# A Framework for Network Aware Caching for Video on Demand Systems

BOGDAN CARBUNAR, Florida International University
RAHUL POTHARAJU, Purdue University
MICHAEL PEARCE, Motorola Solutions
VENUGOPAL VASUDEVAN and MICHAEL NEEDHAM, Motorola Mobility

Video on Demand (VoD) services allow users to select and locally consume remotely stored content. We investigate the use of caching to solve the scalability issues of several existing VoD providers. We propose metrics and goals that define the requirements of a caching framework for CDNs of VoD systems. Using data logs collected from Motorola equipment from Comcast VoD deployments we show that several classic caching solutions do not satisfy the proposed goals. We address this issue by developing novel techniques for predicting future values of several metrics of interest. We rely on computed predictions to define the *penalty* imposed on the system, both network and caching sites, when not storing individual items. We use item penalties to devise novel caching and static content placement strategies. We use the previously mentioned data logs to validate our solutions and show that they satisfy all the defined goals.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Caching, video on demand, content distribution networks

**ACM Reference Format:**

Carbunar, B., Potharaju, R., Pearce, M., Vasudevan, V., and Needham, M. 2013. A framework for network-aware caching for video on demand systems. ACM Trans. Multimedia Comput. Commun. Appl. 9, 4, Article 30 (August 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2501643.2501652

## 1. INTRODUCTION

Most cable providers today support Video on Demand (VoD) solutions, enabling subscribers to access items from a central database, transfer them over a Content Distribution Network (CDN) and view them on their Set Top Boxes (STBs). In this work we focus on CDNs of cable providers (e.g., Comcast, Charter and Time Warner) that are built on a CATV transport network. A typical CDN has a hierarchical architecture and consists of a central Video Server Office (VSO) and multiple Video Hub Office (VHO) sites, all connected through a high-bandwidth, low-latency fiber ring (see Figure 1). The VSO hosts the content library and handles the supported content life cycle; the VHO sites serve disjoint subscriber regions.

Author's address: B. Carbunar; email: carbunar@gmail.com.

DOI: http://dx.doi.org/10.1145/2501643.2501652

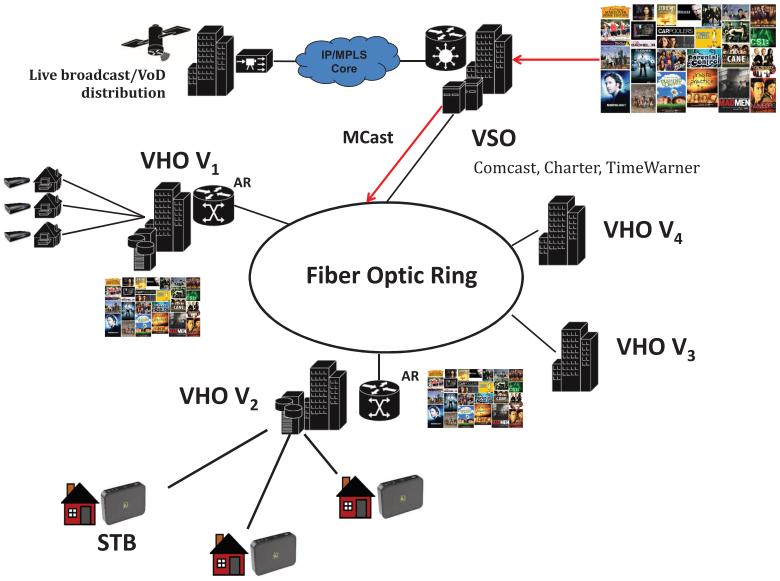



Fig. 1. System Architecture. Thick lines denote the ring topology links, connecting the VSO and the VHOs. Links are bi-directional. The VSO has a B-1 streaming server and each VHO has a lower streaming capacity server, B-3. User requests that cannot be satisfied from local VHO caches are forwarded to the VSO who then sends the content.

Current VoD solutions require each VHO to store all the content supported by the system. This approach ensures high content availability and simplifies the content management process: Newly supported content is propagated from the VSO to all the VHOs, using an efficient multicast protocol over the ring topology. It presents, however, significant hardware scalability issues, as the size of the content library constantly increases with the number of supported items and the evolution of content encoding, moving from standard to high definition and eventually to BlueRay and 3D content.

The use of caching at the VHO level seems to provide a natural solution, enabling the independent management of each VHO site and making hardware scaling dependent on local demand. However, due to VHO level misses (occurring when requested content is not cached), this approach introduces a trade-off between the additional miss traffic imposed on the network links and the hardware scaling cost.

The first contribution of this article consists of identifying several metrics that are fundamental for a VoD CDN architecture, along with goals that need to be satisfied by efficient solutions. The architecture considered differs from existing caching deployments in several aspects. First, the content considered is not only of different sizes but has different consumption rates (e.g., standard vs. high definition). Second, the content imposes strict fetching requirements: once an item download starts, the transfer rate has to equal or exceed the item's consumption rate. Third, unlike traditional caching solutions (e.g., [Dahlin et al. 1994; Kangasharju et al. 2002; Qiu et al. 2001; Wauters et al. 2006; Zaman and Grosu 2011]), here missed items do not need to be cached. Items predicted to be less valuable can be simply streamed from other sites. Finally, given the unique limitations of the SSD storage technologies, the amount of data written on the cache needs to be minimized.

A second contribution consists of the proposal of efficient caching and static placement algorithms that predict a *penalty* value for each item: the network and storage cost of not storing the item during a future interval. We rely on user behavior periodicites extracted from real-world user request logs

recorded by Motorola equipment from several VoD deployment sites. Predicting future values of metrics of interest from previous observations is not a novel idea. We propose, however, a novel approach where an initial weighted average prediction is further refined as consumption patterns evolve. Function based strategies have been used before in caching, for instance, Cao and Irani [1997]. However, the item network penalty we define depends on a unique combination of factors designed for the system model considered, including the size of the item, the predicted number of requests to be received for the item within a predefined future interval, and the cost predicted to be imposed on the network by the item fetching process. We consider item fetching costs that are not only a function of hops traversed, but also one depending on more complex available bandwidth evaluations of existing network paths.

Our solutions take advantage of the existence of streaming servers at the VSO and at all the VHO sites (see Section 2). This allows VHO sites to stream missed requests from peer sites while not forcing them to cache all missed items. We then use item network penalties to drive not only the replacement algorithm—which items to evict from a cache—but also the decision of which items to reliably transfer and cache and which to stream and not cache.

We have implemented our solutions in Java and ns-2 and we have evaluated their ability to satisfy the goals we have identified. We have used log traces collected from Motorola equipment deployed at several Comcast sites, with thousands of users and millions of requests occurring over a period of more than two weeks. Our conclusions are that existing algorithms like LRU, LFU or GDS impose unreasonable daily cache overwrite values (5.5-12 TB per day for 4 TB caches) and are unable to satisfy all user streaming requests at the needed consumption rate. Our techniques significantly reduce the total network traffic (half the value of LRU, which outperforms LFU and GDS), improve its distribution on the network links (one order of magnitude better than LRU) and reduce the cache overwrite value per day to a fraction (10–20%) of the cache size.

The article is organized as follows. Section 2 introduces the system model and describes our data logs. Section 3 studies the metrics and associated goals relevant to the studied framework. Section 4 evaluates the performance of several existing caching algorithms on our data logs. Section 5 describes a prediction based caching (PBC) solution, and Section 6 presents a network aware caching (NAC) algorithm. Section 7 describes a segment based variant of NAC (NAC-Seg) and Section 8 proposes a Static Placement Algorithm (SPA). Section 9 compares the performance of our solutions on the metrics introduced in Section 3. Section 10 describes related work and Section 11 concludes this article.

## 2. SYSTEM MODEL

We consider the content distribution networks of Comcast, Charter and TimeWarner VoD deployments (see Figure 2 for an illustration). The Video Service Office (VSO) is the central data repository. The VSO processes each content item as it enters the system, packages it and stores it in a local *content library*. The VSO has a high-capacity streaming server (e.g., Motorola's B-1 Video Server [2012a]) that can be used to stream items directly to users. The Video Hub Office (VHO) is a smaller replica of the VSO, serving a geographical sub-region of the area served by the VSO. Each VHO consists of a storage component and smaller capacity streaming servers (e.g., Motorola's B-3 [2012b]).

The number of VSOs and VHOs reflects the size of the customer base served by the system. We used the assumption that a VHO can serve 20k homes [Sorento]. Comcast has about 330k subscribers in the Chicago area [Miner 2012], implying a deployment of between 8 and 16 VSOs / VHOs combined, which is consistent with the number we were given (10 in 2010). Thus, we assume that most markets with populations smaller than Chicago would have fewer than 10 VSOs / VHOs (combined) in their systems.

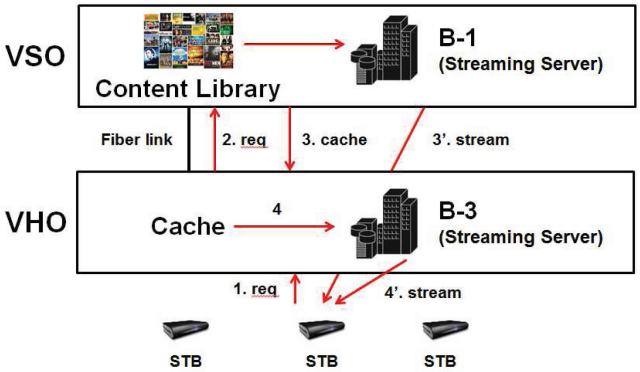



Fig. 2. Single VHO Architecture. Both the VSO and VHO have streaming servers, allowing them to stream requested item to clients.

