Decentralized Hash-Based Coordination of Distributed Multimedia Caches

Anup Mayank Chinya Ravishankar
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92507
{*mayank,ravi*}@*cs.ucr.edu*

Krishna Bandaru Trivikram Phatak
TATA Consultancy Services Ltd.
{*bandaru.krishna,t.phatak*}@*tcs.com*

Abstract

We present a new approach to decentralized and cooperative caching of multimedia streams, based on the notion of virtual hierarchies, which result in very uniform distributions of loads across the system of caches. We show through simulations that our method greatly reduces loads at the server as well as latencies at the client. Our approach is robust, scalable and adapts quickly to changes in object popularity.

1 Introduction

Multimedia applications have become widespread over the Internet in recent years, and this trend will surely strengthen as more and more bandwidth becomes available [11, 16]. The term *streaming media* is typically used when the contents of multimedia objects are displayed as soon as the first chunk of data is received [3]. Specifically, the term is intended to exclude the approach of downloading and caching the entire object prior to playback.

Existing web caching systems are stand-alone systems that cache web objects independently in response to client requests. Caching is a reactive approach which caches an object only when requested. In replication, objects are pushed across one or more caching servers permanently. Replication of large multimedia objects (typically hundreds of megabytes) is hugely wasteful. Caching can be made efficient by breaking multimedia objects into smaller segments and distributing copies of each segment across a system of cooperating caches. Such intercache cooperation can reduce the storage requirements and enhance load-balancing, fault tolerance and scalability of the system.

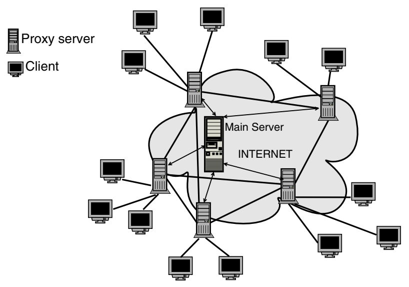

Figure 1. Caching and delivery on the internet

1.1 Caching and Cooperation

Caching (We use the term *cache* and *proxy* interchangeably in this paper) is a widely used method, and has been shown to be effective in reducing server loads and client latencies. Proxies also reduce network traffic by aggregating requests for the same object. Caching is likely to work particularly well with multimedia streams since they are static, sequentially accessed, and have high network resource requirements.

Figure 1 is intended to represent the ubiquity of caching on the Internet. Caching is central to the business of many companies, such as Akamai [1] and Mirror Image [2]. ISPs also commonly have proxying and caching mechanisms to improve performance. However, cooperation among these widely deployed caches is likely to be particularly important for multimedia objects. A typical one-hour movie is about 700 MB in size. In contrast, a typical web object is 5–50 KB in size. Since the number of available multimedia objects is also likely to grow, long-term caching of entire multimedia objects is not a good option, even with increases in disk capacities.

1.2 Our Contributions

Our method partitions a streaming object into segments and assigns them to caches organized in a virtual hierarchy architecture [17], using the name-based hashing scheme described in [13]. Virtual hierarchies outperform static caching hierarchies such as [15], in which the root cache commonly becomes a bottleneck. In contrast, loads are very evenly distributed in virtual hierarchies. We have modified the caching policy used in the virtual hierarchy approach, and have proposed a new cache replacement policy which suits the characteristics of multimedia streams, and quickly adapt to object popularities.

2 Related Work

In Adaptive and Lazy Segmentation policy [9], the object is first segmented, using the average access duration as the base segment size. Further,[9] reports that uniform segmentation performs, on average, as well as the exponential segmentation method of [16]. However, independent caching policies at proxies in [9, 16] cause replication of popular objects at many proxies, resulting in inefficient space utilization.

In MiddleMan [4] architecture proposed by Acharya and Smith, proxies in a LAN cooperate to increase aggregate amount of storage space at the proxy system and decrease loads at the main server. Cache space at the proxies is managed by a central coordinator, which keeps track of files hosted by each proxy. This approach to cache cooperation is inherently centralized, and subject to single-point failures.

