant, $c = \left(\sum_{k=1}^{K} 1/k^{\alpha}\right)^{-1}$. Clearly, $k = 1$ represents the most popular class. Content size distribution of A1 is mix Lognormal (body) - Pareto (tail), the average size being $\sigma_1 = 80$ KB, while content size distribution of A2 is Uniform between 800 KB and 2.45 MB, the average size being $\sigma_2 = 1.6$ MB. A1 represents a typical HTTP Web workload [12], [14], while A2 considers an application with more skewed popularity distribution and larger content items.

Users are connected to leaf nodes in Fig. 1 and generate content requests according to a Poisson process of intensity $\lambda = 5$ req/s. The motivation behind the Poisson assumption comes from the observation that Internet traffic is well modeled at session level by a Poisson process [15].

Unless otherwise specified, we assume a request breakdown of 80% A1 - 20% A2 content requests. Content download is performed by means of the *ccncatchunks2* tool, and we limit the transmission window to a constant W=1 for all users, as in the current CCNx implementation. This implies that requests for the first chunk of a content arrive according to a Poisson process and then the reception of the first data packet triggers the request for the second chunk and so on.

B. Performance metrics

For each application we evaluate the average per-class *chunk hit probability* at every node. It estimates the average probability to find a chunk of a given popularity class at a given level of the network. This metric provides an indication of the resources required to transfer chunks to users, i.e. network cost, but, as we will see in the following section, it also allows to predict and control the end-user performance. In addition, we evaluate the average per-class *chunk download time*, that estimates the time elapsing between the expression of an interest for a chunk of a given popularity class and the reception of that chunk. This metric can be used to quantify the QoE perceived by users.

C. User QoE and chunk-level caching

Let us now consider a scenario where the cache is not partitioned. Fig. 2 (black lines) reports the average chunk download time for the 10 most popular classes of content (we omit the other classes as they represent a small amount of traffic). One can easily observe that A1 users download chunks faster than A2 ones, and users downloading most popular contents experience better QoE for both applications.

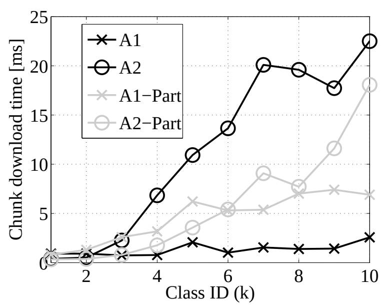


Fig. 2. Average chunk download time as a function of class popularity under request breakdown 80% A1 - 20% A2 and LRU replacement policy.

Indeed, user QoE is strictly related to the hit probability at different levels of our network topology as shown in Fig. 3 (black lines). At leaf nodes, the chunk hit probability decreases as popularity decreases for both applications, and A1 chunks are found with higher probability. At higher levels of the network we observe a "filtering effect" [16], where chunk popularity is modified by requests already served by lower level caches.

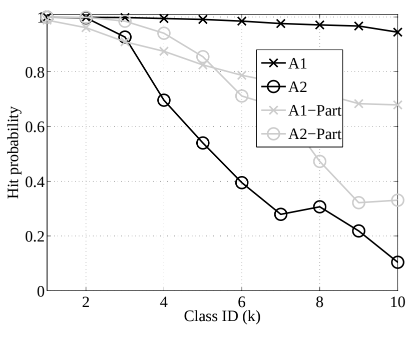
Global content popularity. In this scenario, application A1 is advantaged because more active in terms of content requests than A2. In fact, if the cache does not distinguish among applications the hit probability depends on the global popularity which is strongly related to request breakdown. As a consequence, even if A2 content items have a more skewed popularity and are larger in size, A1 is perceived as more popular because of its higher request rate.