The storage at VSO and VHO sites is a combination of hard drives and solid state drives [Motorola 2012a; Server 2012b]. Solid state drives are preferred for being able to perform high speed parallel, random read/writes with small I/O blocking.

The VSO and the VHOs are connected through a fiber ring (e.g., an OC192 - 9.6 Gbps Metro Ring). The fiber ring consists of bidirectional links between sites (VHOs or VSO). Traffic is directed on the shortest path to the destination. The bidirectional fiber ring provides resilience: When a link or site fails, the two nearest surviving sites become end stations and loop back their ends of the ring. Traffic over the failed link is then re-directed over the remaining links in the ring.

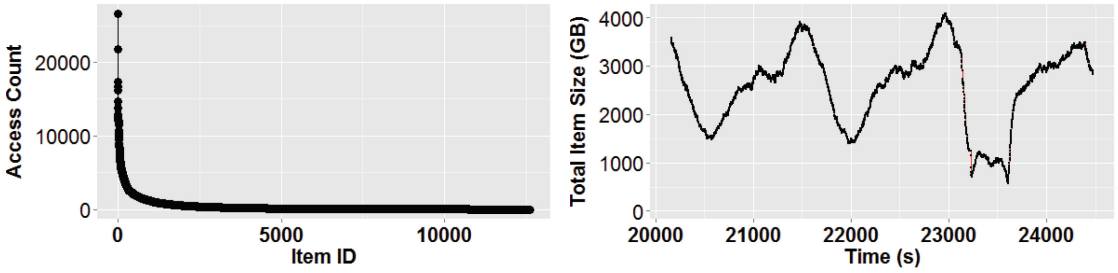
For each new item in the system, the VSO uses a multicast algorithm, for instance, NACK-Oriented Reliable Multicast (NORM) [Adamson et al. 2009], to distribute the item to each VHO. Each VHO chooses whether to cache the item or not. Users browse the content listed by the VoD service and request items using set-top-boxes (STBs). Requests made by users are sent to the VHO server serving the sub-region containing them.

For simplicity we do not include all system details. For instance, a VHO is not connected directly to users but through intermediate nodes. We assume the nodes do not cache content, as they are typically fiber-coax conversion / break-out nodes, and do not contain any headend equipment. In the Warren, MI, case (see Section 2.1), each VHO has 30–40 nodes. Having a video cache in each would be prohibitively expensive. We note however that our solutions can be used in conjunction with caching at the node level, effectively creating a cache hierarchy.

Each VHO stores only a subset of the items stored at the VSO but has metadata for all the items in the central library. Whenever a miss occurs (a user requests an item not stored on the VHO cache), the VHO needs to fetch the item. The VHO can fetch it from another VHO site or from the VSO. The source site streams the item directly to the user or reliably sends it to the VHO where the miss occurred, that then caches and streams the item to the user (see Figure 2).

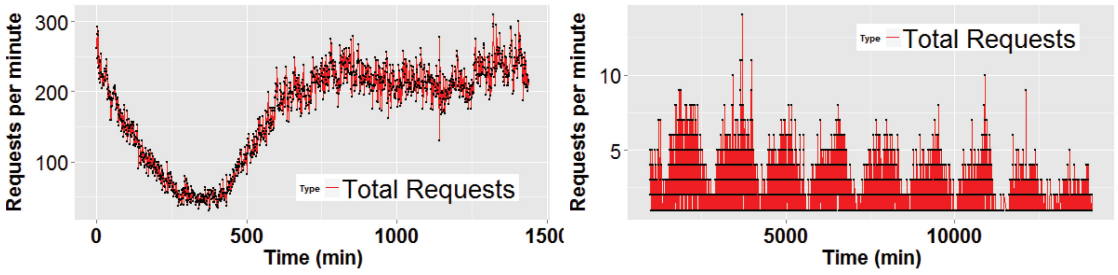
## 2.1 The Data

We have several data sets from VoD deployments in the US. We focus here on our largest data set, collected from Motorola VHO equipment from a Comcast VoD deployment in Warren (Detroit, MI). The "Warren" data has been collected over 18 full days, from August 16, 2010 to September 2, 2010. The total number of items accessed was 12,625 for a total of 4.6 million accesses. Each dataset consists of two types of data. The *content database* contains metadata of all content items stored on the VSO. Each entry in the content database refers to one item and has the format $[Id, Size(B), BitRate(bps)]$,




(a) (b)

Fig. 3. (a) Warren data statistics: per item access count distribution. (b) Time evolution of the (cache) storage space required by the items currently viewed by users for three consecutive days. Peak value exceeds 4TB.




(a) (b)

Fig. 4. (a) System-wide Requests Per Minute (RPM) for a single day. The lowest point is recorded at around 6am, when the system experiences the minimum user load. (b) RPM for a single item over 9 consecutive days.

listing the item's id, size, and consumption rate. There are two types of content encoding in our logs: standard definition, requiring a streaming rate of 3.7Mbps and high definition, with a 14.4Mbps rate. The *stream database* contains information about requests from VoD system users. Each entry refers to one user request and has the format $[StreamId, ContentName, StartT, EndT, IP]$, containing a unique stream id, the name of the content consumed, the consumption interval $[StartT, EndT]$ and the requesting IP address.

Figure 3(a) shows the distribution of the per-item access count for the 18 days. The $x$ axis is the item list in decreasing popularity order and the $y$ axis is the per-item access count. The access count distribution is long-tail, with the most popular item being accessed more than 26000 times but the item ranked 1000 (out of 12625 total items) being accessed only 1100 times. Figure 3(b) shows the evolution of the total size (in MB) of the items being viewed at any time, with one minute granularity, over three days (Fri. Aug. 27–Sun. Aug. 29) . The maximum size is 4.07 TB occurring at 22:50 on Sun. Aug. 31. Each day can be identified as one of the humps in the graph.

*Behavior periodicities.* We have investigated periodicities in the evolution of several metrics. The first metric we document is RPM, the (total and per item) number of requests received in a minute. Figure 4(a) shows the per-minute evolution of the total number of requests that the Warren VoD system received during the first recorded day. Figure 4(b) shows the evolution in time of the RPM value for one item for the entire duration when it was requested. The second metric is the total bandwidth required

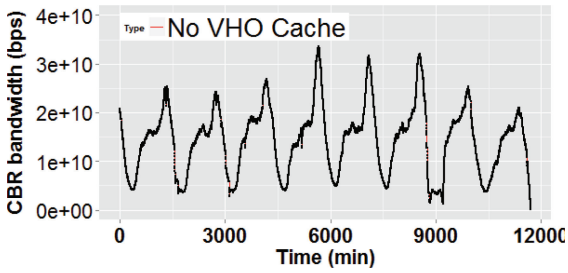



Fig. 5. Bandwidth required to satisfy all requests, over 8 days, when the VSO streams all requested items at their consumption rate. Each point on the x-axis represents one day.

to satisfy all the requests. Figure 5 shows the per-minute bandwidth required to satisfy all the user requests at their needed consumption rate, assuming that all the user requests are sent directly to the VSO, bypassing VHO sites.

The metrics studied vary during a day - high values are recorded around midnight, then a decrease to the lowest point occurs at around 6am and then values pick up again in the evening. This defines a consumption pattern: all days exhibit similar consumption behaviors. These metrics also exhibit weekly consumption patterns. For instance, items tend to be requested the least on Thursdays and the most on Fridays and Saturdays.

## 3. METRICS

Let $\mathcal{V} = \{V_1, \ldots, V_n\}$ be the set of VHOs and let $\mathcal{L}$ be the set of full-duplex (bidirectional), intersite links in the system. $\mathcal{L}$ includes also the links adjacent to the VSO. Let $MISS(V, \Delta T)$ denote the set of items missed on site $V$ during time interval $\Delta T$ and let Cache(V) be the set of items stored on site V at a given time. We now investigate the metrics relevant to the caching framework considered in our work and define the goals that should be satisfied by an efficient solution.

*Definition* 3.1 (*Traffic Metrics*). Let TMT, the Total Miss Traffic be the sum of the size of all the items missed on all VHOs over a time interval $\Delta T$: $TMT(\Delta T) = \sum_{i=1}^{n} Size(I)$, $I \in MISS(V, \Delta T)$, $\forall V \in \mathcal{V}$. Let TLT, the Total Link Traffic be the total traffic imposed on all the links in the system: $TLT(\Delta T) = \sum_i Traffic(L_i, \Delta T)$, $\forall L_i \in \mathcal{L}$.

TMT measures the traffic generated by all the VHOs and TLT measures the way this traffic is placed on the network's links. Note that $TLT(\Delta T) \geq TMT(\Delta T)$. For instance, assume in the system illustrated in Figure 1 that $V_1$ fetches item $I_1$ directly from the VSO and $V_2$ fetches item $I_2$ also from the VSO, but relays it through $V_1$. Then, $TMT = Size(I_1) + Size(I_2)$ and $TLT = Size(I_1) + 2Size(I_2)$. If however $V_2$ can fetch $I_2$ from $V_1$ or $V_3$, then TLT = TMT. This leads to our first goal.

*Goal* 3.1 (*Traffic Reduction*). Minimize *TMT*. Minimize *TLT-TMT*.

Our next metrics attempt to capture how well is the TLT traffic balanced on the system's links, leading to our next goal.

*Definition* 3.2 (*Congestion Metrics*). The Bottleneck Link Traffic (BLT) is the traffic imposed on the most utilized link in the system and the Minimum Link Traffic (MLT) is the traffic incurred on the least congested link. The System Link Balance, SLB, is the difference between BLT and MLT.