In the Agent-based Caching Architecture [14] proposed by Tran et al., caches form an overlay structure across the Internet, and act as application level routers. An agent caches data passing through it, so that the next request for the same object can be served from a nearby cache. A great deal of object replication can result.

Chan and Tobagi [7] study the tradeoff between the local storage and network channels in distributed servers architecture to offer on-demand video services. However, the approach of replicating popular movies entirely at the local caches does not make efficient use of the storage space.

SplitStream [5] splits multimedia content into different streams and multicasts each stream using a separate tree. SplitStream is built on Pastry [12], a generic peer-to-peer content location and routing system. The route lookup system in Pastry [12] is based on identifiers assigned to nodes and objects. We use name based hashing to map a specific object to next level proxy in virtual hierarchy.

In the Silo [6] architecture a multimedia clip is divided into segments od exponentially increasing size. Initial segments are small in size and are cached with high probability, while later segments are large in size and cached with low probability. This approach is similar to technique used in [16]. Proxies compute a local segment map of the cached segments, exchange it with other peer proxies and create a global segment map which is used in routing requests to peer proxies for missing segments. Although Silo [6] is a decentralized architecture, it depends heavily on message exchanges between cooperating proxies.

3 Hash-Based Virtual Hierarchies

Caching reduces server loads and client latencies, but single caches achieve only moderate hit rates due to limited temporal locality [8]. As shown in [8], hierarchical caching reduces server hot spots from globally popular objects, and improves access latencies by aggregating requests up the hierarchy. However, such aggregation is useful only on a per-object basis.

Statically defined hierarchies [15] aggregate requests and misses for *all* objects as one proceeds up the hierarchy. Interior caches become overloaded since they handle misses for all objects at each child cache. The root is also very likely to become a bottleneck since it experiences the aggregated miss rate for all objects from lower levels.

In contrast, the Hash-based Virtual Hierarchy (HVH) approach uses hashing to define a different hierarchy for each object [17], resulting in uniformity of traffic and processing workloads across the caches. we discuss only the features of this approach relevant to the caching of multimedia streams.

3.1 Hash-based Object Allocation

Hashing was first used in [13] for allocating objects to a cluster of caches $\{C_1, C_2, \cdots, C_n\}$, so that cluster clients could agree on which cache should hold each object O_k, without communicating with each other. A related idea appeared subsequently under the name consistent hashing [10]. In the approach in [13], each client independently computes the series of hash values $H(C_1, O_k), H(C_2, O_k), \cdots, H(C_n, O_k)$, and picks the cache C_j that yields the highest hash value. Since all clients use the same hash function H, they obtain the same hash values, and choose the same C_j independently. Each O_k is therefore cached only at its corresponding C_j, minimizing object duplication in the cluster, and maximizing hit rates. The work in [13] shows that hashing on a combination of object name and cache name is effective, among other things, in addressing the issue of cache failures.

3.2 Skeletons for Hierarchies

Virtual hierarchies are built in HVH using hashing on top of tree structures called *skeletons*. All proxies appear at the leaves of this tree, and each non-leaf node represents a

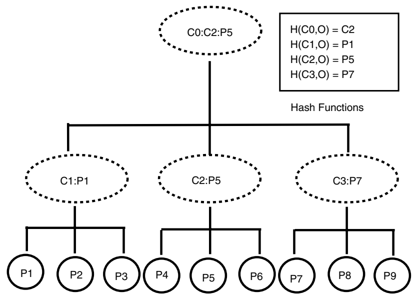

Figure 2. Virtual hierarchy

cluster of nodes at the next lower level (see Figure 2). Clusters can vary in size, and may be defined using any suitable metric, such as hop counts, transfer latencies, or data transfer bandwidth between nodes. Under any chosen metric, proxies within a cluster will be closer to each other than to proxies in other clusters. In Figure 2, the skeleton is rooted at virtual node C_0, and is the parent of sub-clusters C_1,C_2 and C_3.