In Fig. 4 we analyze the impact of the cache size. As expected, the hit probability of both applications decreases as the cache size decreases. The more active application A1 is advantaged for all cache sizes except for class k=1. In fact, even if A2 has a lower request rate, its most popular class is globally the most requested as a consequence of content popularity distribution. Therefore, if the cache does not distinguish among applications, cache dimensioning should take into account the global content popularity distribution and not only per-application characteristics.

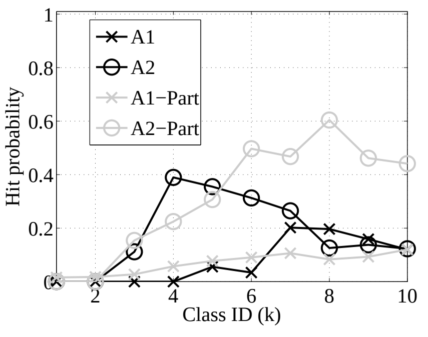
LFU replacement policy. We observe trends similar to the LRU case as reported in Fig. 5 (black lines). In this scenario, A2 chunk download time follows a step function while all chunks of A1 can be downloaded in a constant time regardless of their popularity. This behavior is related to LFU replacement policy which does not take into account temporal locality and is effective in localizing chunks at different levels of the network according to the global content popularity in a static workload.

D. Static cache partitioning

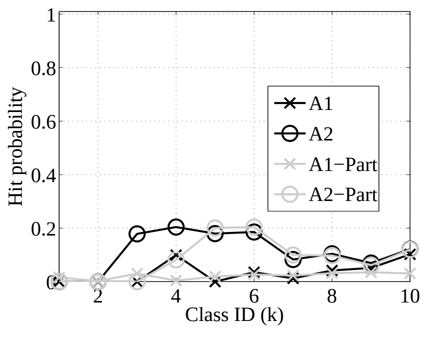
Let us next consider a scenario where the cache is partitioned among the two applications: 20% is statically assigned to A1 while the remaining 80% is devoted to A2. The LRU replacement policy is independently applied on every partition. From Fig. 2 one can infer that the chunk download time of application A2 is reduced w.r.t the non-partitioned case, while



(a) leaf nodes



(b) intermediate nodes



(c) root node

Fig. 3. Hit probability as a function of class popularity under request breakdown 80% A1 - 20% A2 and LRU replacement policy.

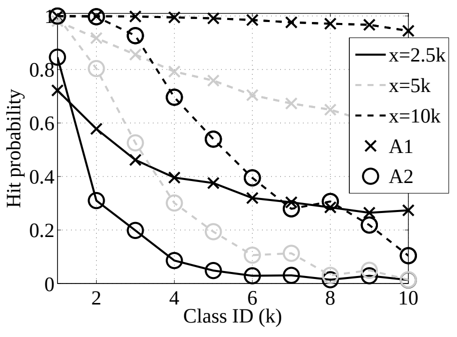


Fig. 4. Hit probability at leaf nodes under request breakdown 80% A1 - 20% A2 for different value of cache size.

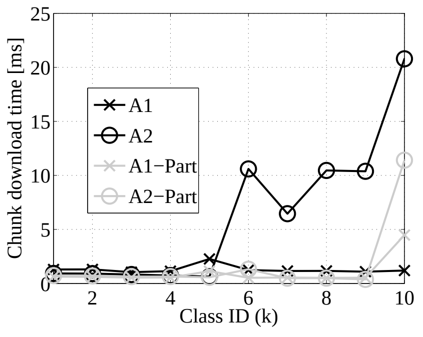


Fig. 5. Hit probability at leaf nodes under LFU replacement policy.