*Goal* 3.2 (*Balance*). Minimize *BLT*. Minimize *SLB*.

We now consider the ability of the system to deliver content at or above its consumption rate, to correctly render on the user STB. Our next goal captures this requirement.

Table I. Definition of Symbols

| Symbol | Definition |
|---|---|
| VSO | Video Service Office |
| VHO | Video Hub Office |
| RPM | Requests Per Minute |
| TMT | Total Miss Traffic |
| TLT | Total Link Traffic |
| BLT | Bottleneck Link Traffic |
| MLT | Minimum Link Traffic |
| SLB | System Link Balance |

*Goal* 3.3 (*User Satisfaction*). For any item $I$ being watched at time $T$ by a user, let $t(I, T)$ denote the number of bytes of $I$ transferred to the user up to time $T$ and let $c(I, T)$ denote the number of bytes of $I$ consumed by the user up to time $T$. Then, at any time $T$, ensure that $t(I, T) \geq c(I, T)$.

Finally, we focus on one important limitation of the technology used for storage at VHO sites: the SSD memory has a finite number of program-erase (P/E) cycles. Most commercially available flash products are guaranteed to withstand around 100,000 P/E cycles, before the wear begins to deteriorate the integrity of the storage [Thatcher et al. 2009]. This leads to our next goal.

*Goal* 3.4 (*Cache Overwrite*). Reduce the amount of data written on the cache to a daily value that is a fraction of the cache size.

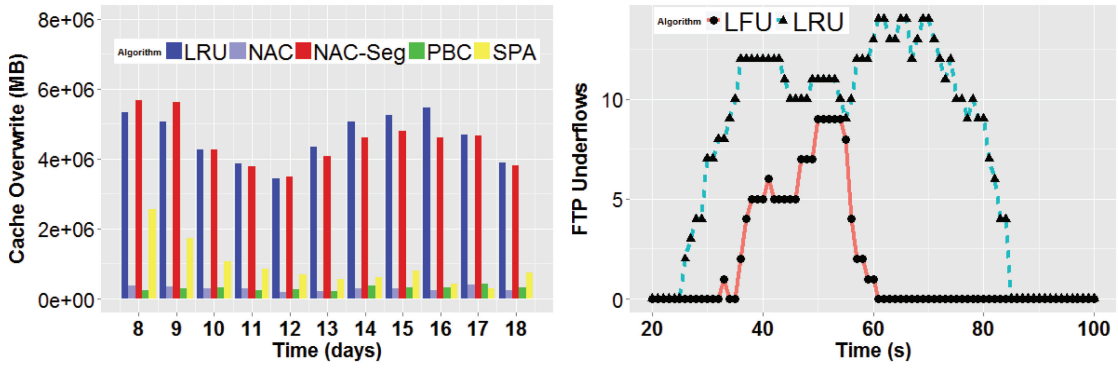
Table I summarizes the symbols we use throughout the article.

## 4. MOTIVATION

Instead of reinventing the wheel, our initial thought was to use existing caching techniques to drive the replacement process in each VHO. We document here our experiments with three classic policies: Least Recently Used (LRU), Least Frequently Used (LFU) and Greedy Dual Size (GDS) [Cao and Irani 1997]. LRU evicts the least recently used items until enough space exists to cache the missed item. LFU evicts the least frequently used items. When multiple items have the same (lowest) frequency, we have used LRU to give preference for eviction to the item that has been least recently used. In GDS, each item has an associated value, H, defined as the ratio between the cost (latency) to fetch the item and size of the item. During eviction, the item with the lowest H value, $min_H$, is replaced first and then all items reduce their H values by $min_H$.

We have tested LRU, LFU and GDS using log data from the Warren Comcast VoD deployment described in Section 2.1, consisting of 4 VHO sites and one VSO (see Figure 1). Each VHO has a 4TB cache, less than one third of all the content stored in the system (more than 12TB). The full-duplex fiber ring supports 1Gbps. Figure 6(a) shows the cache overwrite value per day (see Goal 3.4) imposed by each replacement policy on one VHO ($V_1$ of Figure 1). GDS may perform poorly due to the fact that small items, sparsely stored throughout the network, are given a higher value. This does not work well in this system, where large, frequently accessed items impose a high penalty on the network links.

Figure 6(b) shows the evolution in time of the number of simultaneous, reliable transfers (occurring during item misses) that LRU and LFU are unable to fetch at the required transmission rate. As described in Goal 3.3, such flows are unable to transfer items at the rate required for the content to correctly render on the user's STB. For LRU, up to 14 simultaneous transfers and for LFU up to 9 simultaneous transfers are unable to perform at their required rate.




(a) (b)

Fig. 6. (a) Cache overwrite values for LRU, LFU and GDS. LRU has the smallest footprint, of up to 5.5TB, while GDS imposes up to 13TB per day. LFU is in the middle, with up to 8TB per day. (b) NS-2 evaluation: Number of underflows for LRU and LFU: user satisfaction goal is not supported.

*Conclusions.* Existing replacement techniques do not satisfy the constraints imposed by a VoD CDN architecture. The cache overwrite value per day well exceeds the cache size (30%–200% more). Moreover, not all the users are able to receive the content they consume at the required rate, imposing unpleasant buffering delays during their viewing experience.

## 5. PREDICTION-BASED CACHING

In this section we propose a prediction-based caching (PBC) approach to address the goals of Section 3. PBC consists of a local caching component that is implemented on each VHO site, and a distributed caching component that is invoked when the first component fails (an item is missed). PBC addresses Goal 3.4 by extending the caching algorithm with the decision of storing vs. streaming missed items (see Section 5.2). PBC addresses the second part of Goal 3.1 as well as Goal 3.2 and Goal 3.3 by carefully placing the missed item traffic on the network's links (see Section 5.3).

*Cache organization.* The cache of each VHO site is organized into two lists. One list contains items that are currently consumed—the *viewSet*. The other list stores items that are not consumed but have not yet been evicted—the *stillCached* list. When a request for an item I is received by a VHO V, if I $\in$ Cache(V), V streams the item via its dedicated B3 server. If I $\notin$ Cache(V), V needs to forward this request to other sites that may store I, who then serve this request. V has the option of storing item I locally, or only relaying it to the user. Our approach for making this decision is based on penalties, described next.

### 5.1 Penalty Prediction

We define the penalty of an item on a site to be the cost incurred by the system if the item is not stored at the site during a certain, future interval. Specifically, in PBC, the penalty of an item is defined to be proportional to the item's size and to the item's popularity: $P(I, \Delta T) = S(I) \times Popularity(I, \Delta T)$. As such, items of larger size, that are requested frequently and are more difficult to fetch will likely have higher penalties. Predicting the penalty of an item depends on the ability to infer the future number of requests likely to be received for the item and also the future cost of fetching it over specific links.

We propose a prediction technique based on the observed periodic behavior of the requests-per-minute (RPM) metric for individual items documented in Section 2.1. Our idea is to use the past to

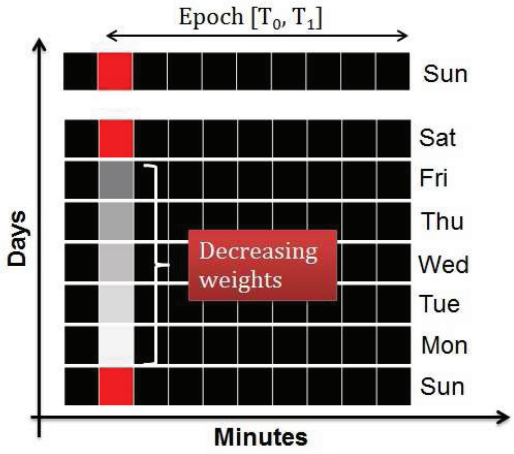



Fig. 7. Illustration of choice of weights.

predict the future. In the following we describe our approach using M as the metric of interest (RPM or link bandwidth), whose future values we need to predict. For each item stored in the system, each VHO site records observed values of the metric M for 7 days, sampled once per minute. This results in storing 1440 values per item per day.

We use the recorded history of M to compute a preliminary predicted value for future values of M. Specifically, for each item I and each minute $T \in \Delta T$, we use a weighted average of values of M recorded during the same minute of each day of the previous week to predict the value of M for I at a (future) minute T. The *initial* predicted value of M ($M_{init}$) is then

$$M_{init}(I, T) = \sum_{d=1}^{7} M(I, T - 1440 \times d) \times w_d \tag{1}$$

Each of the 7 previous days is given a weight, $w_d$, such that $\sum_{d=1}^{7} w_d = 1$. We give more weight to the previous day and to the same day one week before (see Figure 7).

PBC divides time into fixed-length epochs, which in our experiments are set to be 6 hours long. The reason for this value is that as seen in Figure 4(a), the consumption patterns change significantly during a day, with 4 major patterns, starting with 12 midnight: night, morning, early afternoon, late afternoon.

At the beginning of each epoch, the value of $RPM_{init}$ for each item I is computed for each minute of the (next) day using Equation (1). We define then $RPM_{pred}$ to be the sum of all the initial predicted values of RPM for the entire epoch, $RPM_{pred}(I) = \sum_{T=T_0}^{T_1} RPM_{init}(I, T)$, where $T_0$ is the first and $T_1$ is the last minute of the epoch. Thus, $RPM_{pred}$ denotes our prediction on how many times item I will be requested during that epoch. Note that $RPM_{pred}$ is computed only once per epoch.