3.3 Hierarchy Construction

Given a skeleton, a virtual hierarchy is determined as follows. Let a request for an object O arrives at a cache c in cluster C. If c holds O, it directly responds to the request. Otherwise, c acts as a client, and applies the hash function H_C to the members of to own cluster C. A cache is chosen as in Section 3.1, and the request for O forwarded to it. This chosen cache is referred to as the *prime* for object O at cluster C, and denoted by $\Pi_{C,O}$.

Processing continues recursively. If $\Pi_{C,O}$ holds O, it responds to c, which, in turn, caches O, and responds to its own client. If $\Pi_{C,O}$ does not have O, it determines the next-level prime for O by hashing over the nodes in its parent cluster. In the worst case, O will not be cached anywhere in the hierarchy, and the prime at the highest level must forward the request to the remote server.

In Figure 2, for example, P_1, P_5, and P_7 are the primes for object O at clusters C_1, C_2, and C_3, respectively. If a request for O arrives at cache P_2, it first checks its own cache. If O is not found, P_2 applies hash function H_1, and forwards the request to P_1. If P_1 does not have O, it applies the hash function H_0, and descends to C_2, applies hash function H_2 to get to P_5, which is the root prime for O. If O is not in P_5, it must be retrieved from the remote server. The remote server is accessed only when the object is not cached at any of the nodes in the virtual hierarchy, ensuring that the server load is kept low. The structure of the skeleton and the hashing functions can both be stored in a name server and made available to all participating caches.

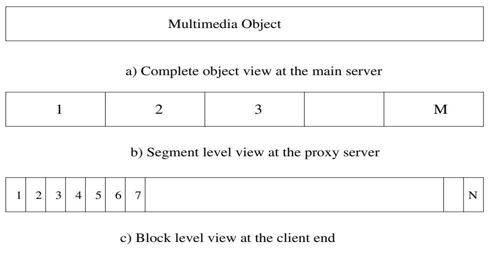
Multimedia Object
a) Complete object view at the main server
1
2
3
M
b) Segment level view at the proxy server
1
2
3
4
5
6
7
N
c) Block level view at the client end

Figure 3. Different views of the same object

4 Our Approach

We use HVH for its load-balancing properties. We also present and evaluate a cache replacement policy suitable for such virtual hierarchy of proxy caches.

4.1 Multiple Access Granularities

In our model, the server, caches, and the client see multimedia streams at different granularities. We define granularity to be the smallest amount of data from a stream that an entity can fetch or manage (see Figure 3). The coarsest granularity is at the main server, which manages entire media objects. A proxy, in contrast, fetches and manages streams at the granularity of *segments*. We use the strategy of dividing multimedia objects into equal-sized segments, since it was observed in [9] that this approach achieves the same performance as exponential sized segmentation [16].

Clients fetch and manage streams at the granularity of *blocks*, the finest granularity. Clients prefetch an entire block, before its playback started. To minimize jitter and delay, a client may prefetch and buffer a small number of blocks, up to some predefined prefetch limit.

4.2 Object Retrieval

We first define some terms. *Segments* in the present discussion are equivalent to *objects* in our presentation of HVH in Section 3. Consequently, a proxy cache may store zero or more segments of a multimedia stream. Because of the way hashing is used in HVH, each proxy serves segments in different roles. Segments for which it generates highest hash value in the cluster are called its *prime segments*. Segments for which it generates the highest hash value in the entire hierarchy are called its *root segments*. Segments that are neither prime nor root for a proxy are called its *alien segments*.

A client sends its request for an object to its designated proxy, negotiates the block size, and requests object blocks sequentially. When a request for a block arrives at the proxy, it first determines the segment number for that block, and returns the block to the client if it holds the segment. Otherwise, the proxy propagates a requrest for the segment up the virtual hierarchy as described in Section 3.3. Since clients request data at a fine granularity, interactivity is easy to implement. A `pause` is easy to realize, since the client stops requesting blocks. For `forward` and `rewind`, a client must request the appropriate block from the proxy it is connected to.

4.3 Cache Replacement

Our replacement policy first calculates a utility value for each cached segment, and replaces segments with the smallest utility value.