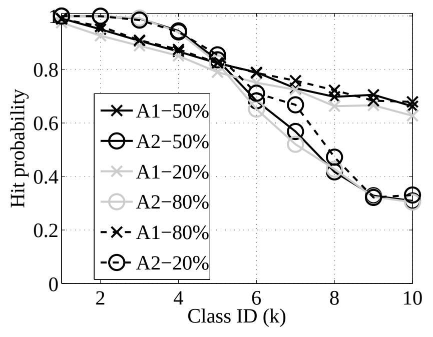


Fig. 6. Hit probability at leaf nodes for several traffic breakdown with cache partitioning.

the one of A1 increases. This is due to the hit probability value at different levels of the network (Fig. 3), and in particular at lower level caches, where we notice an increase of A2 hit probability w.r.t the non-partitioned case.
With cache partitioning, every application independently manages its memory space, thus A2 hit probability is not affected by A1 traffic. The amount of memory devoted to A2 is larger than the amount it can exploit if the cache does not discriminate among applications, hence it improves its hit probability. On the contrary, the amount assigned to A1 is lower than what this application obtains if the cache is not partitioned, therefore its hit probability is reduced.

Insensitivity to traffic breakdown. In Fig. 6 we let request breakdown vary among the two applications. As a result, we observe that the hit probability of both applications is *insensitive* to traffic breakdown variation thanks to the independence among applications in terms of hit/miss rate obtained through static cache partitioning: as a consequence the QoE experienced by A1 users is not affected by the traffic generated by A2 users and viceversa.

Fig. 7 shows the impact of different partition sizes. As expected, the resulting hit probability of each application under different partition sizes is a function of the considered application and it is not affected by the global content popularity. Therefore, we can conclude that partition dimensioning based only on application characteristics allows to control and predict the QoE perceived by users related to a specific application.

E. Dynamic storage management

In this section we explore the potential benefits deriving from the deployment of the two dynamic storage management techniques presented in Sec.III-C,III-D, priority-based and weighted fair respectively, through a comparison with the static cache partitioning analyzed in the previous section. We assume a cache size of x=6000 chunks equally split among the two applications under static partitioning. Under weighted-fair storage management, we assume weights of both applications are equal, whereas under priority storage management, application A1 is supposed to be prioritized with respect to A2.

An over-provisioned partition. We first consider a scenario where $TTL_1 = 6\ s$ and $TTL_2 = 600\ s$. In this scenario, under static storage management application A1 is not able to fully exploit its memory space. Indeed, the average number of valid

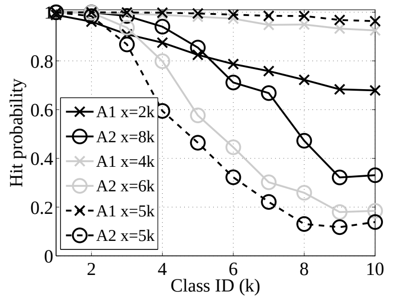


Fig. 7. Hit probability at leaf nodes for several partition size with cache partitioning.

data, given by the product of the content request rate and the application-specific TTL, is lower than the amount of storage statically assigned to A1. Fig. 8 highlights that under dynamic storage management, application A2 is able to use the memory not exploited by application A1, that is wasted in case of static partitioning, without affecting A1 hit probability.

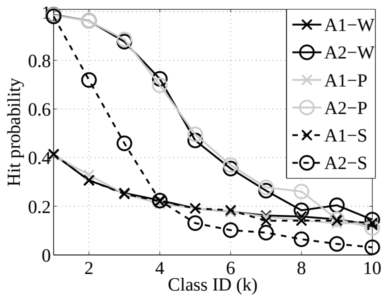


Fig. 8. Hit probability at leaf nodes for static (S), priority (P) and weighted fair management (W). A1 TTL=6 s, A2 TTL=600 s.

Fully exploited partitions. We further consider a second scenario where $TTL_1 = 180\ s$ and $TTL_2 = 600\ s$ (Fig.9): in this setting both applications are able to fully exploit their assigned storage under static cache partitioning. Instead, under priority management policy we notice a performance decay for application A2, due to the fact that A1 is allowed to exploit the memory at the expense of A2 chunks. On the contrary, weighted storage management guarantees a minimum amount of memory to both applications and, as in this case applications' weights are equal, hit probabilities are the same as those observed in the static partitioning case where the storage is equally shared.