During each epoch, we also record the observed values of RPM for each item I. Then, using these recorded values, at current time $T_c \in [T_0, T_1]$ we define the reactive value of an item I, $RPM_{react}(I, T_c)$ to be $RPM_{react}(I, T_c) = \sum_{T=T_0}^{T_c} RPM(I, T)$. That is, $RPM_{react}(I, T_c)$ denotes the total number of requests seen for I since the beginning of the epoch. We use $RPM_{pred}$ and $RPM_{react}$ to define the popularity of an item I at time $T_c$, as the weighted average of $RPM_{pred}$ and $RPM_{react}$: $Popularity(I, T_c) = RPM_{react}(I, T_c) \times \beta(T_c) + RPM_{pred} \times (1 - \beta(T_c))$ The weight $\beta$ is time dependent. We have considered two functions for $\beta$, (i) one exhibiting a logarithmic increase over time, $\beta(T_c) = \log T_c - T_0 / \log T_1 - T_0$

and (ii) one exhibiting a linear increase, $\beta(T) = T - T_1/T_2 - T_1$. Note that for both functions, the requirement $0 \leq \beta \leq 1$ holds. Our PBC implementation uses the linear $\beta$, which performed slightly better in our experiments.

### 5.2 Replacement and Streaming Decisions

PBC uses item penalties computed as described in Section 5.1 to (i) drive the cache replacement process and (ii) decide which items to cache (reliably transfer) and which items to stream to the user directly from the sites that stores them (and not store on the VHO). Streamed items do not impose a cache overwrite and also may impose less traffic on the network: an item is streamed only while the user is consuming it—our data shows that users frequently watch only a fraction of requested items.

On each VHO cache, the *stillCached* items are not currently consumed by any user (in the area served by the VHO) and as such are candidates for eviction during a miss. Let $stillCached = \{I_1, \ldots, I_n\}$. Let $S(I_i)$ be the size of item $I_i$. When a miss occurs for an item I whose size S(I) exceeds the available cache space, the penalties of I and of all the items in stillCached are computed. Let P(I) be the penalty of I and $P(I_i)$ be the penalty of item $I_i$ from stillCached. Then, I is stored in the cache only if there exists a "replacement set", a subset $R = \{I_{i_1}, \ldots, I_{i_r}\}$ of stillCached such that

$$\begin{aligned} \sum_{j=1}^{r} S(I_{i_j}) \;&\geq\; S(I) \\ csf \times \sum_{j=1}^{r} P(I_{i_j}) \;&<\; P(I) \end{aligned} \tag{2}$$

That is, the item is stored only if *stillCached* contains a set of items whose total size exceeds S(I) and whose total penalty is csf times smaller than P(I). In this case the replacement set is also evicted. If a replacement set is not found, the item is streamed directly to the user from another site that stores it. csf, the cache stability factor, defines how fast the replacement algorithm reacts to new items. A high value of csf generates a more static cache, since new items are being stored less frequently. That is, a new item has to have a penalty csf times larger than the penalty of other items in the cache, to be replaced.

It is desirable for the replacement set to be the one that has the minimum penalty among all subsets of *stillCached* of size larger than or equal to S. That is, we want to evict the set likely to inflict the minimum future penalty on the cache. It is straightforward to see that the 0-1 knapsack problem can be reduced to this problem. Thus, this problem is NP-hard (in fact it is NP-complete since verifying the solution takes polynomial time). As we will describe in the implementation section, we use a greedy heuristic to compute a candidate replacement set.

### 5.3 Traffic Balancing

When a miss occurs at a VHO site, the item needs to be fetched: either to be stored or to be forwarded to the requesting user. A straightforward approach is to fetch all missed items directly from the VSO. However, Goal 3.1 requires that the difference TLT-TMT should be minimized. This can be achieved by fetching missed items from the closest sites storing them. Formally, for a miss on an item I, VHO $V_i$ should fetch I from the site $V_j$ (the VSO or another VHO) such that $dH(V_i, V_j) = min(dH(V_i, V_k))$, $\forall V_k \in V$ s.t. $I \in Cache(V_k)$. Moreover, Goal 3.2 requires that link congestion is considered when fetching items—items should be fetched over the least congested links.

Our solution consists of three steps, executed when a MISS on item I occurs at a VHO site V: (i) discover which other sites have the content, (ii) choose the most suitable site and (iii) retrieve the content. In the following we detail each of these steps.

*Peer discovery.* We use a distributed hash table approach [Stoica et al. 2003; Rowstron and Druschel 2001; Ratnasamy et al. 2001; Ratnasamy et al. 2002; Zhao et al. 2001]. However, instead of storing content items according to DHT mechanisms, we store the *content directory* using a DHT. Specifically, each content item is mapped to a VHO site (e.g., through a hash of its content identifier) and each VHO is responsible for storing a portion of the naming space. Each time a VHO site caches or evicts an item, it retrieves and contacts the VHO responsible for storing the content directory entry corresponding to this item—the *pointer VHO*. For each content name of its responsibility, a pointer VHO maintains a list of VHOs storing it (holder VHOs). When a user requests content item I that her VHO V does not store, the VHO contacts the pointer VHO and retrieves the list of holder VHOs.

*Peer choice and item transfer.* Once V has found the list of holder sites, it fetches I from the site S that is the closest in terms of hop-count distance. In the case of a tie, V chooses randomly. V sends a request specifying whether the item needs to be transferred reliably or streamed. S needs to confirm that it can support the transfer, then marks item I as "no evict" until the transfer completes. Otherwise, S aborts and V repeats the above process for the next best site from I's holder list.

## 6. NETWORK-AWARE CACHING: NAC

We propose a caching algorithm that (i) is more reactive to changes in item popularity, for instance, items newly made available or removed from the central VSO library, and (ii) takes into consideration the network topology by making the penalty of a (missing) item dependent on the complexity of the fetching process, for instance, a missed item transferred from a neighbor should have a lower penalty than when transferred from a far away site. We call this solution "network aware caching," or NAC.

If the network topology is unknown, that is, if the VHOs are not aware of other peer VHOs and their paths to them, a network aware approach may still be possible by building an overlay network and using techniques such as virtual coordinates [Dabek et al. 2004] or network location services [Leong et al. 2007] to approximate VHO locations. We will explore such techniques in our future work.

In the following we first introduce a network penalty metric that extends the PBC's definition with the traffic cost imposed by fetching the item across the network. We then propose an alternative solution for predicting future values for different metrics. Finally, we use our technique to predict the item network penalties. We illustrate NAC's behavior using the pseudocode from Algorithm 1.

NAC is based on the observation that if a requested item is not locally stored at a VHO site, fetching the item will generate network traffic that is a function of the number of requests to be received for the item and the congestion of the links to be traversed by the item. Given an item I and a future interval $\Delta T$, let $Reqs(V, I, \Delta T)$ denote the number of requests to be received for I during interval $\Delta T$ on site V. Let $FC(V, I, \Delta T)$ denote the cost to fetch item I to site V. Assuming that I $\notin$ Cache(V), FC is a function of the path traversed by I to reach V. Then, the network penalty during interval $\Delta T$ is defined to be the cost incurred by the network if site V does not store I during $\Delta T$ (see Algorithm 1 lines 4–8):

*Definition* 6.1 (*Network Penalty*). The network penalty of an item I at a VHO site V during a future time interval $\Delta T$ is $NP(V, I, \Delta T) = Reqs(V, I, \Delta T) \times FC(V, I)$.

Reqs and FC are predictions of the actual number of requests and the cost of transfer for an item, during the future interval $\Delta T$. Let $\Delta T = [T_c, T_c + \delta]$, where $\delta$ is a system parameter (60 minutes in the experiments of Section 9). In NAC, the caching vs. streaming decision made during an item miss follows Equation (2), with $csf$ set to 1. We now describe the computation process for Reqs and FC.

**ALGORITHM 1:** NAC: Network Penalty computation.

```
1.Object implementation NAC;
2.  siteId : int; VHO id
3.  T_0, T_1, T_w : int; start and end of epoch, end of warm − up

4.  Operation netPenalty(Item I)
5.      T_c := getCurrentTime();
6.      Reqs := computeReqs(I, T_c, δ);
7.      FC := computeFC(I, T_c);
8.      return Reqs × FC × I.size();
9.  Operation computeReqs(Item I, Time T_c, Time δ)
10.     Reqs := 0;
11.     Acc := getAcc(RPM, I, T_c);
12.     for (T := T_c; T < T_c + δ; T++) do
13.         RPM_x := Pred(RPM, I, T, Acc);
14.         Reqs := Reqs + RPM_x; od
15.     return Reqs;
16.Operation computeFC(Item I, Time T_c)
17.     FC := max;
18.     for each (V ∈ 𝒱 & V.contains(I)) do
19.         PC := 0;
20.         for each (link ∈ getPath(V.getId(), siteId) do
21.             Acc := getAcc(RPM, link, T_c);
22.             time := 0; trans := 0;
23.             do
24.                 FPM_x := Pred(FPM, link, T, Acc);
25.                 trans := trans + link.getCap()/(FPM_x + 1);
26.                 time := time + 1;
27.             while (trans < I.size())
28.             if (PC < time) then PC := time; fi od
29.         if (FC > PC) then FC := PC; fi od
30.Operation Pred(Metric M, Item I, Time T, double Acc)
31.     M_init := 0;
32.     for (i := 1; i ≤ 7; i++) do
33.         M_init[I, T] := M_init[I, T] + M[I, T − 1440 × i] × w[i];
34.     if (T < T_w) then
35.         for (t := T_0; t < T_w; t++) do pred+ = M_init[I, t];
36.         return pred;
37.     else return M_init[I, T] × Acc;
38.Operation getAcc(Metric M, Item I, Time T_c)
39.     sumM := 0; sumM_init := 0;
40.     for (T := T_0; T ≤ T_c; T++) do
41.         sumM := sumM + M[I, T];
42.         sumM_init := sumM_init + M_init[I, T]; od
43.     return sumM/sumM_init;
```

### 6.1 Predicting the Future

Using the notation proposed in Section 5.1, we devise a new prediction solution for a generic metric M, whose initial prediction for a future minute is defined according to Equation (1) (see Algorithm 1, lines 32–33). Similar to PBC, NAC divides time into fixed-length epochs. Let $[T_0, T_1]$ denote one such epoch. During the epoch $(T_0, T_1]$, we record observed values of M, both for computing similar initial predictions in the future and for evaluating the accuracy of our prediction. We use the prediction accuracy to compute a final prediction $M_x$ for M during $[T_0, T_1]$. To achieve this, we reserve a short warm-up period $[T_0, T_w]$, $T_w < T_1$, at the beginning of each epoch, in which to collect enough real-time values for M. During the warm-up period, we lack enough "real" values of M to fine-tune the prediction. Then, we define $M_x(I, T) = \sum_{T=T_0}^{T_w} M_{init}$ for all $T \in [T_0, T_w]$ (see lines 34–36). Thus, for the first $T_w$ minutes, the predicted value of each item is constant.