Let $S_{i,O}$ be the i^{th} segment of object O. Our utility function for segments is

$$U(S_{i,O}) = \omega(S_{i,O}) * \psi(S_{i,O})$$

where $\omega(S_{i,O})$ is the *weight* of the segment at the cache, and $\psi(S_{i,O})$ is the probability of segment $S_{i,O}$ being accessed at the cache.

The weight of a segment is determined by the number of proxies which can request that segment called as out degree. If d is the outdegree of the virtual hierarchy, then $\omega(seg) = d^k$, where $k = 0$ for alien segments, $k = 1$ for level-1 primes, and so on. In our experiments prime proxies are at level 1 and root proxy is at level 2, but our approach can be easily generalized to deeper hierarchies.

The probability of access $\psi(S_{i,O})$ is computed as follows.

$$\psi(S_{i,O}) = \min\{1, \frac{T_{avg}}{T_r - T_c}\},$$

where T_{avg} is the cumulative average request arrival interval of the segment, T_r is its last reference time, and T_c is the current time. T_{avg} is recomputed as

$$T_{avg}^{new} = \beta * T_{avg}^{old} + (1-\beta)(T_r - T_c),$$

where β is a positive constant less than 1. Proxies calculate utility values of the segments present in the cache and evict the one with lowest utility value.

In our experiments we have used $\beta = 0.5$. While our probability function $\psi()$ is reminiscent of the one used in [16, 9], our utility function is quite different. By multiplying the weight of a segment with its access probability, it assigns higher utility values to prime and popular segments.

4.4 Fault Tolerance and Scalability

When a proxy P_i goes down, request for each segment assigned to it is routed to the proxy that generated the next highest hash value. The randomizing property of HVH causes these reassignments to be evenly distributed among the remaining proxies, preserving the load balancing property of our method.

When a proxy $P_i \in C$ comes back up or when a proxy $P_i \notin C$ is added to the cluster C, then the segments which reassigned to it are exactly those which yield a higher hash value for P_i than any other proxy in the cluster. Thus, HVH ensures that the fewest possible number of segments are reassigned in the case of proxy failure or proxy addition. tolerant and scalable.

Table 1. Comparison with Silo

Parameter	DHCMC	Silo
Number of objects (N)	100 or 1000	100
Cacheable fraction of data	10%–100%	50%–100%
Number of caching proxies	10	100
Segment size	10MB	50MB or more

5 EXPERIMENTS AND RESULTS

Our experiments were designed to evaluate the performance of our caching model (abbreviated as DHCMC) under different scenarios. Our metrics of interest were the byte hit ratios at the caches, the average block latencies, and initial startup delays, since they are good indicators of server load, and the jitter and initial delay observed by clients.

5.1 Simulation Model

Proxies are clustered into groups, and proxies within a cluster are connected by a local or medium area network. Our experiments used a 2-level hierarchy with nodes of outdegree 3, for a total of 9 proxies grouped into 3 clusters. In our simulations, I/O bandwidth for proxies was set to 100 Mbps for all connections, to 10 Mbps between proxy and server, and to 3 Mbps between client and proxy. The block size was 1 MB, so that its play time was 16 seconds at a streaming rate of 0.5 Mbps.

5.2 Performance comparison with Silo

We first compared the performance of our scheme with that of Silo. The parameter values in our experiments are shown in Table 1, which represents the parameters reported in [6] for Silo. The work in [6] reports a system-wide BHR of 85% with 100 caching proxies for a database of 100 multimedia objects. Although [6] do not discuss this issue, each proxy in Silo effectively appears to support only one object. Figure 4 compares the performance of Silo with that of DHCMC. In our first series of experiments, we used the same object and database characteristics reported in [6] to facilitate this comparison. We used 100 objects, each of size

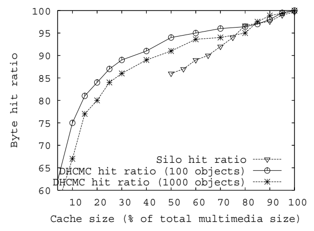

Figure 4. Performance comparison with Silo

Table 2. Simulation parameters

Parameter	Range	Default
# Objects (N)	1000–5000	1000
Zipf parameter	0.4–1.0	0.7
Cacheable fraction of data	1%–50%	10%
Request interval (λ)	2–6 min	3 min
# Blocks per segment	5–10	10
# Proxies in system	4–25	9
Stream bit rate	0.3–1.0 Mbps	0.5
Prefetch limit (blocks)	1–9	No limit

between 1GB–2GB, with a mean size of 1.5GB. We observe that our model achieves the BHR achieved by Silo for much smaller cache sizes.