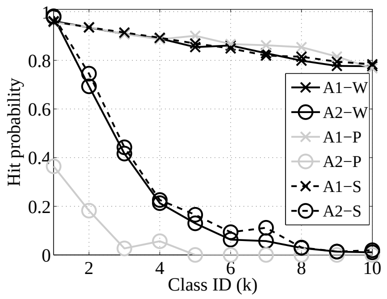


Fig. 9. Hit probability at leaf nodes for static (S), priority (P) and weighted fair management (W). A1 TTL=180 s, A2 TTL=600 s.

V. Conclusions

The paper focuses on the impact of content caching on overall system performance within the CCN architecture proposed in [1]. In a nutshell, the CCN proposal by Jacobson et al. solves the existing mismatch between a host-to-host communication paradigm and the current Internet usage centered around content publication and retrieval. In the CCN design, in-network chunk-level caching and receiver-based transport have a fundamental role in determining the efficiency of content delivery and the consequent QoE perceived by end-users under limited storage resources.

To the best of our knowledge, this is the first attempt to the evaluation of the impact of storage management techniques in presence of multiple applications sharing the same CCN infrastructure. Specifically, we implemented a set of storage allocation techniques in the CCNx prototype, and performed extensive experiments in a real test-bed under fairly realistic network conditions. Experimental results point out the benefits deriving from a static per-application cache partitioning and provide indications on how to perform an optimal sizing of cache partitions based on parameters like content popularity or obsolescence. Dynamic storage allocation enhances service differentiation by adding flexibility in storage resources sharing. In particular, the weighted fair storage allocation algorithm appears to be a promising mechanism to yield high system efficiency while avoiding memory under-utilization. We plan to further investigate its properties as future work.

Acknowledgements

This work is funded by the French national research agency (ANR), CONNECT project, under contract number ANR-10-VERS-001.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. of ACM CoNEXT '09*.
[2] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard, "Voccn: voice-over content-centric networks," in *ReArch '09: Proceedings of the 2009 workshop on Re-architecting the internet*. New York, NY, USA: ACM, 2009, pp. 1–6.
[3] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, 2003.
[4] P. Rodriguez, C. Spanner, and E. Biersack, "Analysis of web caching architectures: hierarchical and distributed caching," *Networking, IEEE/ACM Transactions on*, vol. 9, no. 4, pp. 404 –418, aug. 2001.
[5] Y. Lu, T. F. Abdelzaher, and A. Saxena, "Design, implementation, and evaluation of differentiated caching services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 5, pp. 440–452, 2004.
[6] K. Park and V. S. Pai, "Scale and performance in the coblitz large-file distribution service," in *Proc. of ACM NSDI'06*.
[7] M. J. Freedman, "Experiences with coralcdn: A five-year operational view," in *Proc. of NSDI*, 2010.
[8] "CCNx project," http://www.ccnx.org.
[9] E. Cohen and H. Kaplan, "Aging through cascaded caches: performance issues in the distribution of web content," in *Proc.of SIGCOMM*. New York, NY, USA: ACM, 2001, pp. 41–53.
[10] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, "Memory allocation schemes for chunk-based content oriented networks," http://perso.rd.francetelecom.fr/muscariello/papers/techrep_diff_caching.pdf.
[11] "Grid5000 platform," http://www.grid5000.fr.
[12] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *In INFOCOM*, 1999, pp. 126–134.
[13] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system," in *Proc. of ACM IMC '07*.
[14] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proc. of ACM SIGMETRICS '98*.
[15] E. Chlebus and J. Brazier, "Nonstationary poisson modeling of web browsing session arrivals," *Information Processing Letters*, vol. 102, no. 5, pp. 187 – 190, 2007.
[16] C. Williamson, "On filter effects in web caching hierarchies," *ACM Trans. Internet Technol.*, vol. 2, no. 1, pp. 47–77, 2002.