Following the warm-up period, at any time $T_c \in (T_w, T_1]$, we use $M_{init}$ and values of M recorded from the beginning of the epoch, to evaluate the prediction accuracy. For each item I, we define the accuracy of the prediction for metric M at time $T_c$ to be $Acc_M(I, T_c) = \sum_{T=T_0}^{T_c} M(I, T) / \sum_{T=T_0}^{T_c} M_{init}(I, T)$. $Acc_M(I, T_c)$ is computed once per minute following the warm-up period (see lines 38–43). We use Acc to scale the initial prediction $M_{init}$ for a future minute T and compute our final prediction (line 37). That is, at time $T_c$, our prediction for the value of M at a future minute $T \in [T_c, T_1]$, denoted $M_x(I, T)$, is defined as

$$M_x(I, T) = M_{init}(I, T) \times Acc_M(I, T_c). \tag{3}$$

Note that Acc is computed at time $T_c$, whereas $M_x$ is the prediction for M at a future time $T > T_c$. The accuracy is reset at the beginning of each epoch, but not computed until the end of the warm-up period. For NAC, we define the length of an epoch to be one day and the warm-up period to be 30 minutes long. These values have produced the best results, when compared against combinations of quarter of a day epochs and 10–60 minute warm-up intervals.

### 6.2 Predicting Future Values of Reqs

When a miss occurs at time $T_c$, $Reqs(I, \Delta T)$ needs to be evaluated for each item I in the cache for the future interval $\Delta T = [T_c, T_c + \delta]$, where $\delta$ is a system parameter. We use the RPM metric introduced in Section 2 to define Reqs as the sum of the predicted values of RPM for the interval $\Delta T$ (see Algorithm 1 lines 9–15): $Reqs(I, \Delta T) = \sum_{T=T_c}^{T_c+\delta} RPM_x(I, T)$, where the predicted value $RPM_x(I, T)$ is computed according to Equation (3).

### 6.3 Defining the Fetch Cost

FC defines the load imposed on the network links when transferring an item I to a site V. As later detailed in Section 6.4, site V first discovers which other sites store item I. It then defines $FC(V, I)$ to be the minimum of the cost of all the paths from a site storing I to site V. If $PC(V_i, V_j)$ is the cost of a path between sites $V_i$ and $V_j$ then $FC(V, I) = min\ \{PC(V_j, V, I) | \forall V_j \in \mathcal{V}\ s.t.\ I \in Cache(V_j)\}$ (lines 18–29). We define the cost of a path for an item I to be the time to transfer I over that path, which is the time to transfer the item over the bottleneck link of the path: $PC(V_i, V_j, I) = max\{TransferT(l, I) | \forall l \in Path(V_i, V_j)\}$ (lines 20–28).

TransferT(l,I) defines the time to transfer I over a link $l$. To compute TransferT, we first define a new metric. For any link $l$, let FPM(l,T) be the number of simultaneous flows supported by $l$ during minute T. Our results from Section 2.1 show that the bandwidth requirements in the VoD system studied, exhibit a repetitive pattern. This allows us to use Equation (3) to compute future values of FPM. Note that to compute FPM's prediction, we need to record FPM values for all the links in the system (see Section 6.4 for more details).

Given FPM, we compute TransferT iteratively. At (current) time $T_c$, compute the prediction $FPM_x(l, T_c + 1)$ (line 24) and use it to predict how many bytes can be sent during minute $T_c + 1$ over link $l$ (BPM –Bytes Per Minute), using the formula $BPM_x(l, T) = Cap(l)/(FPM_x(l, T) + 1)$ (line 25). That is, the bytes transferred over link $l$ in one minute for one flow are determined by the capacity of $l$ divided equally among all existing flows on $l$ –the ones already there plus the one for item I. Continue this process, computing $BPM_x(I, T_c + 2), \ldots, BPM_x(I, T_c + T_f)$, until the sum of all BPM values, $\sum_{T=T_c}^{T_f} BPM(l, T)$ exceeds or equals Size(I) (lines 23–27). Then, set $TransferT(I, l) = T_f$.

### 6.4 Traffic Balancing

NAC extends the traffic balancing procedure of PBC (see Section 5.3). When a MISS on item I occurs at a VHO site V, NAC (i) discovers which other sites have the content, (ii) collects link FPM predictions

and (iii) chooses the site with the least congested path. Step (i) is identical to the one in Section 5.3. Step (iii) follows directly from step (ii) and the procedure detailed in the Fetch Cost definition above. For step (ii), each site stores history values (7 days at one minute granularity) for the FPM metric for all its adjacent links, as well as its prediction for the values of FPM in the future interval $\Delta T = [T_c, T_c + \delta]$, where $T_c$ is the current time and $\delta$ is a system parameter. During a miss, NAC needs to determine the fetch cost FC of I and of all the items in Cache(V). For this, V needs to collect FPM predictions for all the relevant links. To ensure the solution's scalability, we require each site to periodically collect FPM predictions from all the sites at most two hops away.

## 7. SEGMENT-BASED CACHING: NAC-SEG

We now investigate a segment-based extension of NAC. Each content item is divided into fixed size segments. The rationale behind this approach is that items that are never or seldom watched entirely impose lower storage and network overheads: some of their segments are not accessed. Our solution, NAC-Seg, makes segments the basis of operation. Specifically, a request for an item is translated into sequential requests for the segments of the item that are accessed by the user. A history of accesses needs to be maintained for each segment instead of for each item, leading to a higher storage overhead. Each segment has a penalty which is a function of the segment's popularity and its cost of fetching over the network. Segments become the caching storage unit: For each missing segment, a single segment, the one with the lowest penalty, needs to be evicted from the cache.

## 8. STATIC PLACEMENT ALGORITHM

We propose the use of the penalty metric defined for NAC, to implement a static placement algorithm, SPA. In static placement algorithms, caches are periodically pre-populated with relevant items. Pre-cached items are chosen such as to ensure that they are the most valuable for the period considered. Following item placement, missed items are fetched from other sources but are not stored in the cache. We now describe how SPA decides when and which items to pre-cache and how misses are handled.

*Choosing next cache memberships.* SPA is driven by the VSO. SPA divides time into epochs. The VSO performs the precaching process at the beginning of each epoch. The items considered for precaching are all the items available at the VSO at that time. SPA uses the network penalty (see Section 6) of each item for the *entire* next epoch to drive the pre-cache process, as follows. For each item, the VSO computes a global penalty (GP) value. For item I, $GP(I) = \sum_{i=1}^{n} NP(V_i, I, \Delta T)$, is defined to be the sum over the network penalties of item I at all VHOs. $\Delta T$ denotes the duration of the entire epoch. The VSO sorts all the items in decreasing order of their GP values. The VSO schedules each item for a serialized multicast transmission: after one item is transmitted, the next item from the sorted list is selected, removed and multicast. Each VHO receives all multicast transmissions. If the VHO already stores the item, it ignores it. Otherwise, the VSO uses Equation (2) to decide if it needs to store the item.

*Example.* To illustrate this approach, consider a VSO serving two VHOs, $V_0$ and $V_1$, and supporting 3 items, $I_1$ of size 6, $I_2$ of size 4 and $I_3$ of size 4. At the beginning of the current epoch, $V_0$ stores items $I_1$, with $NP(V_0, I_1) = 5$ and $I_2$, with $NP(V_0, I_2) = 6$. $V_1$ stores items $I_1$, with $NP(V_1, I_1) = 6$, and $I_3$ with $NP(V_1, I_3) = 5$. While not stored on $V_0$, $I_3$ has penalty $NP(V_0, I_3) = 3$. Similarly, while not stored on $V_1$, $I_2$ has penalty $NP(V_1, I_2) = 8$. We have ignored the $\Delta T$ value in the above definitions of the network penalty values, as it refers to the length of the current epoch, common in all NP definitions above. At the beginning of the epoch, the VSO computes $GP(I_1) = 9$, $GP(I_2) = 14$ and $GP(I_3) = 8$. Thus, the VSO first sends $I_2$. $V_0$ already stores it, thus ignores it. $V_1$ however evicts $I_3$ and caches $I_2$. The VSO then sends $I_1$, which both $V_0$ and $V_1$ ignore, since they already cache it. Finally, the VSO sends $I_3$, which neither $V_0$ nor $V_1$ caches.