In our second series of experiments, we increased the number of objects in DHCMC by a factor of 10, to 1000. The performance of our method degrades only slightly, despite the huge increase in the size of the database. These results demonstrate the excellent scalability of our approach.

5.3 Comparison with Middleman

A typical Middleman configuration consists of a number of proxies and a single coordinator which keeps track of proxy contents and makes cache replacement decisions for the entire system [4]. We studied the performance of our our model against Middleman under various system parameters settings. Table 2 summarizes the parameters used in our experiments, their ranges, and their default values. To isolate the effects of various parameters, our experiments varied them one at a time, keeping all others at their default values. However, discussing effect of each parameter on system performance is outside the scope of this paper.

5.3.1 Effects of Inter-arrival Times

We varied the inter-arrival times for object requests from 2–6 minutes. Lower inter-arrival times result in longer queues at proxies, and increase the delay in fetching segments. As Figure 5 shows, average startup latency dropped from 8.5 seconds to 4.6 seconds, and average block latency dropped from 1.3 ms to 0.1 ms in the range studied. The cache hit ratio increased from 86% to 91%.

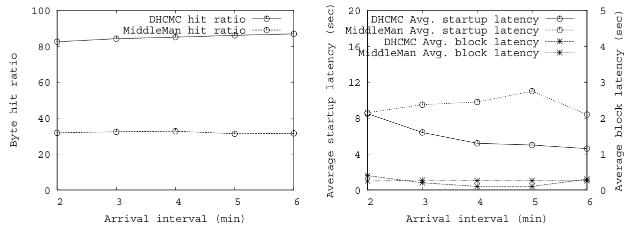

Figure 5. Effects of request arrival interval

5.3.2 Effects of Cache Size

Larger caches allow proxies to store more segments, increase BHR and reduce startup and average block latencies. As Figure 6 shows, we achieve very high BHR (nearly 89%) even when the total size of caches in all proxies is only 10% of the size of the database. In contrast, [4], using centralized cache coordination, reports a BHR of only 77%, with an average cache size of 9% of database size. These results demonstrate that our architecture and cache replacement policy do an excellent job of caching popular objects and segments.

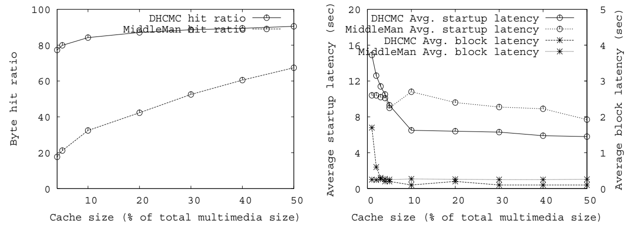

Figure 6. Effects of cache size

5.3.3 Effects of Number of Distinct Objects

As the number of objects increases, client requests are spread over more objects. Caching performance worsens since more unpopular objects are present. In [16], performance degraded drastically, with BHR dropping from 55% to 40% as the number of objects increased from 1000 to 2500. As shown in Figure 7, our approach shows a minor degradation in BHR, from 87.9% to 86.5%, when the number of objects increased from 1000 to 5000. Our caching policies appear to work well.

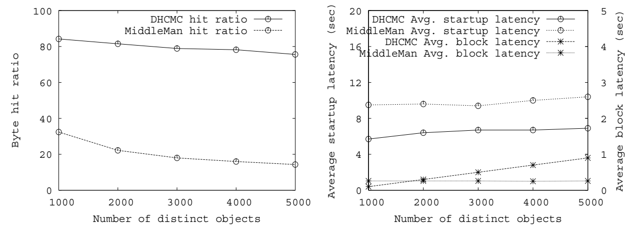

Figure 7. Effects of number of distinct objects

5.3.4 Effects of Number of Proxies

The space at each proxy decreases as the number of proxies increases, but our experiments show that BHR and startup delay remain good, since our popular and initial segments remain cached. The average block latency does increase, as later segments are obtained from other proxies. Figure 8 illustrates this effect.