*Handling misses.* Following the static placement process, for the remainder of the epoch, the cache membership remains unchanged. Missed items are streamed to the users requesting them, using the approach proposed in Section 6.4.

## 9. EVALUATION

In this section we evaluate and compare the caching and placement algorithms introduced in this article. Our evaluation has been conducted using (i) an event-based simulator we have implemented in Java and (ii) ns-2. Our event-based simulator emulates the content distribution network of Warren, MI, that consists of 4 sites, each with a 4TB cache (one third of the 12TB system wide content library) connected by a full-duplex 1Gbps fiber ring. We have used the Warren data (August 16, 2010 to September 2, 2010) described in Section 2.1. While we have used data sets from both 2008 and 2010, due to the similarity of the results, in the following we only report results over the 2010 data set.

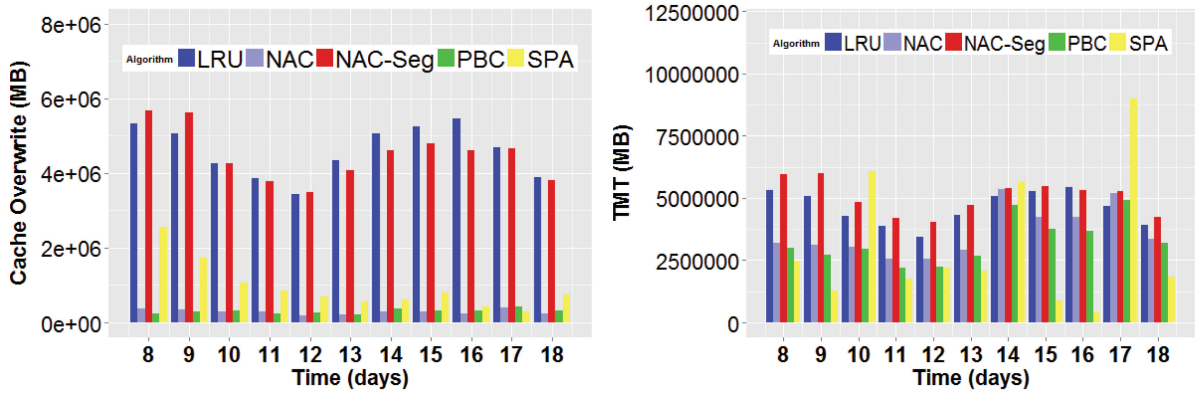
We have implemented the prediction algorithms and all the evaluated caching policies, including the traffic balancing solutions in Java. The event-based simulator runs over a sorted list of start and end times of user requests. The requests are organized into days, with special cases handling requests that span over 2 days. Each request is received by one of the VHO sites. The simulator uses the first 7 days of the logs to build request histories and does not report the performance of the caching algorithms during those days. This is because NAC needs 7 days to build the initial item prediction values. The output of the Java simulator, that is, the list of items missed, along with their miss time, VHOs missing them and the links on which they are fetched, to the ns-2 simulator.

We set the epoch length to 1 day for NAC but use quarter of a day epochs for SPA. For both NAC and SPA we set the warm-up period to be 30 minutes long. We have evaluated the performance of warm-up period length values ranging from 10 to 60 minutes and 30 minutes provided the best results. In NAC-Seg we set the segment size to 500MB. Smaller segment sizes lead to thrashing and larger segments impose unnecessary cache overwrites (especially for small items).

Figure 6(a) compares the performance of LRU, LFU and GDS in the same system setup. We observed that LFU often performs worse than LRU and consistently performs significantly worse than PBC and NAC. This is because (i) LFU does not consider the size of items during eviction—thus imposing high network penalties when a large evicted item is missed in the future—and (ii) the limited history of frequencies used by LFU does not accurately predict the future number of requests to be received for an item. Furthermore, Figure 6(a) shows that LRU and LFU consistently outperform GDS. In the following we compare the performance of PBC, NAC, NAC-Seg and SPA only against LRU.

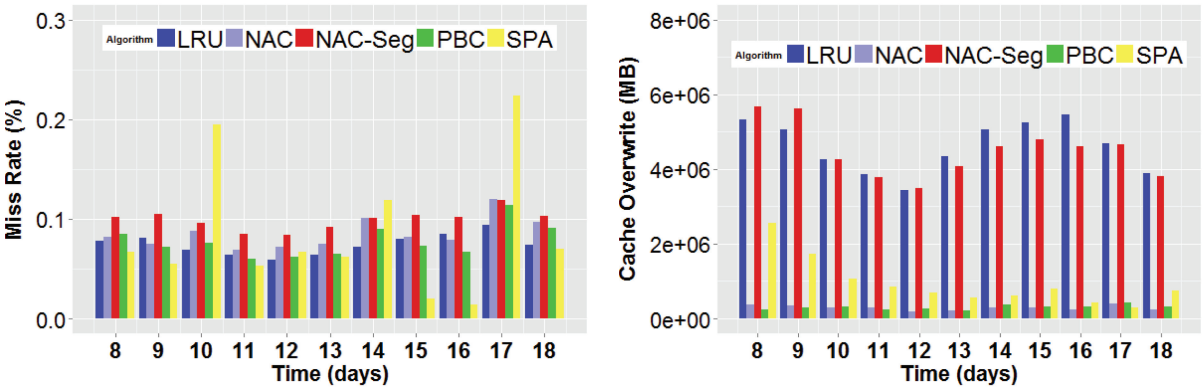
### 9.1 VHO Level Measurements

*Fine Tuning csf.* We first investigated the effects of the cache stability factor, csf, on our caching algorithms. A new item is cached only if the cache stores a set of items whose cumulative size exceeds the size of the new item and whose cumulative penalty is csf times smaller than the penalty of the new item. csf decides which items should be cached and which should be streamed, thus it impacts the TMT value and the cache overwrite value for any VHO cache. In the following we evaluate this impact on all 4 VHO caches, running PBC, for an entire day. Figure 8(a) shows the cache overwrite value for PBC for the 10th day when csf ranges from 1 to 1000. Figure 8(b) shows the TMT value for the same experiment. Note that as expected, a larger csf value decreases the overwrite value for each VHO cache. however, TMT is not significantly impacted by csf. The most significant improvement in the overwrite factor is experienced for a csf of 10, of around 300 GB per day for each VHO, less than 10% of the cache size. In the following experiments we set csf to 10 for all PBC runs.




(a) (b)

Fig. 8. (a) PBC daily cache overwrite dependence on csf. (b) Daily TMT dependence on the cache stability factor (csf).




(a) (b)

Fig. 9. Performance of NAC, NAC-Seg, SPA and LRU on VHO $V_1$ from Figure 1. (a) Miss rate. (b) Cache Overwrite: PBC, NAC and SPA overwrite the cache an order of magnitude less than LRU.

*Algorithm comparison.* We first study the performance for a single VHO on three metrics: the miss rate, the cache overwrite value and the total miss transfer (TMT). Figure 9 shows our results for VHO $V_1$ of Figure 1. Figure 9(a) shows that SPA exhibits the highest miss rate spikes, with a maximum of 23% per day. This is expected, since the cache membership does not change frequently. NAC and NAC-Seg follow, with a maximum of 12% per day. On average, LRU has the lowest miss rate.

Figure 9(b) shows that the cache overwrite values of PBC, SPA and NAC are by far smaller than those imposed by LRU and NAC-Seg on the cache. NAC and PBC overwrite at most 380 GB per day. SPA overwrites at most 2.5 TB per day, but it is frequently under 1 TB. LRU and NAC-Seg overwrite up to 5.7 TB on a single day. NAC outperforms PBC on the overwrite value, with a minimum of 190 GB overwritten per day. This graphs shows that LRU would considerably shorten the lifetime of the flash, overwriting more than the size of the cache per day.

Figure 10(a) shows the TMT value per day recorded by each algorithm. Most traffic imposed by all algorithms is under 6 TB per day, with the exception of one day for SPA, approaching 9TB. However,

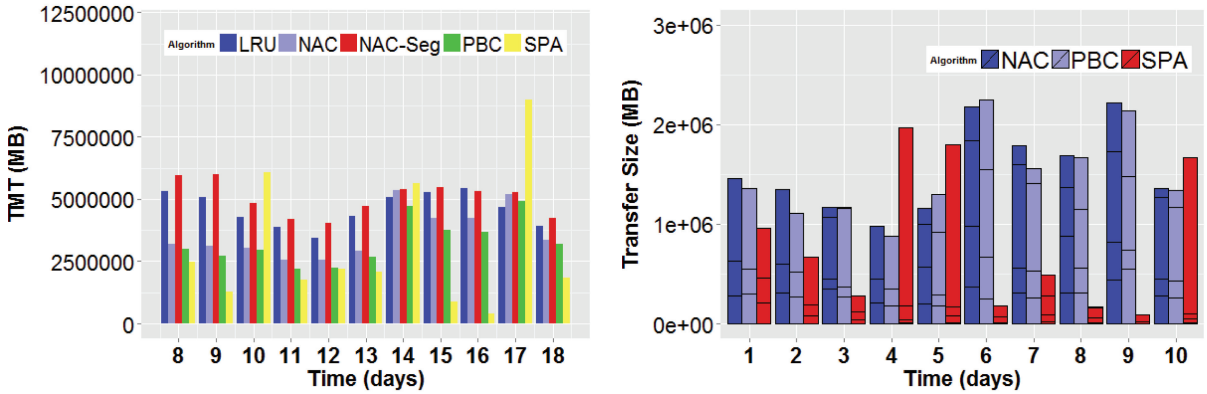



Fig. 10. (a) TMT of NAC, NAC-Seg, SPA and LRU on VHO $V_1$ from Figure 1. PBC and NAC generate half the traffic of LRU. (b) Transfer from peers: the total content size transferred by VHO $V_1$, running SPA and NAC, from the VSO and the other VHOs.

overall, SPA imposes the lowest TMT, with all remaining days being under 2 TB (on one occasion around 400GB per day). NAC's value ranges between 3 to 6TB per day. Note that PBC outperforms NAC by an average of a few hundred GB per day.

*Transfer from Peers.* Figure 10(b) shows a detailed view of part of Figure 10(a): the load balancing capabilities of SPA and NAC for $V_1$. In particular it shows the break-out of the daily TMT value among $V_1$'s peers, including the VSO. The bottom segment corresponds to traffic from the VSO, followed by segments for $V_2$, $V_3$ and $V_4$. While for SPA most missed items are fetched from the VSO, for NAC, on multiple occasions the traffic from $V_2$ exceeds that of items fetched from the VSO. As desired, even in the worst days, for NAC, much less than half of the traffic is generated from the VSO.

*Conclusions.* For variable sized items, the hit rate is not the best metric for measuring the performance of a caching algorithm. SPA's epoch length offers a tradeoff between the cache overwrite and the generated TMT values: Pre-caching 4 times a day imposes a 4 hold increase in the cache overwrite value when compared to once per day, but it significantly reduces the TMT value.

NAC outperforms NAC-Seg since (i) segmentation dilutes the statistics as fewer requests will be made on a per-segment basis than on items and (ii) given access pattern changes throughout a day, the cache will store many segments with close to 0 penalties. Such segments are almost always candidates for replacement, leading to frequent cache overwrites (see Figure 9(b)), instead of streaming. In the following we no longer evaluate NAC-Seg.

### 9.2 Traffic Load

We now focus on the traffic imposed by NAC, SPA and LRU on the ring's links. Figure 11 shows the TLT per day imposed by each tested algorithm. PBC and NAC are consistently outperforming LRU, on several occasions imposing half the TLT of LRU. SPA has a jittery performance, ranging from one tenth of the TLT of LRU to twice that of LRU. Figure 12(a) shows the traffic imposed on the most congested link. The traffic on the bottleneck link is significantly lower for PBC and NAC when compared with LRU—often less than half that of LRU. NAC is more stable than SPA, always improving on LRU. Figure 12(b) shows the load balance achieved by the three algorithms. NAC and PBC achieve a balance that is one order of magnitude better than that of LRU (a few hundreds GBs per day when compared

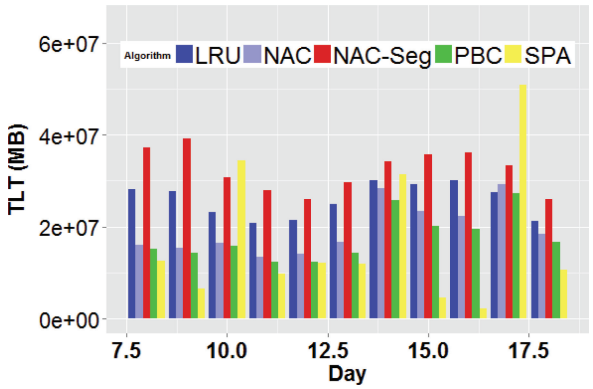



Fig. 11. Daily TLT imposed by NAC, SPA and LRU.

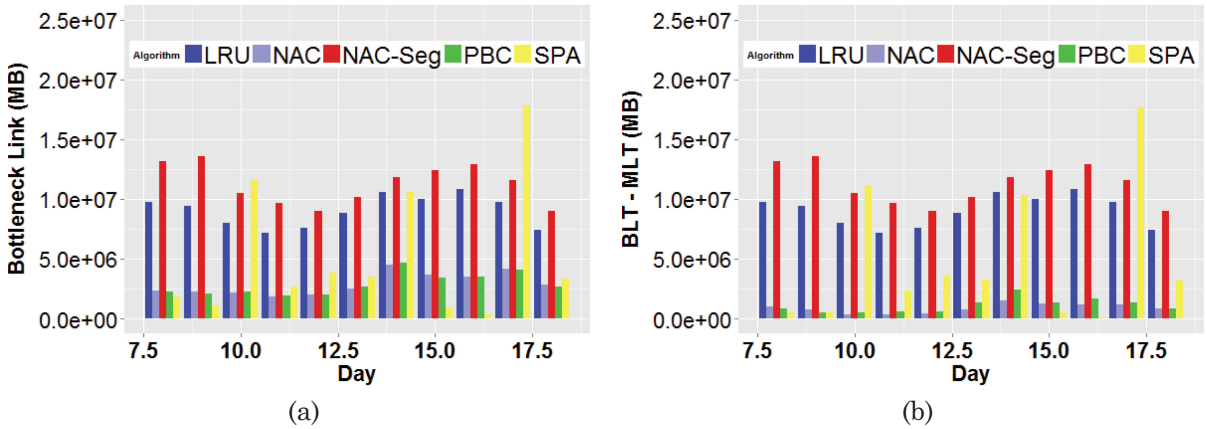
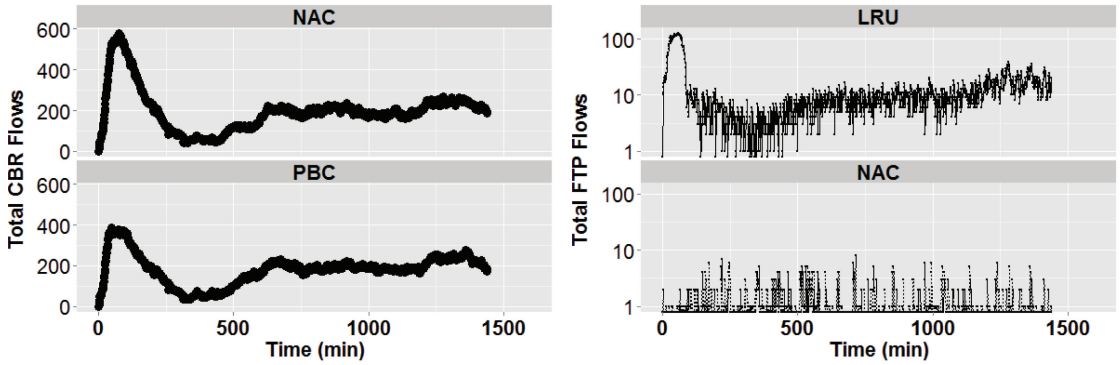



Fig. 12. Load imposed on the network by NAC, NAC-Seg, PBC, SPA and LRU. (a) Bottleneck link: NAC has bottleneck links of much less than half of those of LRU. (b) Traffic balance: NAC balances the load one order of magnitude better than LRU.

to 10 TB per day of LRU). SPA performs similarly, except for three days when it performs up to twice worst than LRU.

*Ns-2 link congestion evaluation.* In the following experiment, performed using ns-2, we study the effects of link congestion on the item transfer performance. For this, we have evaluated the number of flows simultaneously supported during the 10th day of the Warren data set, as generated by NAC, PBC and LRU. Figure 13(a) shows the evolution of the number of simultaneous streams (constant bit rate flows) imposed by PBC and NAC. LRU does not stream, thus is not shown. The data is displayed with a 1 minute granularity. PBC outperforms NAC at the beginning of the day, however, following early morning the two algorithms exhibit similar performance. The number of streams imposed by PBC and NAC is higher at the beginning of the day and the peak at the end of the day has smaller amplitude. This is because PBC and NAC are able to make more accurate predictions later in the day.

Figure 13(a) shows the number of simultaneous transfers (FTP flows) imposed by NAC and LRU during one day with 1 minute granularity. The y axis is shown in logarithmic scale. Unlike LRU, that does not generate any streams but only reliable transfers, NAC generates mostly streams, with only up to 7 simultaneous FTP flows: an order of magnitude less than LRU. The number of FTP flows imposed




(a) (b)

Fig. 13. (a) NS-2 evaluation of NAC. Evolution in time of the number of simultaneous streams generated during the 10th day. (b) NS-2 comparison of NAC and LRU. Evolution in time of the number of simultaneous reliable transfers (FTP) generated by NAC and LRU during the 10th day. NAC imposes an order of magnitude fewer FTPs than LRU.

by LRU is again larger at the beginning of the day. During that time, some of the LRU flows do not perform at the required transmission rate. This was shown in Figure 6(b), depicting the number of underflowing FTP transfers. As described in Goal 3.3, such flows are unable to transfer item bytes at the rate required for the content to correctly render on the user's STB. For LRU, up to 14 simultaneous transfers are unable to perform at the required rate.

## 10. RELATED WORK

This article extends our initial work [Carbunar et al. 2012] with a new predictive caching algorithm (PBC), a more detailed description of NAC and additional statistics and experimental measurements.

*Distributed caching.* The seminal work of Dahlin et al. [1994] introduced the concept of collaborative caching along with several caching algorithms. The algorithms rely on a client's ability to use other clients as caches: clients can store items for other clients or have a disk space dedicated to the system. Our work differs in this respect, since in our model clients can access the caches of other clients but do not decide what they store. Moreover, clients (VHOs) in the system considered in our work are not equal: clients decide between caches from which to retrieve missed items, based on a network cost metric.