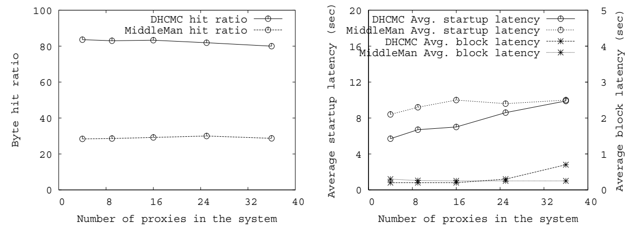

Figure 8. Effects of number of proxies

6 Conclusion And Future Work

Our work is a significant advance over previous approaches since our caching policies are decentralized, and our architecture uses virtual hierarchies for cooperation between proxies in different clusters.

We have shown through simulations with synthetic workloads that our methods achieve a high byte hit ratio at proxies, thus reducing the load on the main server. Our mechanism captures object popularities very effectively, and decreases startup delays greatly. Our mechanism requires small amounts of buffer space to work effectively, making it usable with resource-sensitive thin clients.

We are planning to build a prototype implementation for our caching mechanism to further study system performance. We propose to investigate the behavior and performance of VCR functions such as `forward, pause, rewind` and `stop` in our model.

References

[1] Akamai. http://www.akamai.com.

[2] Mirror image. http://www.mirror-image.com.

[3] Streaming media definition. http://www.webwisdom.com.

[4] S. Acharya and B. Smith. Middleman: A Video Caching Proxy Server. In *Proceedings of the NOSSDAV 2000*, June 2000.

[5] M. Castro, A. Rowston, and P. Druschel. Splitstream: High bandwidth multicast in cooperative environments. *SOSP'03*.

[6] Y. Chae, K. Guo, M. M. Buddhikot, S. Suri, and E. W. Zegura. Silo, Rainbow, and Caching Token: Schemes for Scalable, Fault Tolerant Stream Caching. *IEEE Journal on selected areas in communication*, 20:1328–1344, September 2002.

[7] S. G. Chan and F. Tobagi. Distributed servers architecture for networked video services. *IEEE Transactions on Networking*, 9, June 2000.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *USENIX Annual Technical Conference*, January 1996.

[9] S. Chen, B. Shen, S. Wee, and X. Zhang. Adaptive and Lazy Segmentation Based Proxy Caching for Streaming Media Delivery. In *Proceedings of the 13th international workshop on Networks and operating systems support for digital audio and video*, number 1-58113-694-3, pages 22–31, June 2003.

[10] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of 8th International World Wide Web Conference*, May 1999.

[11] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia Proxy Caching Mechanism for Quality Adaptive Streaming Applications in the Internet. In *Proceedings of IEEE INFOCOM*, number 0-7803-5880-5, March 2000.

[12] A. Rowston and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer sytems. *ACM International Conference on Distributed Systems Platforms*, Nov 2001.

[13] D. G. Thahler and C. V. Ravishankar. Using Name-Based Mapping to Increase Hit Rates. *IEEE/ACM Transactions on Networking*, 6:1–13, February 1998.

[14] D. A. Tran, K. A. Hua, and S. Sheu. A New Caching Architecture for Efficient Video-on-Demand Services on the Internet. In *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)*, January 2003.

[15] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. http://icp.ircache.net/carp.txt.

[16] K. L. Wu, P. S. Yu, and J. L. Wolf. Segment-Based Proxy Caching of Multimedia Streams. In *Proceedings of the tenth international conference on World Wide Web*, number 1-58113-348-0, pages 36–44, May 2001.

[17] Z. Yao, C. V. Ravishankar, and S. Tripathi. Hash-Based Virtual Hierarchies for Caching in Hybrid Content-Delivery Networks. Technical Report 62, UCR, May 2001.