*CDN-level caching.* Considerable work has been done in the area of Content Distribution Networks. Kangasarju et al. [2002] proposed collaborative and centralized placement techniques based on item popularity, item request rates and transmission distance. The collaborative "popularity-based" algorithm uses the number of accesses as the ranking criterion for items to place. Leff et al. [1993] proposed a ranking algorithm that defines cost based on distance. Qiu et al. [2001] proposed a centralized heuristic that only considered accesses from clients within a given radius around each node. Wauters et al. [2006] propose a set of distributed replica placement algorithms (RPAs), based on an Integer Linear Programming (ILP) formulation of the centralized content placement problem in ring based CDNs. Karlsson and Mahalingam [2002] show that most replica placement algorithms in CDNs are less efficient than a simple delayed-LRU caching algorithm. Karlsson and Mahalingam [2002] show that most replica placement algorithms in CDNs are less efficient than a simple delayed-LRU caching algorithm.

Zaman and Grosu [2011] focus on improving the efficiency of object replication within a given distributed replication group, where participating servers dedicate memory for replicating content requested by their clients. Laoutaris et al. [2007] investigate two causes of mistreatment in collaborative caching, cache state interactions and the adoption of a common scheme for cache management policies. They show that online cooperation schemes using caching are fairly robust to mistreatment caused by state interactions.

Caching for streaming data includes work on prefix caching [Sen et al. 1999; Wang et al. 2004], segment-based caching [Wu et al. 2001] and multicast cache MCache [Ramesh et al. 2001], where the main concern is minimizing the start-up latency. Caching for content distribution networks has also been addressed in theoretical frameworks in recent work [Borst et al. 2010; Amble et al. 2011]. Our work differs in that it (i) identifies the constraints defining the existing CDNs of VoD providers including Comcast, Charter and TimeWarner, (ii) proposes novel predictive caching solutions and (iii) validates the solutions using data collected from Motorola equipment from existing VoD deployments.

*TV and VoD caching.* Zhuo et al. [2008] study the cache placement problem in the context of time-shifted TV: multiple caches exist in various time zones and content consumption patterns shift in time. They propose a classification replication algorithm and a novel cache placement algorithm (CALLF). The algorithms focus on the time-varying nature of time-shifted TV, and explore previous stored information to reduce the cost of redeploying already cached data.

Most caching algorithms in VoD system are concerned with segment caching. Existing strategies include prefix caching, where the first bytes of items are cached [Wujuan et al. 2006] and interval caching, where random contiguous byte ranges of items are being cached [Park et al. 2001]. Our results show that a segment based approach did not perform as well as a 0-1 caching strategy.

## 11. CONCLUSIONS

*Contributions.* We have studied the effects of caching on the content distribution networks of existing Video on Demand systems. We have defined the CDN architecture used by popular VoD providers and we have introduced several essential metrics and associated constraints. Our analysis of user request logs from Motorola equipment in several VoD deployments has shown that user and content access behaviors exhibit periodicities. We have used these observations to introduce novel techniques for predicting future values of metrics of interests. We have used these techniques to define novel item penalty metrics. We relied on these metrics to introduce predictive based caching (PBC) and network aware caching (NAC) algorithms as well as segment (NAC-Seg) and static placement (SPA) variants.

*Evaluation conclusions.* NAC and PBC are better suited for the system architecture considered than LRU. They overwrite only fractions (10–20%) of the total cache size per day, generate significantly less traffic, most of which streaming, and balance the traffic one order of magnitude better than LRU. Unlike LRU, NAC, PBC and SPA are able to support all the user requests at their required consumption rates. The epoch length in SPA introduces a tradeoff between the generated cache overwrite and the total miss traffic values. NAC-Seg consistently performs worse than NAC.

*Limitations of our approach.* Studying the proposed solutions on larger scale deployments can provide a clearer image of their performance. The data sets we have are from a deployment consisting of 4 VHOs. We could extrapolate these sets to create synthetic data for multiple VHOs. This is however a delicate task whose outcome is hard to evaluate for correctness or realism of request behaviors, potentially impacting the credibility of simulation results.

We have focused our evaluation on a ring topology, given its prevalence for multi-VHO cable deployments as well as its intriguing interdependencies of traffic from different VHOs over a given link, and

the impact of the VHO location relative to the VSO. Evaluating our solutions on other topologies would certainly provide further insights into their performance. We note however that for instance, for a star topology, there are much fewer issues with respect to traffic balance, bottlenecks, TLT vs TMT, etc.

## REFERENCES


ADAMSON, B., BORMANN, C., HANDLEY, M., AND MACKER, J. 2009. NACK-oriented reliable multicast (NORM) transport protocol. Internet Engineering Task Force (IETF) RFC 5740.

AMBLE, M., PARAG, P., SHAKKOTTAI, S., AND YING, L. 2011. Content aware caching and traffic management in content distribution networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*.

BORST, S. C., GUPTA, V., AND WALID, A. 2010. Distributed caching algorithms for content distribution networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*.

CAO, P. AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

CARBUNAR, B., POTHARAJU, R., PEARCE, M., AND VASUDEVAN, V. 2012. Network aware caching for video on demand systems. In *Proceedings of the 13th International Symposium on a World of Wireless, Mobile and Multimedia Networks*.

DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. 2004. Vivaldi: A decentralized network coordinate system. *ACM SIGCOMM Comput. Comm. Rev. 34*, 15–26.

DAHLIN, M. D., WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. 1994. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*.

KANGASHARJU, J., ROBERTS, J. W., AND ROSS, K. W. 2002. Object replication strategies in content distribution networks. *Comput. Commun. 25*, 4.

KARLSSON, M. AND MAHALINGAM, M. 2002. Do we need replica placement algorithms in content delivery networks? In *Proceedings of the 7th International Web Content Caching and Distribution Workshop*.

LAOUTARIS, N., SMARAGDAKIS, G., BESTAVROS, A., MATTA, I., AND STAVRAKAKIS, I. 2007. Distributed selfish caching. *IEEE Trans. Parallel Distrib. Syst. 18*, 10, 1361–1376.

LEFF, A., WOLF, J. L., AND YU, P. S. 1993. Replication algorithms in a remote caching architecture. *IEEE Trans. Parallel Distrib. Syst. 4*, 11, 1185–1204.

LEONG, B., LISKOV, B., AND MORRIS, R. 2007. Greedy virtual coordinates for geographic routing. In *Proceedings of the IEEE International Conference on Network Protocols*. IEEE, 71–80.

MINER, M. 2012. A new cable deal for Chicago. http://www.chicagoreader.com/Bleader/archives/2012/05/02/a-new-cable-deal-for-chicago.

MOTOROLA. 2012a. B-1 Video Server. http://www.motorola.com/Video-Solutions/US-EN/Products-and-Services/Video-Infrastructure/On-Demand-Systems/B-1_US-EN.

MOTOROLA. 2012b. B-3: Motorola expands on demand platform to enhance support for rapidly growing on demand libraries. http://mediacenter.motorola.com/content/Detail.aspx?ReleaseID=10874&NewsAreaID=2.

PARK, S.-H., LIM, E.-J., AND CHUNG, K.-D. 2001. Popularity-based partial caching for vod systems using a proxy server. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*.

QIU, L., PADMANABHAN, V. N., AND VOELKER, G. M. 2001. On the placement of web server replicas. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*. 1587–1596.

RAMESH, S., RHEE, I., AND GUO, K. 2001. Multicast with cache (mcache): An adaptive zero-delay video-on-demand service. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*. 85–94.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*. ACM, New York, 161–172.

RATNASAMY, S., KARP, B., YIN, L., YU, F., ESTRIN, D., GOVINDAN, R., AND SHENKER, S. 2002. GHT: A geographic hash table for data-centric storage. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*. ACM, New York, 78–87.

ROWSTRON, A. I. T. AND DRUSCHEL, P. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*. Springer, 329–350.

SEN, S., REXFORD, J., AND TOWSLEY, D. 1999. Proxy prefix caching formultimedia streams. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*.

SORENTO. Solution architectures for cable video-on-demand. Sorento Networks, http://www.cascaderange.org/presentations/Solution_Architectures_for_Cable_Video_on_Demand.pdf.

STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. 2003. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1, 17–32.

THATCHER, J., COUGHLIN, T., HANDY, J., AND EKKER, N. 2009. Nand flash solid state storage for the enterprise, an in-depth look at reliability. In *Solid State Storage Initiative (SSSI) of the SNIA*.

WANG, B., SEN, S., ADLER, M., AND TOWSLEY, D. 2004. Optimal proxy cache allocation for efficient streaming media distribution. *IEEE Trans. Multimedia*.

WAUTERS, T., COPPENS, J., DE TURCK, F., DHOEDT, B., AND DEMEESTER, P. 2006. Replica placement in ring based content delivery networks. *Comput. Commun. 29*, 16, 3313–3326.

WU, K.-L., YU, P. S., AND WOLF, J. L. 2001. Segment-based proxy caching of multimedia streams. In *Proceedings of the International World Wide Web Conference*.

WUJUAN, L., YONG, L. S., AND LEONG, Y. K. 2006. A client-assisted interval caching strategy for video-on-demand systems. *Comput. Comm. 29*, 18.

ZAMAN, S. AND GROSU, D. 2011. A distributed algorithm for the replica placement problem. *IEEE Trans. Parallel Distrib. Syst. 22*, 9, 1455–1468.

ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. rep. UCB/CSD-01-1141, University of California, Berkeley.

ZHUO, J., LI, J., WU, G., AND XU, S. 2008. Efficient cache placement scheme for clustered time-shifted tv servers. *IEEE Trans. Consum. Electron. 54*, 4, 1947–1955.

Received October 2012; revised January 2013; accepted March 2013